LICS'13, IEEE, pp 358-367. 2013

One-Path Reachability Logic

Grigore Roşu*†, Andrei Ştefănescu*, Ştefan Ciobâcă†, Brandon M. Moore*
*University of Illinois, USA
{grosu, stefane1,bmmoore}@illinois.edu
†University "Alexandru Ioan Cuza", Romania
stefan.ciobaca@info.uaic.ro

***Abstract*—This paper introduces *(one-path) reachability logic*, a language-independent proof system for program verification, which takes an operational semantics as axioms and derives *reachability rules*, which generalize Hoare triples. This system improves on previous work by allowing operational semantics given with *conditional* rewrite rules, which are known to support all major styles of operational semantics. In particular, Kahn's big-step and Plotkin's small-step semantic styles are now supported. The reachability logic proof system is shown sound (i.e., partially correct) and (relatively) complete. Reachability logic thus eliminates the need to independently define an axiomatic and an operational semantics for each language, and the non-negligible effort to prove the former sound and complete w.r.t. the latter. The soundness result has also been formalized in Coq, allowing reachability logic derivations to serve as formal proof certificates that rely only on the operational semantics.**

I. Introduction

This paper is part of our agenda [26]–[28] to prove that *operational semantics*, if used in combination with an appropriate *language-independent* proof system, is sufficient for program verification. No axiomatic (or Hoare), dynamic, or other auxiliary semantics of the same language is needed for verification purposes, the language-independent proof system offers all the good properties of these formalisms, including small size and compositionality of proof derivations.

Operational semantics are easy to define and understand. They can be thought of as formal implementations of the languages they define. For example, a big-step semantics can be thought of as a recursive interpreter, and a small-step semantics as an execution engine describing each computational step that can be performed. Operational semantics typically require little formal training, making them common introductory topics in programming language courses. Moreover, operational semantics scale and, being executable, can be tested against existing implementations for faithfulness or, conversely, yield trusted reference models for the defined languages.

In spite of all the advantages above, operational semantics typically are not used as a basis for program verification. Proofs based on operational semantics tend to be low level and tedious, formalizing and then working directly with the corresponding transition system. Existing approaches for verifying programs, such as Hoare logic or dynamic logic, require (re)defining the language with a set of abstract proof rules, which are often hard to understand and trust, and may additionally require non-trivial program transformations (e.g., to eliminate the side effects from expressions). The state-of-the-art in mechanical verification is to develop and prove such language-specific proof systems sound with respect to more trusted semantics [1], [9], [14], [16], [20], [31]. In our experience defining operational semantics for real languages like C [5], Java (1.4) [6], Verilog [18], etc., the capability to *execute* semantics on thousands of programs (e.g., benchmarks used to test compilers) is a quite effective means to catch semantic errors. It goes without saying that, ideally, we would like a language-independent proof system which takes *any operational semantics as input* and then can derive any properties that can be derived with language-specific proof systems, like Hoare logics or dynamic logics, at the same cost and proof granularity. Such a proof system would make it unnecessary (1) to define multiple semantics for the same language and (2) to give proofs of their equivalence.

In previous work [26]–[28] we showed that the above *is possible*. Specifically, we proposed a language-independent proof system by introducing (unconditional) *reachability rules*, which generalize both term-rewrite rules and Hoare triples. The proof system derives reachability rules which are a consequence of a set of reachability rules, such as the trusted semantics of a language. The verification of a program reduces to checking if the specification (given as a reachability rule) is derivable. However, the existing proof system has two important limitations: (1) it only works with *unconditional* reachability rules, so it only supports operational semantics defined with rules without premises, such as reduction semantics with evaluation contexts [33], the chemical abstract machine [2], or K [25], but it does *not* support two of the most popular operational semantics approaches, namely Kahn's big-step and Plotkin's small-step semantics; and (2) it only derives reachability rules with a "one-path" semantics, that is, the proved property is guaranteed to hold on one execution path of the program only, and not on all paths, thereby capturing partial correctness only for deterministic programs. In this paper we only eliminate limitation (1) above, leaving (2) for future work.

In this paper we overcome limitation (1) above by working with *conditional reachability rules*, which have the form

$$\varphi \Rightarrow \varphi' \text{ if } \varphi_1 \Rightarrow \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow \varphi'_n.$$

The φ are *matching logic patterns* (defined in Section III). An unconditional rule is restricted to the form $\varphi \Rightarrow \varphi'$. Conditions are absolutely necessary in most operational semantics approaches, to express rule premises, such as

$$\langle e_1 + e_2, \sigma\rangle \Rightarrow \langle i +_{Int} j\rangle \text{ if } \langle e_1, \sigma\rangle \Rightarrow \langle i\rangle \wedge \langle e_2, \sigma\rangle \Rightarrow \langle j\rangle$$

in the style of Kahn's big-step [4] semantics, or

$$\langle e_1 + e_2, \sigma\rangle \Rightarrow \langle e'_1 + e_2, \sigma\rangle \text{ if } \langle e_1, \sigma\rangle \Rightarrow \langle e'_1, \sigma\rangle$$

in the style of Plotkin's small-step [23] semantics. By accepting conditional rules as axioms, our approach now supports virtually all popular styles of operational semantics [32]. The users of our approach are therefore not required anymore to follow particular operational styles for defining their languages.

Specifications can be given as unconditional reachability rules. If SUM is the program overwriting s with the sum of all the numbers up to n, then the rule

$$\langle \texttt{SUM}, (\texttt{s} \mapsto s, \texttt{n} \mapsto n) \rangle \wedge n \geq 0 \;\Rightarrow\; \langle (\texttt{s} \mapsto n * (n+1)/2, \texttt{n} \mapsto 0) \rangle$$

states the desired property of SUM using the configuration syntax of a big-step semantics. In words, if we execute SUM in a state binding program variables n and s to integers n and s with $n \geq 0$, then some execution (thus all, as SUM is deterministic) reaches a final state binding program variables n and s to 0 and to the sum of numbers up to n, respectively.

To verify programs under operational semantics requiring conditional reachability rules we give a new proof system for reachability, referred to as *(one-path) reachability logic*. The proof system will be used to derive sequents of the form $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$, where $\varphi \Rightarrow \varphi'$ is a reachability property and $\mathcal{A}$ is a set of possibly-conditional rules giving the operational semantics of the language. The proof system works with more general sequents, of the form $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$, where C is an additional set of unconditional reachability rules. The intuition for such sequents is that the reachability property $\varphi \Rightarrow \varphi'$ holds under the hypotheses $\mathcal{A}$ and C, where the hypotheses in C may only be used after taking a step according to a rule from $\mathcal{A}$. The rules in C are called *circularities*. The desired sequents $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ are recovered when C is empty.

The characteristic rule of our proof systems is the following:

$$\textbf{Circularity}: \quad \frac{\mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'}$$

The Circularity rule allows one to claim the current goal as a new circularity at any point in a derivation. This form of the proof rule was introduced in [26]. In practice Circularity is typically used for code with potentially repetitive behaviors, such as loops, recursive functions, jumps, etc., as a language-independent replacement of the usual language-dependent invariant rules. The new assumption can be used to handle cases where the program may re-execute the same code.

The following proof rule, called Axiom, is the key rule of reachability logic which deals with conditional rules:

$$\textbf{Axiom}: \quad \frac{\begin{array}{c}\varphi \Rightarrow \varphi' \text{ if } \varphi_1 \Rightarrow \varphi'_1 \wedge \cdots \wedge \varphi_n \Rightarrow \varphi'_n \;\in\; \mathcal{A} \\ \psi \text{ is a structureless pattern} \\ \mathcal{A} \cup C \vdash_\emptyset \varphi_i \wedge \psi \Rightarrow \varphi'_i \quad \text{for} \quad i \in 1, \ldots, n\end{array}}{\mathcal{A} \vdash_C \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

To show the current goal is a consequence of a conditional rule we must show that the conditions of the rule are all satisfied. The conditions may be satisfied only because of conditions on the logical variables in the initial state, so the subproofs are allowed the additional assumptions in ψ. The circularities are made available because applying the axiom counts as taking a step. In a big step semantics this is the only kind of step other than complete evaluation, and releasing the circularities while verifying conditions is essential for reasoning about loops and other repetitive behavior.

Contributions. This paper makes the following contributions:

1) It introduces the *conditional reachability rule*, to allow capturing the rules of small-step, big-step, and virtually all other styles of operational semantics.
2) It introduces *(one-path) reachability logic*, a proof system for deriving unconditional reachability rules (e.g., ones corresponding to Hoare triples) from a set of conditional reachability rules (e.g., an operational semantics).
3) It proves the proof system *sound*, that is, partially correct. Due to its practical relevance in producing proof objects, this result is also formalized and proved in Coq.
4) It proves the proof system *relatively complete*. This result is significantly more powerful than the relative completeness of Hoare logic, because it is proved once for all languages, rather than separately for each language.

All proofs are included in a companion report [29], which can be found together with all the supporting code and Coq mechanical proofs at http://fsl.cs.uiuc.edu/RL.

Due to space constraints, this paper does not discuss our implementation of reachability logic in the program verifier MatchC (for C). MatchC is based completely on an operational semantics of C. MatchC has efficiently and automatically verified the functional correctness of programs implementing sorting algorithms, AVL trees, the Schorr-Waite graph marking algorithm, etc., and involving arithmetic, heap data structures, I/O, and so on. The URL above provides an online interface to MatchC, together with dozens of examples. MatchC strengthens the motivation for the foundational work in this paper, showing that our overall verification approach based on a language-independent proof system is also practical.

II. Operational Semantics

To give a proof system parameterized over an operational semantics, we must first fix a language for defining operational semantics. A common denominator of typical syntaxes for big-step, small-step, and reduction semantics is *conditional term rewriting*. The general form of a rewrite rule is

$$p \Rightarrow p' \text{ if } b \wedge p_1 \Rightarrow p'_1 \wedge b_1 \wedge \ldots \wedge p_n \Rightarrow p'_n \wedge b_n$$

where each p_i is a pattern and each b_i is a boolean expression over variables bound in earlier patterns. A set of rules generates a transition system which includes a step between a pair of configurations if there is some rule and some environment mapping variables to subterms so that the first term matches p, the second term matches p', all of the b_i are true, and for each $1 \leq i \leq n$ the term obtained by instantiating p_i can take zero or more steps in the transition system to reach p'_i. A rule is called unconditional if $n = 0$, whether or not b is trivial. E.g.,

$$\langle \texttt{if}\ e\ s_1\ s_2, \sigma \rangle \Rightarrow \langle \sigma' \rangle \text{ if } \langle e, \sigma \rangle \Rightarrow \langle i \rangle \wedge i \neq 0 \wedge \langle s_1, \sigma \rangle \Rightarrow \langle \sigma' \rangle$$

handles the semantics of the positive case of `if` in a big-step style, where configurations are pairs $\langle \texttt{code}, \sigma \rangle$ of a statement or expression `code` to evaluate, and a state/store σ.

Conditional term rewriting is sufficient to express every style of operational semantics, perhaps through a translation adding some auxiliary configurations and rules (e.g., to capture the one-step reduction $\Rightarrow^1$). This is covered in detail in [32] for small-step and big-step semantics, reduction semantics [33], the chemical abstract machine [2], and continuation-based semantics [7]. The representations are strongly faithful in the sense that two configurations are related by a single step in the original system iff appropriate injections into the domain of the term rewriting system are related by a single step.

Fig. 1 shows a small-step and a big-step semantics of a simple imperative language, called IMP, using rewrite rules. In the sequel, we will refer to three IMP programs:

```
SUM    ≡  s := 0; while (n > 0) (s := s+n; n := n-1)
SUM'   ≡  s := 0; while (n > 0)  s := s+n
SUM∞   ≡  n := 1; while (n > 0)  s := s+n
```

`SUM` always terminates, `SUM'` only terminates when $\texttt{n} \leq 0$, and $\texttt{SUM}_\infty$ never terminates.

In conclusion, rewrite rules can be used to formally and uniformly define operational semantics. The rest of the paper is dedicated to showing that with an appropriate proof system, a rewrite-based operational semantics is also sufficient to support program reasoning; no other (axiomatic) semantics is needed.

IMP syntax

PVar	::=	identifiers to be used as program variables
Exp	::=	*PVar* | *Int* | *Exp* + *Exp* | ...
Stmt	::=	`skip` | *PVar* := *Exp* | *Stmt* ; *Stmt*
	|	`if` *Exp Stmt Stmt* | `while` *Exp Stmt*

IMP small-step semantics

+$_1$	$\langle e_1 + e_2, \sigma \rangle \Rightarrow^1 \langle e_1' + e_2, \sigma \rangle$ if $\langle e_1, \sigma \rangle \Rightarrow^1 \langle e_1', \sigma \rangle$
+$_2$	$\langle i_1 + e_2, \sigma \rangle \Rightarrow^1 \langle i_1 + e_2', \sigma \rangle$ if $\langle e_2, \sigma \rangle \Rightarrow^1 \langle e_2', \sigma \rangle$
+$_3$	$\langle i_1 + i_2, \sigma \rangle \Rightarrow^1 \langle i_1 +_{Int} i_2, \sigma \rangle$
lookup	$\langle x, \sigma \rangle \Rightarrow^1 \langle \sigma(x), \sigma \rangle$ if $x \in Dom(\sigma)$
asgn$_1$	$\langle x := e, \sigma \rangle \Rightarrow^1 \langle x := e', \sigma \rangle$ if $\langle e, \sigma \rangle \Rightarrow^1 \langle e', \sigma \rangle$
asgn$_2$	$\langle x := i, \sigma \rangle \Rightarrow^1 \langle \texttt{skip}, \sigma[x \leftarrow i] \rangle$ if $x \in Dom(\sigma)$
seq$_1$	$\langle s_1 ; s_2, \sigma \rangle \Rightarrow^1 \langle s_1' ; s_2, \sigma' \rangle$ if $\langle s_1, \sigma \rangle \Rightarrow^1 \langle s_1', \sigma' \rangle$
seq$_2$	$\langle \texttt{skip}; s_2, \sigma \rangle \Rightarrow^1 \langle s_2, \sigma \rangle$
cond$_1$	$\langle \texttt{if } e\ s_1\ s_2, \sigma \rangle \Rightarrow^1 \langle \texttt{if } e'\ s_1\ s_2, \sigma \rangle$ if $\langle e, \sigma \rangle \Rightarrow^1 \langle e', \sigma \rangle$
cond$_2$	$\langle \texttt{if } i\ s_1\ s_2, \sigma \rangle \Rightarrow^1 \langle s_1, \sigma \rangle$ if $i \neq 0$
cond$_3$	$\langle \texttt{if } 0\ s_1\ s_2, \sigma \rangle \Rightarrow^1 \langle s_2, \sigma \rangle$
while	$\langle \texttt{while } e\ s, \sigma \rangle \Rightarrow^1 \langle \texttt{if } e\ (s\texttt{;while } e\ s)\ \texttt{skip}, \sigma \rangle$

IMP big-step semantics

+	$\langle e_1 + e_2, \sigma \rangle \Rightarrow \langle i_1 +_{Int} i_2 \rangle$ if $\langle e_1, \sigma \rangle \Rightarrow \langle i_1 \rangle, \langle e_2, \sigma \rangle \Rightarrow \langle i_2 \rangle$
int	$\langle i, \sigma \rangle \Rightarrow \langle i \rangle$
lookup	$\langle x, \sigma \rangle \Rightarrow \langle \sigma(x) \rangle$ if $x \in Dom(\sigma)$
skip	$\langle \texttt{skip}, \sigma \rangle \Rightarrow \langle \sigma \rangle$
asgn	$\langle x := e, \sigma \rangle \Rightarrow \langle \sigma[x \leftarrow i] \rangle$ if $x \in Dom(\sigma), \langle e, \sigma \rangle \Rightarrow \langle i \rangle$
seq	$\langle s_1; s_2, \sigma \rangle \Rightarrow \langle \sigma_2 \rangle$ if $\langle s_1, \sigma \rangle \Rightarrow \langle \sigma_1 \rangle$, $\langle s_2, \sigma_1 \rangle \Rightarrow \langle \sigma_2 \rangle$
cond$_1$	$\langle \texttt{if } e\ s_1\ s_2, \sigma \rangle \Rightarrow \langle \sigma' \rangle$ if $\langle e, \sigma \rangle \Rightarrow \langle i \rangle, i \neq 0, \langle s_1, \sigma \rangle \Rightarrow \langle \sigma' \rangle$
cond$_2$	$\langle \texttt{if } e\ s_1\ s_2, \sigma \rangle \Rightarrow \langle \sigma' \rangle$ if $\langle e, \sigma \rangle \Rightarrow \langle 0 \rangle$, $\langle s_2, \sigma \rangle \Rightarrow \langle \sigma' \rangle$
while$_1$	$\langle \texttt{while } e\ s, \sigma \rangle \Rightarrow \langle \sigma \rangle$ if $\langle e, \sigma \rangle \Rightarrow \langle 0 \rangle$
while$_2$	$\langle \texttt{while } e\ s, \sigma \rangle \Rightarrow \langle \sigma' \rangle$ if $\langle e, \sigma \rangle \Rightarrow \langle i \rangle$, $i \neq 0$, $\langle \texttt{s; while } e\ s, \sigma \rangle \Rightarrow \langle \sigma' \rangle$

Fig. 1. The IMP language: syntax, a small-step and a big-step operational semantics. The operational semantics contain rewrite rules making use of ordinary first-order variables: $e, e', e_1, e_1', e_2, e_2'$ are variables of sort *Exp*; σ, σ' are variables of sort *State*; i, i_1, i_2 are variables of sort *Int*; x is a variable of sort *PVar*; s, s_1, s_1', s_2 are variables of sort *Stmt*; *code*, *code'* are variables of sort *Exp* or *Stmt*. The underlying mathematical domain is assumed to provide all the needed operations, for example $+_{Int}$, $*_{Int}$, $<_{Int}$, etc., for integers, and $\sigma(x)$, $\sigma[x \leftarrow i]$, $x \in Dom(\sigma)$, etc., for maps.

III. Matching Logic

Traditionally, program logics are deliberately not concerned with low level details about program configurations, those details being almost entirely deferred to operational semantics. This is a lost opportunity, since configurations contain precious information about the structure of the various data in a program's state, such as the heap, the stack, the input, the output, etc. Without direct access to this information, program logics end up having to either encode it by means of sometimes hard to define predicates, or extend themselves in non-conventional ways, or sometimes both. To design a language-independent proof-system, we cannot rely on language-specific logics. Instead, we use *matching logic* over program configurations. Matching logic [26], [30] is a logic suitable for specifying and reasoning about program or system configurations.

Matching logic formulas combine definition of structure and properties. For simplicity, we present it as a methodological fragment of multi-sorted first-order logic, but the ideas may easily be applied in other settings. Matching logic is parametric in a syntax and a model for configurations. Configurations can be as simple as pairs $\langle \texttt{code}, \sigma \rangle$ of code and store as used in Section II, or as complex as that of the C language [5], which contains more than 70 semantic components.

We assume the reader is familiar with basic concepts of first-order logic. Given a *signature* Σ specifying the sorts and arities of the function symbols (constructors or operators) used in configurations, let $T_\Sigma(Var)$ denote the *free* Σ*-algebra* of terms with variables in *Var*. $T_{\Sigma,s}(Var)$ is the set of Σ-terms of sort s. Maps $\rho : Var \to \mathcal{T}$ with $\mathcal{T}$ a Σ-algebra extend uniquely to (homonymous) Σ*-algebra morphisms* $\rho : T_\Sigma(Var) \to \mathcal{T}$. Many mathematical structures needed for language semantics have been defined as Σ-algebras: boolean algebras, natural/integer/rational numbers, lists, sets, bags (or multisets), maps (e.g., for states, heaps), trees, queues, stacks, etc. We refer the reader to the CASL [19] and Maude [3] manuals for examples.

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

Definition 1: [30] A matching logic formula, or a **pattern**, is a first-order logic (FOL) formula extended to allow terms in $T_{\Sigma,Cfg}(Var)$, called **basic patterns**, as predicates. We define the satisfaction $(\gamma,\rho) \models \varphi$ of a pattern φ w.r.t. a pair of a configuration $\gamma \in \mathcal{T}_{Cfg}$ and a valuation $\rho : Var \to \mathcal{T}$ as follows:

$$(\gamma,\rho) \models \exists X\, \varphi \text{ iff } (\gamma,\rho') \models \varphi \text{ for some } \rho' : Var \to \mathcal{T} \text{ with } \rho'(y) = \rho(y) \text{ for all } y \in Var \backslash X$$
$$(\gamma,\rho) \models \pi \text{ iff } \gamma = \rho(\pi), \text{ where } \pi \in T_{\Sigma,Cfg}(Var)$$

Satisfaction of other FOL constructs is defined in the usual way in terms of satisfaction of subformulas against the same environment. A pattern φ is **valid**, written $\models \varphi$, when $(\gamma,\rho) \models \varphi$ for all $\gamma \in \mathcal{T}_{Cfg}$ and all $\rho : Var \to \mathcal{T}$. A matching logic formula ψ is **patternless** iff it contains no basic pattern.

A basic pattern π is satisfied by all the configurations γ that *match* it; the ρ in $(\gamma,\rho) \models \pi$ can be thought of as the "witness" of the matching, and can be further constrained in a pattern. If `SUM` is the IMP program in Section II, then the pattern $\exists s\ (\langle\ \texttt{SUM}, (\texttt{s} \mapsto s,\ \texttt{n} \mapsto n)\ \rangle\ \wedge\ n \geq_{Int} 0)$ matches the configurations with code `SUM` and state binding program variables `s` and `n` respectively to integers s and n with $n \geq_{Int} 0$. We typically use `typewriter` for program variables and *italic* for mathematical variables in *Var*. Pattern reasoning reduces to FOL reasoning in the configuration model $\mathcal{T}$ [26], [30].

Not all patterns are equally meaningful. For example, the pattern *true* is matched by all configurations, the pattern *false* is matched by no configurations, and some patterns are satisfiable only under some valuations ρ. For our subsequent results, we are interested in two classes of patterns:

Definition 2: A pattern φ is **weakly well-defined** iff for any valuation $\rho : Var \to \mathcal{T}$ some configuration $\gamma \in \mathcal{T}_{Cfg}$ validates $(\gamma,\rho) \models \varphi$, and it is **well-defined** iff ρ uniquely determines γ.

For example, all basic patterns π are well-defined, while patterns of the form $\pi_1 \vee \pi_2$ are weakly well-defined.

IV. Conditional Reachability Rules

Unconditional reachability rules were introduced in [28], which showed they can express particular operational semantics that do not require rule premises. They were studied further in [27], which showed they can express the Hoare triples of axiomatic semantics. Here we introduce *conditional* reachability rules, a generalization capturing as special instances the rules used in conventional operational semantics with rule premises.

Definition 3: A **conditional reachability rule** is a sentence

$$\varphi \Rightarrow \varphi' \text{ if } \varphi_1 \Rightarrow \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow \varphi'_n$$

with $n \geq 0$ and with φ, φ', φ_1, φ'_1, ..., φ_n, φ'_n matching logic patterns. We call φ the **left-hand side (LHS)** and φ' the **right-hand side (RHS)** of the rule. A rule is **unconditional** when $n = 0$. A **reachability system** is a set of reachability rules.

A. Operational Semantics using Reachability Rules

As discussed in Section II, in this paper we assume that operational semantics are defined with rewrite rules of the form

$$cfg \Rightarrow cfg' \text{ if } b \wedge cfg_1 \Rightarrow cfg'_1 \wedge b_1 \wedge \ldots \wedge cfg_n \Rightarrow cfg'_n \wedge b_n,$$

which can now be seen as syntactic sugar for reachability rules

$$cfg \wedge b \wedge b_1 \wedge \ldots \wedge b_n \Rightarrow cfg' \text{ if } cfg_1 \Rightarrow cfg'_1 \wedge \ldots \wedge cfg_n \Rightarrow cfg'_n$$

Here the Boolean side conditions have been all conjuncted with the LHS pattern. Recall from Definition 1 that matching logic includes configuration terms as patterns and allows the use of FOL constructs, like conjunction, to build new patterns, so the above is a correct reachability rule, where φ is $cfg \wedge b \wedge b_1 \wedge \ldots \wedge b_n$. For example, the rule $\textbf{cond}_1$ in the big-step semantics of IMP in Fig. 1 is syntactic sugar for the reachability rule

$$\langle \texttt{if}\ e\ s_1\ s_2, \sigma\rangle \wedge i \neq 0 \Rightarrow \langle \sigma'\rangle \text{ if } \langle e,\sigma\rangle \Rightarrow \langle i\rangle \wedge \langle s_1,\sigma\rangle \Rightarrow \langle \sigma'\rangle$$

From here on we assume that a language/calculus/system is defined as a reachability system and, unless otherwise specified, fix an arbitrary reachability system $\mathcal{S}$. It is irrelevant for the subsequent developments whether such rules represent a small-step, a big-step, or any other particular operational semantics.

An operational semantics typically describes program behaviors by generating a transition system over program configurations, which can associate a behavior to any given program in any given state. In some cases, e.g., small-step semantics, the transition system comprises all the atomic computational steps; in other cases, e.g., big-step semantics, the transition system consists of a binary relationship mapping configurations holding (fragments of) programs to their resulting configurations after evaluation. Recall (Definition 1) that matching logic comes equipped with a model of configurations, $\mathcal{T}$. We next show how $\mathcal{S}$ yields a transition system over the configurations of $\mathcal{T}$.

Definition 4: The **transition relation induced by** $\mathcal{S}$, $\to_{\mathcal{S}} \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ (written infix), is the least fixpoint of the following condition: $\gamma \to_{\mathcal{S}} \gamma'$ if there exists a reachability rule

$$\varphi \Rightarrow \varphi' \text{ if } \varphi_1 \Rightarrow \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow \varphi'_n$$

in $\mathcal{S}$ and some valuation $\rho : Var \to \mathcal{T}$ such that:

1) $(\gamma,\rho) \models \varphi$ and $(\gamma',\rho) \models \varphi'$; and
2) for all $\gamma_1, \ldots, \gamma_n \in \mathcal{T}_{Cfg}$ with $(\gamma_i\ ,\rho) \models \varphi_i$ for all $1 \leq i \leq n$ there exist $\gamma'_1, \ldots, \gamma'_n$ with $(\gamma'_i\ ,\rho) \models \varphi'_i$ and $\gamma_i \to^{\star}_{\mathcal{S}} \gamma'_i$ for all $1 \leq i \leq n$ ($\to^{\star}_{\mathcal{S}}$ is the transitive/reflexive closure of $\to_{\mathcal{S}}$).

Then $(\mathcal{T}_{Cfg}, \to_{\mathcal{S}})$ is the **transition system induced by** $\mathcal{S}$.

Intuitively, $\to_{\mathcal{S}}$ is the least relation compatible with all the rules in $\mathcal{S}$, with all rule conditions interpreted as $\to_{\mathcal{S}}$-reachability. The existence of a least fixpoint is guaranteed by the Knaster-Tarski theorem: the set of binary relations on $\mathcal{T}_{Cfg}$ with inclusion forms a complete lattice, and the condition is monotonic.

If $\mathcal{S}$ contains only rewrite rules, that is, rules whose patterns are all basic, then all the configurations γ, γ', γ_1, γ'_1, ..., γ_n, γ'_n in Definition 4 are uniquely determined by ρ, since $(\gamma,\rho) \models \pi$ iff $\gamma = \rho(\pi)$ for any basic pattern π (by Definition 1). In this case, $\to_{\mathcal{S}}$ becomes the usual transition relation induced by a (top-most) term rewrite system ($\mathcal{S}$) on a Σ-algebra ($\mathcal{T}$). For example, if $\mathcal{S}$ is IMP's small-step semantics in Fig. 1 then the following are valid transitions (`LOOP` is the loop of `SUM` in Section II; for notational simplicity, we make no distinction

between ground terms and their interpretation in $\mathcal{T}$):

$$\langle \texttt{SUM}, (\texttt{s} \mapsto 7, \texttt{n} \mapsto 10)\rangle \;\; \to_{\mathcal{S}} \;\; \langle \texttt{LOOP}, (\texttt{s} \mapsto 0, \texttt{n} \mapsto 10)\rangle \;\; \to_{\mathcal{S}}$$
$$\langle \texttt{if (n>0) (s:=s+n;n:=n-1;LOOP) skip}, (\texttt{s} \mapsto 0, \texttt{n} \mapsto 10)\rangle \;\; \to_{\mathcal{S}}$$
$$\langle \texttt{if(10>0) (s:=s+n;n:=n-1;LOOP) skip}, (\texttt{s} \mapsto 0, \texttt{n} \mapsto 10)\rangle \;\; \to_{\mathcal{S}}$$
$$\ldots \;\; \to_{\mathcal{S}} \;\; \langle \texttt{LOOP}, (\texttt{s} \mapsto 10, \texttt{n} \mapsto 9)\rangle \;\; \to_{\mathcal{S}} \;\; \ldots \;\; \to_{\mathcal{S}}$$
$$\langle \texttt{LOOP}, (\texttt{s} \mapsto 55, \texttt{n} \mapsto 0)\rangle \;\; \to_{\mathcal{S}} \;\; \ldots \;\; \to_{\mathcal{S}} \;\; \langle \texttt{skip}, (\texttt{s} \mapsto 55, \texttt{n} \mapsto 0)\rangle$$

In computing the transitions above, we need to go up to 3 nested conditional rules in Definition 4. On the other hand, if $\mathcal{S}$ is the big-step semantics in Fig. 1, then we have

$$\langle \texttt{SUM}, (\texttt{s} \mapsto 7, \texttt{n} \mapsto 10)\rangle \;\; \to_{\mathcal{S}} \;\; \langle \texttt{s} \mapsto 55, \texttt{n} \mapsto 0\rangle$$

in one transition step, but in order to compute that we need to apply more than 40 nested conditional rules.

To define rule validity with the sense of partial correctness we need to say which configurations terminate. In some cases, e.g., small-step semantics, nontermination is captured by the ability to take an infinite sequence of transitions starting with the given configuration; in other cases, e.g., big-step semantics, nontermination is captured by the ability to make an infinite sequence of nested attempts to fulfill conditions of rules while trying to take a step—which is not the same as a stuck configuration which cannot take a step because no rules apply.

We define a novel notion of termination of configurations with respect to $\mathcal{S}$, which captures both cases above. Our definition is based on a preorder on configurations, which will be well-founded under terminating configurations. This order is inspired by quasi-decreasing orders for conditional term rewriting systems [10]. Our definition is also somewhat related to operational termination of conditional term rewrite systems [17], although the latter is a property of a rewrite system as whole, while our notion of termination refers to a particular configuration in a particular model.

Definition 5: Let $(\mathcal{T}_{Cfg}, \succ)$ be the **termination dependence** relation defined as follows:

- $\gamma \succ \gamma'$ if $\gamma \to_{\mathcal{S}} \gamma'$; and
- $\gamma \succ \gamma'$ if there is a rule $\varphi \Rightarrow \varphi'$ if $\varphi_1 \Rightarrow \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow \varphi'_n$ in $\mathcal{S}$, valuation $\rho : Var \to \mathcal{T}$, and index $1 \le i \le n$ so that:
  1) $(\gamma, \rho) \models \varphi$
  2) $(\gamma', \rho) \models \varphi_i$
  3) For each $1 \le j < i$, $\varphi_j \Rightarrow \varphi'_j$ is "strongly ρ-valid": for any γ_j such that $(\gamma_j, \rho) \models \varphi_j$ there exists γ'_j such that $\gamma_j \to^{\star}_{\mathcal{S}} \gamma'_j$ and $(\gamma'_i, \rho) \models \varphi'_j$

$\gamma \in \mathcal{T}_{Cfg}$ **terminates** iff there are no infinite decreasing $\succ$ chains starting at γ; γ **diverges** otherwise. We let $\succeq$ denote the partial order associated to $\succ$, i.e., its reflexive and transitive closure.

Our definition of termination above mimics the application of conditional rules in the configuration model, in that conditions are solved in order and a condition is considered only if all the previous conditions are successfully solved. Taking into account the order of conditions is essential to get the correct notion of termination. If condition 3 were dropped, then any while loop could be said to diverge in the big-step semantics by using the **while**$_2$ rule and recursing into the second condition which executes the body again, without first checking that the test of the loop actually passes. Termination dependence is essential in the proof of soundness, which justifies circularity by well-founded induction on $\succ$ under terminating configurations.

Let us consider IMP again. In Section II we informally claimed that SUM always terminates, SUM' only terminates when $\texttt{n} \le 0$, and $\texttt{SUM}_\infty$ never terminates. We can now make these claims formal. For SUM, we can show that any configuration γ of the form $\langle \texttt{SUM}, \sigma\rangle$ terminates with any of the two semantics in Fig. 1, for any state σ (including σ's which lack s or n). For SUM', any configuration $\langle \texttt{SUM'}, (\texttt{n} \mapsto n, \sigma)\rangle$ with $n \le 0$ terminates in both semantics, whether or not σ binds s. However, our informal claim "SUM' only terminates when $\texttt{n} \le 0$" in Section II was (purposely) imprecise. Indeed, configurations $\langle \texttt{SUM'}, \sigma\rangle$ with n or s undefined in σ also terminate. Finally, our informal claim "$\texttt{SUM}_\infty$ never terminates" was also imprecise for similar reasons. Stated precisely, configurations of the form $\langle \texttt{SUM}_\infty, (\texttt{n} \mapsto n,\ \texttt{s} \mapsto s,\ \sigma)\rangle$ diverge. Interestingly, such configurations diverge for different reasons in the two semantics, descending by the first bullet of Definition 5 in small-step semantics, and by the second in big-step semantics.

Definition 6: A pattern φ **terminates** (resp. **diverges**), written $\mathcal{S} \models \varphi\downarrow$ (resp. $\mathcal{S} \models \varphi\uparrow$), iff for all $\gamma \in \mathcal{T}_{Cfg}$ and for all $\rho : Var \to \mathcal{T}$, if $(\gamma, \rho) \models \varphi$ then γ terminates (resp. diverges).

In the case of IMP with $\mathcal{S}$ either its small-step or its big-step semantics, from the discussion above we can conclude

$\mathcal{S} \models \langle \texttt{SUM}, \sigma\rangle\downarrow$
$\mathcal{S} \models (\langle \texttt{SUM,'}(\texttt{n} \mapsto n, \sigma)\rangle \wedge n \le_{Int} 0 \vee \langle \texttt{SUM,'}\sigma\rangle \wedge (\texttt{n} \notin Dom(\sigma) \vee \texttt{s} \notin Dom(\sigma))\downarrow$
$\mathcal{S} \models (\langle \texttt{SUM'}, (\texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma)\rangle \wedge n >_{Int} 0)\uparrow$
$\mathcal{S} \models \langle \texttt{SUM}_\infty, (\texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma)\rangle\uparrow$

B. Validity and Well-Definedness

In Hoare logic, $\{pre\}\,\texttt{code}\,\{post\}$ is (semantically) valid, in the sense of partial correctness, iff for any state satisfying *pre*, if code terminates then the resulting state satisfies *post*. This elegant definition has the luxury of relying on another formal semantics of the target language which provides the language-specific notions of "state", "satisfaction", and "termination". Since here everything happens in a single language-independent framework, we generalize the notion of validity as follows:

Definition 7: Given valuation $\rho : Var \to \mathcal{T}$, an unconditional reachability rule $\varphi \Rightarrow \varphi'$ is ρ**-valid**, written $\mathcal{S}, \rho \models \varphi \Rightarrow \varphi'$, iff for any $\gamma \in \mathcal{T}_{Cfg}$ with $(\gamma, \rho) \models \varphi$, if γ terminates then there is a $\gamma' \in \mathcal{T}_{Cfg}$ such that $(\gamma', \rho) \models \varphi'$ and $\gamma \to^{\star}_{\mathcal{S}} \gamma'$. Rule $\varphi \Rightarrow \varphi'$ is **valid**, written $\mathcal{S} \models \varphi \Rightarrow \varphi'$, iff it is ρ-valid for each $\rho : Var \to \mathcal{T}$.

Intuitively, $\mathcal{S} \models \varphi \Rightarrow \varphi'$ specifies reachability: any terminating configuration matching φ transits, on some execution path, to a configuration matching φ'. This notion of validity becomes the usual Hoare logic validity when the reachability rule $\varphi \Rightarrow \varphi'$ corresponds to a Hoare triple as described in Section IV-C and $\mathcal{S}$ is deterministic. Both IMP definitions in Fig. 1 are deterministic. A major difference between our validity and Hoare validity is that the language-specific "state" and "code" are replaced by the language-independent "configuration".

Recall that $\mathcal{S}$ is an arbitrary reachability system, thought of as a "semantics". However, not all reachability systems are meaningful as semantics in all situations. Consider a reachability system containing a rule of the form $\varphi \Rightarrow \mathit{false}$. Such a rule is semantically useless (because it generates no transitions), but also makes reachability reasoning unsound,

because there are no transitions in the generated transition system which would validate $\varphi \Rightarrow \mathit{false}$. Some of the subsequent results require that $\mathcal{S} \models \mu$ for any unconditional $\mu \in \mathcal{S}$, which can be ensured by simple conditions on $\mathcal{S}$ such as:

Definition 8: Rule $\varphi \Rightarrow \varphi'$ if $\varphi_1 \Rightarrow \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow \varphi'_n$ is **(weakly) well-defined** iff φ', φ_1, ..., φ_n are (weakly) well-defined. $\mathcal{S}$ is **(weakly) well-defined** iff all its rules are.

Rules of the form $\varphi \Rightarrow \mathit{false}$ are *not* (weakly) well-defined. Since operational semantics rules contain only configuration terms except possibly for their LHSs (see discussion at beginning of Section IV-A), and since configuration terms are basic patterns, which are always well-defined, it is safe to say that the reachability systems of interest are expected to be well-defined. Nevertheless, weak well-definedness suffices for the soundness of reachability logic, although we need full well-definedness for completeness.

C. Specifying Program Properties using Reachability Rules

Reachability rules can specify not only operational semantics, but also program properties. In fact, each Hoare triple can be translated into a particular reachability rule [27], although the translation needs to be mechanized separately for each language. However, it is *not* recommend to follow this route when specifying program properties, because Hoare triples can be more complex than reachability rules expressing the same property, even without the additional complexity added by the mechanical translation. Consider, for example, the following Hoare triple expressing `SUM`'s property:

$$\{\texttt{n} = \texttt{oldn} \wedge \texttt{n} \geq 0\} \ \texttt{SUM} \ \{\texttt{s} = \texttt{oldn} * (\texttt{oldn} + 1) \,/\, 2 \wedge \texttt{n} = 0\}$$

The introduction of the additional `oldn` variable follows a common Hoare logic "trick" to save the initial value of `n`. Following [27], this Hoare triple translates mechanically into

$$\exists s, n(\langle \texttt{SUM}, (\texttt{s} \mapsto s, \texttt{n} \mapsto n)\rangle \wedge n = \mathit{oldn} \wedge n \geq_{\mathit{Int}} 0) \ \Rightarrow$$
$$\exists s, n(\langle \texttt{skip}, (\texttt{s} \mapsto s, \texttt{n} \mapsto n)\rangle \wedge s = \mathit{oldn} *_{\mathit{Int}} (\mathit{oldn} +_{\mathit{Int}} 1) /_{\mathit{Int}} 2 \wedge n = 0)$$

On the other hand, with the configurations of IMP's big-step semantics in Fig. 1, we can express the same property as:

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

In words, if we execute the configuration holding the program `SUM` and a state binding program variables `s` and `n` to integers s and $n \geq 0$ using IMP's big-step semantics, then we reach a configuration holding a state that binds `s` and `n` to the sum of numbers up to n and to 0, respectively. Technically, s and n are variables of sort *Int*; one can also think of them as "symbolic" integers. On the other hand, `s` and `n` are constants of sort *PVar*.

One could argue that the Hoare triple above is more natural because it is more compact and the FOL specifications make direct use of program's variables. However, one should note that the reachability rule is more informative, since it also states that `s` and `n` must be available in the state before `SUM` is executed. To state these properties using Hoare logic we need additional specification contents, e.g. definedness predicates. Also, Hoare logic conflating program variables (like `s`, `n`) and specification variables (like `oldn`) is often a source of complexity and confusion, particularly in combination with substitution and pointers. Unlike Hoare triples, which only specify properties about final program states, reachability rules can also specify properties of intermediate states as reachability rules where the right hand side has some intermediate code. Hoare triples correspond to reachability rules whose RHS holds the empty code, like the one above. We refer the reader to [27] for more details on the expressiveness of reachability rules.

$$\varphi \Rightarrow \varphi' \text{ if } \varphi_1 \Rightarrow \varphi'_1 \wedge \cdots \wedge \varphi_n \Rightarrow \varphi'_n \in \mathcal{A}$$
$$\psi \text{ is a structureless pattern}$$

Axiom : $$\frac{\mathcal{A} \cup C \vdash_\emptyset \varphi_i \wedge \psi \Rightarrow \varphi'_i \ \text{ for } \ i \in 1, \ldots, n}{\mathcal{A} \vdash_C \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

Reflexivity : $$\mathcal{A} \vdash_\emptyset \varphi \Rightarrow \varphi$$

Transitivity : $$\frac{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_2 \qquad \mathcal{A} \cup C \vdash_\emptyset \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_3}$$

Consequence : $$\frac{\models \varphi_1 \rightarrow \varphi'_1 \quad \mathcal{A} \vdash_C \varphi'_1 \Rightarrow \varphi'_2 \quad \models \varphi'_2 \rightarrow \varphi_2}{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi_2}$$

Case Analysis : $$\frac{\mathcal{A} \vdash_C \varphi_1 \Rightarrow \varphi \qquad \mathcal{A} \vdash_C \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow \varphi}$$

Abstraction : $$\frac{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi' \ \text{ where } \ X \cap FV(\varphi') = \emptyset}{\mathcal{A} \vdash_C \exists X \ \varphi \Rightarrow \varphi'}$$

Circularity : $$\frac{\mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'}$$

Fig. 2. Proof system for (one-path) reachability using conditional rules.

V. Proof System

Figure 2 shows the reachability logic proof system. The target language is given as a weakly well-defined reachability system $\mathcal{S}$. The soundness result (Theorem 1) guarantees that $\mathcal{S} \models \varphi \Rightarrow \varphi'$ if $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ is derivable. Note that the proof system derives more general sequents of the form $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$, 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 C does not appear in a sequent, it means it is empty: $\mathcal{A} \vdash \varphi \Rightarrow \varphi'$ is a shorthand for $\mathcal{A} \vdash_\emptyset \varphi \Rightarrow \varphi'$. Initially, C is empty and $\mathcal{A}$ is $\mathcal{S}$. During the proof, circularities can be added to C via Circularity and flushed into $\mathcal{A}$ by Transitivity or Axiom.

The intuition is that rules in $\mathcal{A}$ can be assumed valid, while those in C have been postulated but not yet justified. After making progress it becomes (coinductively) valid to rely on them. The intuition for sequent $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$, read "$\mathcal{A}$ with circularities C proves $\varphi \Rightarrow \varphi'$", is: $\varphi \Rightarrow \varphi'$ is true if the rules in $\mathcal{A}$ are true and those in C are true after making progress, and if C is nonempty then φ reaches φ' (or diverges) after at least one transition. Let us now discuss the proof rules.

Axiom states that a trusted rule can be used in any *logical frame* ψ. The logical frame is formalized as a patternless formula, as it is meant to only add logical but no structural constraints. Incorporating framing into the axiom rule is

General macros

$\texttt{SUM} \equiv \texttt{s := 0; while (n>0) (s := s+n; n := n-1)}$

$\texttt{S}_1 \equiv \texttt{s := s + n; n := n - 1; LOOP}$

$\texttt{IF} \equiv \texttt{if (n > 0) then } \texttt{S}_1 \texttt{ else skip}$

$\varphi_{\texttt{SUM}} \equiv \langle \texttt{SUM}, (\texttt{s} \mapsto s, \texttt{n} \mapsto n) \rangle \wedge n \geq_{Int} 0$

$\varphi_{\texttt{INV}} \equiv \langle \texttt{LOOP}, (\texttt{s} \mapsto \texttt{sum}_{\texttt{inv}}(n, n'), \texttt{n} \mapsto n') \rangle \wedge n' \geq_{Int} 0$

$\varphi_{\texttt{S}_1} \equiv \langle \texttt{S}_1, (\texttt{s} \mapsto \texttt{sum}_{\texttt{inv}}(n, n'), \texttt{n} \mapsto n') \rangle$

$\texttt{LOOP} \equiv \texttt{while (n>0) (s := s+n; n := n-1)}$

$\texttt{S}_2 \equiv \texttt{n := n - 1; LOOP}$

$\texttt{sum}_{\texttt{inv}}(n, n') \equiv (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1)/_{Int} 2$

$\varphi_{\texttt{IF}} \equiv \langle \texttt{IF}, (\texttt{s} \mapsto \texttt{sum}_{\texttt{inv}}(n, n'), \texttt{n} \mapsto n') \rangle$

$\varphi^{after}_{\texttt{LOOP}} \equiv \langle \texttt{LOOP}, (\texttt{s} \mapsto \texttt{sum}_{\texttt{inv}}(n, n' -_{Int} 1), \texttt{n} \mapsto n' -_{Int} 1) \rangle$

$\varphi_{\texttt{S}_2} \equiv \langle \texttt{S}_2, (\texttt{s} \mapsto \texttt{sum}_{\texttt{inv}}(n, n' -_{Int} 1), \texttt{n} \mapsto n') \rangle$

Small-step macros

$\varphi \equiv \langle \texttt{skip}, (\texttt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int} 2, \texttt{n} \mapsto 0) \rangle$

$\mu \equiv \exists n' \, \varphi_{\texttt{INV}} \Rightarrow \varphi$

Big-step macros

$\varphi \equiv \langle (\texttt{s} \mapsto n *_{Int} (n +_{Int} 1)/_{Int} 2, \texttt{n} \mapsto 0) \rangle$

$\mu \equiv \exists n' \, \varphi_{\texttt{INV}} \Rightarrow \varphi$

Small-step proof derivation ($\mathcal{S}$ is IMP's small-step semantics)

1. $\mathcal{S} \vdash \varphi_{\texttt{SUM}} \Rightarrow \varphi_{\texttt{INV}} \wedge n' =_{Int} n$	[**asgn**$_2$, **seq**$_1$, **seq**$_2$]
2. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{INV}} \Rightarrow \varphi_{\texttt{IF}} \wedge n' \geq_{Int} 0$	[**while**]
3. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{IF}} \wedge n' >_{Int} 0 \Rightarrow \varphi_{\texttt{S}_1} \wedge n' >_{Int} 0$	[**lookup**, **>**$_1$, **>**$_3$, **cond**$_1$, **cond**$_2$]
4. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{S}_1} \wedge n' >_{Int} 0 \Rightarrow \varphi_{\texttt{S}_2} \wedge n' >_{Int} 0$	[**lookup**,**+**$_1$,**+**$_2$,**+**$_3$,**asgn**$_1$,**asgn**$_2$,**seq**$_1$,**seq**$_2$]
5. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{S}_2} \wedge n' >_{Int} 0 \Rightarrow \varphi^{after}_{\texttt{LOOP}} \wedge n' >_{Int} 0$	[**lookup**,**-**$_1$,**-**$_3$,**asgn**$_1$,**asgn**$_2$,**seq**$_1$,**seq**$_2$]
6. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{S}_2} \wedge n' >_{Int} 0 \Rightarrow \exists n' \, \varphi_{\texttt{INV}}$	[**Consequence**(5)]
7. $\mathcal{S} \cup \{\mu\} \vdash \exists n' \, \varphi_{\texttt{INV}} \Rightarrow \varphi$	[μ]
8. $\mathcal{S} \cup \{\mu\} \vdash \varphi_{\texttt{IF}} \wedge n' >_{Int} 0 \Rightarrow \varphi$	[**Transitivity**(3, 4, 6, 7)]
9. $\mathcal{S} \cup \{\mu\} \vdash \varphi_{\texttt{IF}} \wedge n' =_{Int} 0 \Rightarrow \varphi$	[**lookup**, **>**$_1$, **>**$_3$, **cond**$_1$, **cond**$_3$]
10. $\mathcal{S} \cup \{\mu\} \vdash \varphi_{\texttt{IF}} \wedge n' \geq_{Int} 0 \Rightarrow \varphi$	[**Case Analysis**(8, 9)]
11. $\mathcal{S} \vdash_{\{\mu\}} \exists n' \, \varphi_{\texttt{INV}} \Rightarrow \varphi$	[**Transitivity**(2, 10); **Abstraction**]
12. $\mathcal{S} \vdash \exists n' \, \varphi_{\texttt{INV}} \Rightarrow \varphi$	[**Circularity**(11)]
13. $\mathcal{S} \vdash \varphi_{\texttt{SUM}} \Rightarrow \exists n' \, \varphi_{\texttt{INV}}$	[**Consequence**(1)]
14. $\mathcal{S} \vdash \varphi_{\texttt{SUM}} \Rightarrow \varphi$	[**Transitivity**(13, 12)]

Big-step proof derivation ($\mathcal{S}$ is IMP's big-step semantics)

1. $\mathcal{S} \cup \{\mu\} \vdash \exists n' \, \varphi_{\texttt{INV}} \Rightarrow \varphi$	[μ]
2. $\mathcal{S} \cup \{\mu\} \vdash \varphi^{after}_{\texttt{LOOP}} \wedge n' >_{Int} 0 \Rightarrow \varphi$	[**Consequence**(1)]
3. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{S}_2} \wedge n' >_{Int} 0 \Rightarrow \varphi$	[**lookup**, **int**, **-**, **asgn**, **seq**(2)]
4. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{S}_1} \wedge n' >_{Int} 0 \Rightarrow \varphi$	[**lookup**, **int**, **+**, **asgn**, **seq**(3)]
5. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{INV}} \wedge n' >_{Int} 0 \Rightarrow \varphi$	[**lookup**, **int**, **>**, **while**$_2$(4)]
6. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{INV}} \wedge n' =_{Int} 0 \Rightarrow \varphi$	[**lookup**, **int**, **>**, **while**$_1$]
7. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{INV}} \Rightarrow \varphi$	[**Case Analysis**(5, 6)]
8. $\mathcal{S} \vdash_{\{\mu\}} \exists n' \, \varphi_{\texttt{INV}} \Rightarrow \varphi$	[**Abstraction**(7)]
9. $\mathcal{S} \vdash \exists n' \, \varphi_{\texttt{INV}} \Rightarrow \varphi$	[**Circularity**(8)]
10. $\mathcal{S} \vdash \varphi_{\texttt{INV}} \wedge n' =_{Int} n \Rightarrow \varphi$	[**Consequence**(9)]
11. $\mathcal{S} \vdash \varphi_{\texttt{SUM}} \Rightarrow \varphi$	[**int**, **asgn**, **skip**, **seq**(10)]

Fig. 3. Formal reachability logic proofs for SUM. Simple Consequence rules used to perform domain reasoning are elided for readability.

necessary to make logical constraints available while proving the conditions of the axiom hold. Since reachability logic keeps a clear separation between program variables and logical variables the logical constraints are persistent, that is, they do not interfere with the dynamic nature of the operational rules and can therefore be safely used for framing. This is not the case for structural constraints. Consider, for example, a structural constraint given as pattern $\langle \texttt{skip}, \sigma \rangle$. We cannot use this pattern as a frame ψ for the rule **skip** of the big-step semantics of IMP, because $\langle \texttt{skip}, \sigma \rangle \wedge \langle \sigma \rangle$ is matched by no pattern, like *false*, so the proof system would unsoundly derive $\langle \texttt{skip}, \sigma \rangle \Rightarrow \mathit{false}$. Additionally, note that the circularities are released as trusted axioms when deriving the rule's conditions, which is consistent with the intuition above for sequents.

Reflexivity and transitivity correspond to corresponding closure properties of the reachability relation. Reflexivity requires C to be empty to meet the requirement above, that a reachability property derived with nonempty C takes one or more steps. Transitivity releases the circularities as axioms for the second premise, because if there are any circularities to release the first premise is guaranteed to make progress.

Consequence and Case Analysis are adapted from Hoare logic. In Hoare logic Case Analysis is typically a derived rule, but there is no way to derive it language-independently. Ignoring circularities, we can think of these five rules discussed so far as a formal infrastructure for symbolic execution.

Abstraction allows us to hide irrelevant details of φ behind an existential quantifier, which is particularly useful in combination with the next proof rule.

Circularity has a coinductive nature and allows us to make a new circularity claim at any moment. We typically make such claims for code with repetitive behaviors, such as loops, recursive functions, jumps, etc. If we succeed in proving 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 before circularities can be used ensures that only diverging executions can correspond to endless invocation of a circularity.

Fig. 3 shows detailed formal proofs that the SUM program (Section II) indeed calculates the sum of the first n natural numbers in s, for the small-step and big-step semantics of IMP from Fig. 1. In the small-step case (left column) the circularity corresponding to the loop is used via the Transitivity rule, while in the big-step case (right column) the circularity is used via the Axiom rule. Below we discuss these proofs informally.

In small-step, the specification $\varphi_{\text{SUM}} \Rightarrow \varphi$ (sequent 14) is

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

We begin by Transitivity (12,13) through $\exists n' \, \varphi_{\text{INV}}$, with φ_{INV}

$$\langle \texttt{LOOP}, (\texttt{s} \mapsto (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1)/_{Int} 2, \ \texttt{n} \mapsto n') \rangle \wedge n' \geq_{Int} 0.$$

$\varphi_{\text{SUM}} \Rightarrow \exists n' \, \varphi_{\text{INV}}$ (13) holds by running the operational semantics on SUM (1), and abstracting this as $\exists n' \, \varphi_{\text{INV}}$ by Consequence. The property $\mu \equiv \exists n' \, \varphi_{\text{INV}} \Rightarrow \varphi$ (12) is proved by Circularity (from 11). Sequent 11 is proven by using Abstraction to remove the quantifier and fix an arbitrary n', and using Transitivity between 2 and 10. Sequent 2 holds by applying the while rule to unroll the loop into a conditional. This progress releases the circularity μ in 10. We continue by Case Analysis on $n' =_{Int} 0 \vee n' >_{Int} 0$, running the operational semantics in each case (the two cases are described by sequents 8 and 9). When $n' =_{Int} 0$ the goal is reached directly (sequent 9), and when $n' >_{Int} 0$ we reach a configuration implying $\exists n' \, \varphi_{\text{INV}}$ and finish by applying the recently-added axiom μ (sequent 7).

In the big-step case the specification (11) $\varphi_{\text{SUM}} \Rightarrow \varphi$ is now

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

As before, we prove $\mu \equiv \exists n' \, \varphi_{\text{INV}} \Rightarrow \varphi$, with the same φ_{INV} as before. We reach $\exists n' \, \varphi_{\text{INV}}$ from φ_{SUM} by applying the big-step semantics of assignment and sequential composition (10) and then Consequence (9). The difference is that this is reached in a premise of applications of conditional axioms, rather than a premise of Transitivity. Property $\exists n' \, \varphi_{\text{INV}} \Rightarrow \varphi$ is also proved by Circularity (8), but this time the circularity is released (in 1) by applying the **while**$_2$ axiom, and used in one of its conditions.

We take this opportunity to emphasize the verification philosophy in the examples above meets our goals: a property of a program in a particular language, IMP, has been formally derived using only the operational semantics of IMP and a fixed, language-independent seven-rule proof system. No axiomatic semantics and no induction on the program or on its transition system were necessary! Nothing specific to IMP has been done, except applying its operational semantics rules via the Axiom proof rule. The only trusted base for the proofs above is therefore the operational semantics of IMP (provided that the generic proof system is sound, which is shown next). In some sense, the addition of the reachability logic proof system turned the operational semantics of IMP into an axiomatic semantics—at no additional cost. This is the ideal scenario for certifiable program verification. A program verifier for a particular programming language now only needs to implement proof strategies (and possibly syntactic sugar) to conveniently use the generic proof system. This is precisely what our current MatchC program verifier does.

There are two other important practical aspects, which we do not address in depth in this paper. One is the domain (static) reasoning, which is necessary in order to rearrange patterns (via the Consequence rule) so that the operational semantics rules can match and apply. Domain reasoning cannot be avoided by any verification framework, is handled by domain-specific or heuristic techniques in tools, and assumed decidable (via oracles) when proving relative completeness results. We do the same. Another is proof compositionality. As some languages have non-compositional semantics, reachability logic cannot guarantee compositionality of proofs (imagine for example a construct returning the size of the remaining program). Instead, it can be achieved methodologically for languages with compositional semantics, as an instance of the transitivity rule. This is discussed in detail in [27], for an unconditional precursor of reachability logic.

VI. Soundness and Relative Completeness

Here we discuss the soundness and relative completeness of our proof system. Unlike the soundness and relative completeness of Hoare logics and dynamic logics, which are shown for each language separately taking into account the particularities of that language, our mission here is much harder: we need to prove the soundness and relative completeness of our proof system for all languages at once.

*Theorem 1 (**Soundness**):* If $\mathcal{S}$ is a weakly well-defined reachability system, then $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ implies $\mathcal{S} \models \varphi \Rightarrow \varphi'$.

Proof Sketch (see companion report[29] for complete details): By well-founded induction (on the termination dependence relation) on terminating configurations $g \in \mathcal{T}_{Cfg}$, using the following claims and generalization of validity.

Definition 9: Let $g \in \mathcal{T}_{Cfg}$ be any configuration. Unconditional reachability rule $\varphi \Rightarrow \varphi'$ is $(g, \geq)$-**strongly-valid** (resp. $(g, \geq)$-**strictly-strongly-valid**), written $\mathcal{S} \models^{\circledast}_{g \geq} \varphi \Rightarrow \varphi'$ (resp. $\mathcal{S} \models^{+}_{g \geq} \varphi \Rightarrow \varphi'$), if for all γ with $g \geq \gamma$ and all ρ with $(\gamma, \rho) \models \varphi$, there is a γ' such that $\gamma \rightarrow^{\star}_{\mathcal{S}} \gamma'$ (resp. $\gamma \rightarrow^{+}_{\mathcal{S}} \gamma'$) and $(\gamma', \rho) \models \varphi'$.

Intuitively, "$(g, \geq)$-strongly-valid" is similar to "strongly valid", but only concerns configurations less than g, according to the termination dependence relation. If g terminates, then "$(g, \geq)$-strongly-valid" is similar to "valid". The following claim (Proposition 4 in [29]) captures the link between the two:

Claim 1: $\mathcal{S} \models \varphi \Rightarrow \varphi'$ iff $\mathcal{S} \models^{\circledast}_{g \geq} \varphi \Rightarrow \varphi'$ for all terminating g.

To prove Theorem 1, we generalize the induction hypothesis:

Claim 2: For any proof tree concluding $\mathcal{A} \vdash_C \varphi \Rightarrow \varphi'$, for all terminating configurations $g \in \mathcal{T}_{Cfg}$, if the conditional rules in $\mathcal{A}$ are weakly well-defined, if the unconditional rules in $\mathcal{A}$ are $(g, \geq)$-strictly-strongly-valid and if C is $(g_0, \geq)$-strictly-strongly-valid for all g_0 such that $g > g_0$, we have that:

1) if $C = \emptyset$ then $\varphi \Rightarrow \varphi'$ is $(g, \geq)$-strongly-valid and
2) if $C \neq \emptyset$ then $\varphi \Rightarrow \varphi'$ is $(g, \geq)$-strictly-strongly-valid.

Claim 2, proved as Lemma 1 in [29], follows by induction on the proof tree with case analysis on the last rule. For the Circularity case, an inner induction on the termination proof of g is performed. Theorem 1 follows from these claims. ■

Because of the utmost importance of the result above, we have also mechanized its proof. Our complete Coq formalization can be found at http://fsl.cs.uiuc.edu/RL. The proof is parametric in the operational semantics $\mathcal{S}$ and thus can be used to produce formal correctness certificates for program verification tasks. The URL also includes several derived proof rules which are useful for verifying programs, together with their soundness proofs, such as weakening, logic framing, set circularity, and substitution. Set circularity allows introducing several circularities in advance, rather than just one, and is at the core of the MatchC prover, where is useful for proving properties about mutually recursive functions.

We next show that reachability logic is relatively complete: any valid reachability property of any program in any language with an operational semantics given as a reachability system $\mathcal{S}$ is formally derivable with the proof system in Fig. 2 with the rules in $\mathcal{S}$ as axioms. Like in Hoare and dynamic logics, relativity refers to the fact that 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 is that Circularity is sufficient to derive any repetitive behavior occurring in any programs written in any languages! We can only afford to give a high-level sketch of the proof here, as the complete proof with all the details, which can found in the companion technical report [29], is itself the size of this paper.

The main difficulty in proving our result comes from dealing with conditions. One problem arises from the combination of

divergence and conditions. We can easily express divergence: $\mathcal{S} \models \varphi\!\uparrow$ iff $\mathcal{S} \models \varphi \Rightarrow \mathit{false}$. However, in order to derive corresponding sequents, we need to refer to divergence directly. Let us construct the ω-closure of $\mathcal{S}$, $\mathcal{S}^\omega$, as follows: (1) add to $\mathcal{T}_{Cfg}$ a new constant ω; (2) add to $\mathcal{S}$ a new rule $\omega \Rightarrow \omega$; and (3) for each rule $\varphi \Rightarrow \varphi'$ if $\varphi_1 \Rightarrow \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow \varphi'_n$ in $\mathcal{S}$ and each $1 \leq i \leq n$, add to $\mathcal{S}$ a conditional reachability rule

$$\varphi \Rightarrow \omega \text{ if } \varphi_1 \Rightarrow \varphi'_1 \wedge \ldots \wedge \varphi_{i-1} \Rightarrow \varphi'_{i-1} \wedge \varphi_i \Rightarrow \omega.$$

The ω-closure operation does not affect well-definedness: $\mathcal{S}$ is (weakly) well-defined iff $\mathcal{S}^\omega$ is (weakly) well-defined. It also has no semantic effect: $\mathcal{S} \models \varphi \Rightarrow \varphi'$ iff $\mathcal{S}^\omega \models \varphi \Rightarrow \varphi'$. It only ensures that we can prove divergence: $\mathcal{S} \models \varphi\!\uparrow$ iff $\mathcal{S}^\omega \vdash \varphi \Rightarrow \omega$. Our approach to divergence is reminiscent of the one in [15], except that instead of using coinduction directly, we use our proof system (particularly the Circularity proof rule).

We establish the relative completeness of reachability logic under the following assumptions: (1) the reachability system $\mathcal{S}$ is well-defined; (2) $\mathcal{S}$ is ω-closed; (3) the termination dependence relation $\succ$ is finitely branching; (4) the set of configurations $\mathcal{T}_{Cfg}$ is countable; and (5) the model $\mathcal{T}$ includes natural numbers with addition and multiplication. Assumptions (1) and (3) allow the encoding of finite and infinite transition sequences into FOL formulas. Assumption (4) allows the encoding of sequences of configurations into sequences of natural numbers. Assumption (5) is a standard assumption (made also by Hoare and dynamic logic completeness results) which allows the expressing of Gödel's β predicate. We expect the operational semantics of any reasonable language to satisfy these conditions. Formally, we have the following

*Theorem 2 (**Relative Completeness**):* Under the above five assumptions, if $\mathcal{S} \models \varphi \Rightarrow \varphi'$ then $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$.

Proof Sketch (see companion report[29] for complete details): Our proof relies on the fact that pattern reasoning in first-order matching logic reduces to FOL reasoning in the configuration model $\mathcal{T}$. Let $\Box$ be a fresh configuration variable. For a pattern φ, let $\varphi^\Box$ be the FOL formula obtained by replacing each basic pattern π with an equality $\Box = \pi$. Further, for $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : \mathit{Var} \to \mathcal{T}$, let ρ^γ be the valuation extending ρ by mapping $\Box$ into γ. Then $(\gamma, \rho) \models \varphi$ iff $\rho^\gamma \models \varphi^\Box$, and from now on we use φ to unambiguously refer to both the matching logic pattern and its FOL equivalent. Also, $\varphi[c/\Box]$ stands for the patternless formula obtained from φ by substituting $\Box$ with the variable c.

A key component of the proof is the encoding of a finite transition sequence, namely $\mathit{path}(c, c')$, in FOL, in the context of conditional rules. To achieve that, we use Gödel's β function to existentially quantify over the configurations occurring in the sequence, and in the conditions of the rules applied in the sequence, and in the conditions of the rules applied in the conditions, and so on. The configuration variables are indexed by integers according to a tree-like scheme which is parametric in the maximum sequence length and the maximum condition nesting. Then we use β to encode the quantification over a sequence into quantification over two integer variables, thereby expressing $\mathit{path}(c, c')$ in FOL. The formal definition is quite involved, and due to space constrains is not given here.

$$\begin{aligned}
\mathit{step}(c, c') \equiv\ & \exists c_1 \ldots c_{maxn}\ \exists c'_1 \ldots c'_{maxn} \exists \bar{x}\ (\bigvee_{\mu \in \mathcal{S}} (\varphi[c/\Box] \wedge \varphi'[c'/\Box] \\
& \wedge \bigwedge_{1 \leq i \leq n} (\varphi_i[c_i/\Box] \wedge \varphi'_i[c'_i/\Box] \wedge \mathit{path}(c_i, c'_i))))\\
\mathit{succ}(c, c') \equiv\ & \mathit{step}(c, c') \\
& \vee\ \exists c_1 \ldots c_{maxn}\ \exists c'_1 \ldots c'_{maxn} \exists \bar{x}\ (\bigvee_{\mu \in \mathcal{S}} (\varphi[c/\Box] \wedge \bigvee_{1 \leq i \leq n} (\varphi_i[c'/\Box] \\
& \wedge \bigwedge_{1 \leq j < i} (\varphi_j[c_j/\Box] \wedge \varphi'_j[c'_j/\Box] \wedge \mathit{path}(c_j, c'_j)))))\\
\mathit{diverge}(c) \equiv\ & \forall m \exists c_0 \ldots c_m (\bigwedge_{0 \leq i < m} \mathit{succ}(c_i, c_{i+1}) \wedge c_0 = c)\\
\mathit{coreach}(\varphi) \equiv\ & \exists c \exists c'\ (c = \Box \wedge \varphi[c'/\Box] \wedge \mathit{path}(c, c'))
\end{aligned}$$

Fig. 4. FOL encodings for properties of the transition system

Using $\mathit{path}(c, c')$ we can encode in FOL the following: (1) $\mathit{step}(c, c')$, the one step transition relation ($\rightarrow_{\mathcal{S}}$), (2) $\mathit{succ}(c, c')$, the termination dependence relation ($\succ$), (3) $\mathit{diverge}(c)$, the divergence predicate ($\uparrow$), and (4) $\mathit{coreach}(\varphi)$, the configurations reaching some formula φ. These encodings are shown in Fig. 4, with μ the rule $\varphi \Rightarrow \varphi'$ if $\varphi_1 \Rightarrow \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow \varphi'_n$ in $\mathcal{S}$.

Next, we can use the definitions above to encode the semantic validity of reachability rules as FOL validity: $\mathcal{S} \models \varphi \Rightarrow \varphi'$ iff $\models \varphi \rightarrow \exists c\, (\Box = c \wedge \mathit{diverge}(c)) \vee \mathit{coreach}(\varphi')$. Then the theorem follows by using Consequence and Case Analysis from sequents $\mathcal{S} \vdash \Box = c \wedge \mathit{diverge}(c) \Rightarrow \varphi$ and $\mathcal{S} \vdash \mathit{coreach}(\varphi) \Rightarrow \varphi$. These sequents are derived using Circularity, with $\mathcal{S}$ being ω-closed playing a crucial role in the derivation of the former. ■

VII. Related work

We fully share the goal of the unified theory of programming initiative [13] and of the mechanical verification community to reduce the correctness of program verification to a trusted formal semantics of the target language [1], [9], [14], [16], [20], [31], although our methods are different. Instead of a framework to ease the task of giving multiple semantics of the same language and proving systematic relationships between them, we advocate developing frameworks which eliminate the task by requiring only one semantics (which is operational), and offering an underlying theory with the necessary machinery to achieve the benefits of multiple semantics without the costs.

Regarding a program as a forwards specification transformer goes back to Floyd [8]. However, his rules are not concerned with structural configurations, are not meant to be operational, and introduce quantifiers. Similar ideas have been used in equational algebraic specifications of programming languages [11] and in evolving specifications [22]. Conceptually, what distinguishes our approach from these is the use of matching logic to specify configurations of interest by means of patterns, which give access to all the structural details. The use of variables in patterns offers a comfortable level of abstraction by mentioning in each rule only the necessary configuration components.

Dynamic logic [12] adds modal operators to FOL to embed program fragments within specifications. For example, $\psi \rightarrow [\mathtt{s}]\psi'$ means "after executing $\mathtt{s}$ in a state satisfying ψ, a state may be reached which satisfies ψ'". In matching logic, programs and specifications also coexist in the same logic, but we use it only for expressing static properties. We express

the dynamic properties using reachability logic which consists of only language-independent proof rules and works with any operational semantics, unlike dynamic logic which still requires language-specific proof rules (e.g., invariant rules).

Separation logic [21], [24] has been proposed as a state specification formalism to enhance Hoare logic in the presence of heaps. While reachability logic is an alternative to, rather than an extension of Hoare logic, one could use separation logic as a pattern specification formalism. This is not precluded by presenting our results in terms of matching logic, as separation logic has been shown to be a matching logic instance when the configuration model is chosen to be that of heaps [26]. However, operational semantics of complex languages require many configuration components besides the heap, and matching logic works with arbitrarily complex configurations by design.

Language-independent proof systems: Matching logic and a first proof system were presented in [30]. However, that approach required a specialized proof system for each target language. A first sound language-independent proof system for unconditional rules can be found in [28], and [27] gives a mechanical and compositional translation of Hoare logic proof derivations for IMP into derivations in the proof system. Finally, [26] gives a different proof system, introducing the Circularity proof rule. All the previous language-independent proof systems and results could only be applied to operational semantics given in terms of unconditional rules, so the most popular styles of operational semantics were excluded.

VIII. Conclusion

We presented *(one-path) reachability logic*, a novel framework for reasoning about reachability which unifies operational and axiomatic semantics. Its sentences, the *reachability rules*, can express both transitions between configurations, as needed for operational semantics, and Hoare-style triples, as needed for axiomatic semantics. A programming language is given as a set of reachability rules defining its operational semantics, and a sound and relatively complete language-independent seven-rule proof system then can derive any reachability property of the language. The soundness result was also mechanized in Coq, which lets reachability logic proofs serve as proof certificates.

Until now, reachability rules were unconditional, so they could only express a few styles of operational semantics. By allowing *conditional* rules, reachability logic can now express virtually all operational semantics styles, including the standard small-step (by Plotkin) and big-step (by Kahn) semantics which were excluded before.

We hope reachability logic may serve as a new foundation for verifying programs. Also, since Hoare-style proof derivations can be mechanically translated into reachability logic proof derivations, reachability logic may also serve as a more elegant means to establish soundness of axiomatic semantics.

Acknowledgements: We thank Chucky Ellison and the anonymous reviewers for their valuable comments on drafts of this paper. This work is partially funded by NSA contract H98230-10-C-0294, NSF contract CCF-1218605, DARPA HACMS, and Romanian SMIS-CSNR 602-12516 contract 161/15.06.2010.

References

[1] A. W. Appel, "Verified software toolchain," in *ESOP*, ser. LNCS, vol. 6602. Springer, 2011, pp. 1–17.
[2] G. Berry and G. Boudol, "The chemical abstract machine," *Theoretical Computer Science*, vol. 96, no. 1, pp. 217–248, 1992.
[3] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott, *All About Maude*, ser. LNCS. Springer, 2007, vol. 4350.
[4] D. Clément, J. Despeyroux, L. Hascoet, and G. Kahn, "Natural semantics on the computer," in *Proceedings of the France-Japan AI and CS Symposium*. ICOT, Japan, 1986, pp. 49–89.
[5] C. Ellison and G. Roşu, "An executable formal semantics of C with applications," in *POPL*. ACM, 2012, pp. 533–544.
[6] A. Farzan, F. Chen, J. Meseguer, and G. Roşu, "Formal analysis of Java programs in JavaFAN," in *CAV*, ser. LNCS, vol. 3114. Springer, 2004, pp. 501–505.
[7] M. Felleisen and D. P. Friedman, "Control operators, the SECD-machine, and the lambda-calculus," in *3rd Working Conference on the Formal Description of Programming Concepts*. IFIP, Aug. 1986, pp. 193–219.
[8] R. W. Floyd, "Assigning meaning to programs," in *Symposium on Applied Mathematics*, vol. 19. A.M.S., 1967, pp. 19–32.
[9] C. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen, *The RAISE Development Method*. Prentice Hall, 1995.
[10] J. Giesl and T. Arts, "Verification of Erlang processes by dependency pairs," *AAECC*, vol. 12, no. 1/2, pp. 39–72, 2001.
[11] J. Goguen and G. Malcolm, *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.
[12] D. Harel, D. Kozen, and J. Tiuryn, "Dynamic logic," in *Handbook of Philosophical Logic*, 1984, pp. 497–604.
[13] C. A. R. Hoare and H. Jifeng, *Unifying Theories of Programming*. Prentice Hall, 1998.
[14] B. Jacobs, "Weakest pre-condition reasoning for Java programs with JML annotations," *Journal of Logic & Algebraic Programming*, vol. 58, no. 1-2, pp. 61–88, 2004.
[15] X. Leroy and H. Grall, "Coinductive big-step operational semantics," *Information & Computation*, vol. 207, no. 2, pp. 284–304, 2009.
[16] H. Liu and J. S. Moore, "Java program verification via a JVM deep embedding in ACL2," in *TPHOLs'04*, ser. LNCS, vol. 3223, pp. 184–200.
[17] S. Lucas, C. Marché, and J. Meseguer, "Operational termination of conditional term rewriting systems," *Information Processing Letters*, vol. 95, no. 4, pp. 446–453, 2005.
[18] P. O. Meredith, M. Katelman, J. Meseguer, and G. Roşu, "A formal executable semantics of Verilog," in *MEMOCODE*. IEEE, 2010, pp. 179–188.
[19] P. D. Mosses, *CASL Reference Manual*. Springer, 2004.
[20] T. Nipkow, "Winskel is (almost) right: Towards a mechanized semantics textbook," *Formal Aspects of Computing*, vol. 10, pp. 171–186, 1998.
[21] P. W. O'Hearn and D. J. Pym, "The logic of bunched implications," *Bulletin of Symbolic Logic*, vol. 5, no. 2, pp. 215–244, 1999.
[22] D. Pavlovic and D. R. Smith, "Composition and refinement of behavioral specifications," in *ASE*, 2001, pp. 157–165.
[23] G. Plotkin, "A structural approach to operational semantics," *Journal of Logic & Algebraic Programming*, vol. 60-61, pp. 17–139, 2004.
[24] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *LICS*. IEEE, 2002, pp. 55–74.
[25] G. Roşu and T.-F. Şerbanuţă, "An overview of the K semantic framework," *Journal of Logic & Algebraic Programming*, vol. 79, pp. 397–434, 2010.
[26] G. Roşu and A. Ştefănescu, "Checking reachability using matching logic," in *OOPSLA'12*. ACM, 2012, pp. 555–574.
[27] ——, "From Hoare logic to matching logic reachability," in *FM'12*, ser. LNCS, vol. 7436. Springer, 2012, pp. 387–402.
[28] ——, "Towards a unified theory of operational and axiomatic semantics," in *ICALP'12*, ser. LNCS, vol. 7392. Springer, 2012, pp. 351–363.
[29] G. Roşu, A. Ştefănescu, Ştefan Ciobâcă, and B. M. Moore, "Reachability logic," UIUC, Tech. Rep. http://hdl.handle.net/2142/32952, Jul 2012.
[30] G. Roşu, C. Ellison, and W. Schulte, "Matching logic: An alternative to Hoare/Floyd logic," in *AMAST*, ser. LNCS, vol. 6486, 2010, pp. 142–162.
[31] R. Sasse and J. Meseguer, "Java+ITP: A verification tool based on Hoare logic and algebraic semantics," *ENTCS*, vol. 176, no. 4, pp. 29–46, 2007.
[32] T.-F. Şerbanuţă, G. Roşu, and J. Meseguer, "A rewriting logic approach to operational semantics," *Information & Computation*, vol. 207, no. 2, pp. 305–340, 2009.
[33] A. Wright and M. Felleisen, "A syntactic approach to type soundness," *Information & Computation*, vol. 115, no. 1, pp. 38–94, 1994.