Low-Level Program Verification using Matching Logic Reachability

Dwight Guth Andrei Ştefănescu Grigore Roşu
University of Illinois at Urbana-Champaign
{dguth2, stefane1, grosu}@illinois.edu

Abstract

Matching logic reachability is an emerging verification approach which uses a language-independent proof system to prove program properties based on the operational semantics. In this paper we apply this approach in the context of a low-level real-time language with interrupts, in which each instruction takes a specified time to execute. In particular, we verify that if the interrupts are scheduled with large enough intervals, the program execution terminates yielding the correct result. Surprisingly, it turns out that matching logic reachability can handle the low-level and real-time features of the language just by using their operational semantics, and that language specific reasoning is unnecessary.

1 Matching Logic Reachability

In this section we give background on matching logic reachability [6–10]. The matching logic reachability approach to program verification is to directly use the operation semantics of the target language, together with an appropriate language-independent proof system, in order to prove that a program meets its specifications. Both the operational semantics and the program specifications are given as reachability rules. 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.

1.1 Matching Logic

Here we briefly recall matching logic [10], 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 structure (via basic patterns) and logical constraints: a configuration satisfies the pattern iff it matches the structure and satisfies the constraints.

Matching logic is parametric in a signature and a model of configurations, making it a prime candidate for expressing program state properties in a language-independent verification framework. The configuration signature can be as simple as a pair $\langle \mathsf{code}, \sigma \rangle$ with code a program fragment and σ a "state" mapping program variables into integers, in the case when one wants to reason about simple imperative languages. It can also be as complex as that of the C language [3], which contains more than 70 semantic components.

We use basic concepts of multi-sorted 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. A valuation $\rho : Var \to \mathcal{T}$ with $\mathcal{T}$ a Σ-algebra extends uniquely to a (homonymous) Σ*-algebra morphism* $\rho : T_\Sigma(Var) \to \mathcal{T}$. 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 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, called *configurations*.

Definition 1. *[10] A matching logic formula, or a* ***pattern****, is a first-order logic (FOL) formula which allows terms in $T_{\Sigma,Cfg}(Var)$, called* ***basic patterns****, as predicates. We define satisfaction $(\gamma,\rho) \models \varphi$ over configurations $\gamma \in \mathcal{T}_{Cfg}$, valuations $\rho : Var \to \mathcal{T}$ and patterns φ as follows (among the FOL constructs, we only show $\exists$):*

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

$$(\gamma,\rho) \models \pi \qquad \textit{iff} \quad \gamma = \rho(\pi) \qquad \textit{where } \pi \in \mathcal{T}_{\Sigma,Cfg}(Var)$$

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

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. For example, if `SUM` is the code "`s:=0; while(n>0)(s:=s+n; n:=n-1)`" in a simple imperative language with configurations $\langle \texttt{code}, \sigma \rangle$, 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` to integer s and non-negative integer n. Note that we use `typewriter` for program variables in *PVar* and *italic* for mathematical variables in *Var*.

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 the following class of patterns:

Definition 2. *[8] A pattern φ is* ***weakly well-defined*** *iff for any valuation $\rho : Var \to \mathcal{T}$ some configuration $\gamma \in \mathcal{T}_{Cfg}$ validates $(\gamma,\rho) \models \varphi$.*

For example, all basic patterns π are weakly well-defined.

1.2 Conditional Reachability Rules

Unconditional reachability rules were introduced in [8], which shows that they can express particular operational semantics that do not require rule premises, such as evaluation contexts [11] and $\mathbb{K}$ [5]. They were studied further in [7], which shows that they can express the Hoare triples of axiomatic semantics. Finally, conditional reachability rules were introduced as a generalization of unconditional rules in [9], which shows that they capture conventional operational semantics with rule premises, such as big-step [1] and small-step [4].

Definition 3. *[9] 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)****. A rule is* ***unconditional*** *when $n = 0$. A* ***reachability system*** *is a set of reachability rules.*

As discussed at length in [8, 9], 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 $cfg \wedge b \wedge b_1 \wedge \ldots \wedge b_n$. In the particular case of operational semantics with rewrite rules without premises, the associated reachability rules are unconditional.

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, an evaluation contexts, a $\mathbb{K}$ 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, $\mathcal{T}$. Then $\mathcal{S}$ yields a transition system over the configurations of $\mathcal{T}_{Cfg}$.

Definition 4. *[9] 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, $\ldots$, 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 ($\mathcal{T}$).

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.

The notion of termination of configurations with respect to $\mathcal{S}$ given below captures both cases above. This 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. This definition is also somewhat related to operational termination of conditional term rewrite systems, although the latter is a property of a rewrite system as whole, while this notion of termination refers to a particular configuration in a particular model.

Definition 5. *[9] 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 $1 \leq i \leq n$ so that:*
  1. *$(\gamma, \rho) \models \varphi$*
  2. *$(\gamma', \rho) \models \varphi_i$*
  3. *For each $1 \leq j < i$, $\varphi_j \Rightarrow \varphi'_j$ 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.*

This 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 a big-step semantics by recursing into the condition which executes the body again, without first checking that the test of the loop actually passes. Termination dependence is essential for soundness, which justifies circularity by well-founded induction on $\succ$ under terminating configurations.

Definition 6. *[9] 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 Hoare logic, $\{pre\}$ `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 for the language-specific notions of "state", "satisfaction", and "termination". Here everything happens in a single language-independent framework, thus validity is generalized as follows:

Definition 7. *[9] 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 $\gamma \to_{\mathcal{S}}^{\star} \gamma'$. 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 (as described later) and $\mathcal{S}$ is deterministic. The definition of the assembly language in Section 2.1 is 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 $\varphi \Rightarrow \mathit{false}$. The soundness of the proof system in Figure 1 requires that $\mathcal{S} \models \mu$ for any unconditional $\mu \in \mathcal{S}$, which can be ensured by simple conditions on $\mathcal{S}$ as follows:

Definition 8. *[9] 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. As mention in the beginning of Section 1.2, an operational semantics rule contains only configuration terms except possibly for its LHS. Since configuration terms are basic patterns, which are always weakly well-defined, it is safe to say that the reachability systems of interest are expected to be weakly well-defined.

Reachability rules can specify not only operational semantics, but also program properties. As shown in [7], each Hoare triple can be translated into a particular reachability rule, 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 complexity added by the mechanical translation. Consider 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`. 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 we can express the same property more directly as:

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

Axiom : $$\frac{\varphi \Rightarrow \varphi' \text{ if } \varphi_1 \Rightarrow \varphi'_1 \bigwedge \cdots \bigwedge \varphi_n \Rightarrow \varphi'_n \in \mathcal{A} \qquad \psi \text{ is a structureless pattern} \qquad \mathcal{A} \cup C \vdash \varphi_1 \wedge \psi \Rightarrow \varphi'_1 \qquad \cdots \qquad \mathcal{A} \cup C \vdash \varphi_n \wedge \psi \Rightarrow \varphi'_n}{\mathcal{A} \vdash_C \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi}$$

Reflexivity : $\mathcal{A} \vdash \varphi \Rightarrow \varphi$

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

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

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

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

Figure 1: Proof system for (one-path) reachability using conditional rules.

In words, if we execute the configuration holding the program `SUM` and a state binding program variables `s` and `n` to integer s and non-negative integer n, 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 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, pointers, and more complex variable scoping rules. Unlike Hoare triples, which only specify properties about final program states, reachability rules can also specify properties of intermediate states as rules where the RHS has some intermediate code. A Hoare triple corresponds to a reachability rule whose RHS holds the empty code, like the one above.

1.3 Proof System

Figure 1 shows the reachability logic proof system. The target language is given as a weakly well-defined reachability system $\mathcal{S}$. The soundness result in [9] 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 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. In other words, it is safe to add more logical constraints on existing reachability properties, but it is unsafe to add more structural constraints. 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.

2 Low-level verification

Once we have established all the machinery of matching logic reachability, our focus turns toward whether we can use it to prove interesting properties of low-level real-time programming languages. Our goal in this endeavor is to take a trusted operational semantics of a programming language which includes scheduling information such as how long an operation takes to perform, as well as operations which occur at specific points in time, and prove properties of programs written in such a language, *without* further extending the proof system.

2.1 A simple low-level programming language

We consider a simple programming language and we define it $\mathbb{K}$ semantic framework. This programming language is designed to be similar in structure to a simple RISC assembly language. It contains basic arithmetic instructions, basic control flow instructions, a load/store architecture for communicating between registers and memory, and I/O instructions for reading and writing external data which is provided at runtime nondeterministically, and an instruction to schedule timer interrupts.

For simplicity, the language we define has an infinite number of registers and an infinite amount of memory space. In practice, this has no real bearing on the semantics. We could just as easily write a language which uses register or memory locations from a particular finite set. We also require that each label-prefixed block of code end with either a `jmp` or a `halt` instruction. Finally, each instruction is assigned a fixed number of cycles that it takes to execute. A more complex language might have modelled some kind of processor architecture in order to determine how long each instruction takes to execute, but for simplicity we assume here that each instruction always takes the same fixed amount of time (although different instructions can take different amounts of time). The full list of instructions and a simple description of their behavior is given in Table 1.

Of particular interest are the `read`, `write`, `rw`, and `int` instructions which have scheduling considerations. The first three of these are the I/O instructions of the language: `read` takes the name of an external data register and reads its value into an internal register; `write` takes the name of an external data register and writes to it; finally, `rw` performs atomically first a read of an external data register, then a write to the same external data register. The reason these three instructions have scheduling considerations is that the environment the program is executing in can access (for read

Opcode	Description
`add`, `sub`, `mul`, `div`, `or`, `and`, `not`	Arithmetic and bitwise operations, three-argument form. Can take either register or immediate arguments to input. Stores to register.
`load`	Load value into register from memory. Can index memory by register or immediate.
`store`	Store value from register into memory. Can index memory by register or immediate, and can take either register or immediate arguments to input.
`jmp`	Unconditional jump instruction. Takes a program label.
`beq`, `bne`, `blt`, `ble`, `bgt`, `bge`	Conditional branch instructions. Takes two inputs from register or immediate, performs an arithmetic comparison, and conditionally jumps or falls through based on its value.
`int`	Interrupt instruction. Takes two timer parameters and schedules a timer interrupt.
`halt`	Immediately terminate program execution.
`rfi`	Complete interrupt and return to normal mode, or proceed to next, lower-priority interrupt.
`sleep`	Do nothing for a number of cycles expressed as input using a register or an immediate value.
`read`, `write`, `rw`	Perform basic I/O operations.

Table 1: Instructions in the assembly-like language

or write) the values in the external data registers. Thus, depending on how much time has elapsed since the program has begun executing, a single read instruction (in for example a loop) is capable of reading different values at each execution of the instruction. If the environment writes a value to such a register in the middle of an instruction, the value can be read with a `read` instruction as soon as the current instruction has finished executing. In particular, if the environment writes to a register while it is being read, the read will read the old value. If the immediate next instruction in the program is a `write` to the same external data register, the value written by the environment will be overwritten by the program and lost. The atomic `rw` instruction addresses this issue, as it is guaranteed to not overwrite the value on the external data register if it has already been overwritten by the environment while the instruction is executing.

Finally, the `int` instruction implements timer interrupts in the language. For simplicity, the user cannot disable an interrupt once it has been scheduled, and cannot explicitly specify the priority of the interrupt. However, interrupts do have priority based on ascending order of the time in the program's execution that they are scheduled. The `int` instruction takes two timer parameters. The first is the number of cycles after executing the instruction that the interrupt will first fire, and the second is the overall period of the interrupt thereafter. Finally, high-priority interrupts will interrupt low-priority interrupts, so the high-priority interrupt always finishes executing first. Only when all interrupts have been serviced does the program return to normal execution mode. One side effect of this is that if the program schedules interrupts too frequently, the execution will never terminate as the program will spend all the time treating interrupts. Our goal is to prove under what conditions such a program will terminate using matching logic reachability.

Defining the semantics of this language is fairly straightforward and much resembles the way semantics of higher-level languages are represented in the $\mathbb{K}$ framework. We declare a number of cells in the configuration to store the current state of the program, and then provide a number of rules for simply rewriting programs. We begin by processing the entire program into a map from block labels to the contents of their blocks, and then we jump to the block labelled "main". To keep the language simple, each block must end with an instruction describing how to get to the next block (or describing the termination of the program), and we provide one rule for each instruction which modifies the configuration with the result of the operation, and then increments the count of the current time, which in turn will cause I/O to occur or interrupts to fire, based on the scheduling mechanics of the language. As an example of one rule in the language, consider the below which, when combined with the mechanisms in the $\mathbb{K}$ framework for evaluating the subterms of operations, serves to perform lookups of values in registers for instructions:

$\mathsf{rule}\langle \cdots\ \mathsf{rI} : \mathsf{Int} \Rightarrow \mathsf{I2} : \mathsf{Int} \rangle_{\mathsf{k}} \langle \cdots\ \mathsf{I} \mapsto \mathsf{I2}\ \cdots \rangle_{\mathsf{reg}}$

In order to represent this language's concrete state, we use a number of different top-level configuration cells such as the currently executing block $\langle ... \rangle_{\mathsf{k}}$, the code of the entire program $\langle ... \rangle_{\mathsf{pgm}}$, the registers $\langle ... \rangle_{\mathsf{reg}}$, the memory $\langle ... \rangle_{\mathsf{mem}}$, the current values of the external data registers $\langle ... \rangle_{\mathsf{status}}$, the external I/O data events $\langle ... \rangle_{\mathsf{input}}$, the time each instruction takes to execute $\langle ... \rangle_{\mathsf{timing}}$, the time the program has taken to execute so far $\langle ... \rangle_{\mathsf{wcet}}$, the list of currently scheduled timer interrupts $\langle ... \rangle_{\mathsf{timers}}$, the current interrupt priority $\langle ... \rangle_{\mathsf{priority}}$, and the stack of currently executing interrupts $\langle ... \rangle_{\mathsf{interrupts}}$.

Given this programming language, we have written two sample programs which rely on scheduling information

```
main: store #0, #0
  li r0, #100
  jmp loop
loop: rw r1, data, #0
  load r2, #0
  add r2, r2, r1
  store #0, r2
  sub r0, r0, #1
  bne loop, r0, #0
  halt
```

Figure 2: Assembly-like program to poll for external data.

```
main: li r 0 , #100
  li r 1 , #0
  li r 2 , #0
  int t1, #7, #10
  int t2, #10, #15
  jmp loop
```

$$\begin{aligned}
&\text{rule } \langle \$ \Rightarrow \cdot \rangle_{\mathsf{k}} \langle \$ \rangle_{\mathsf{pgm}} \\
&\quad \langle 0 \mapsto (\mathsf{N} \Rightarrow 0), 1 \mapsto (\mathsf{T1} \Rightarrow (\mathsf{T1} + \max(0, \lceil \tfrac{\mathsf{?Duration} - \mathsf{Time1} + \mathsf{Time}}{10} \rceil))), 2 \mapsto (\mathsf{T2} \Rightarrow (\mathsf{T2} + max(0, \lceil \tfrac{\mathsf{?Duration} - \mathsf{Tim2} + \mathsf{Time}}{15} \rceil))) \rangle_{\mathsf{reg}} \\
&\quad \langle \cdots \ \mathsf{add} \mapsto 1, \mathsf{rfi} \mapsto 2 \ \cdots \rangle_{\mathsf{timing}} \\
&\quad \langle \mathsf{Time} \Rightarrow \mathsf{Time} + \mathsf{?Duration} \rangle_{\mathsf{wcet}} \\
&\quad \langle (\mathsf{t1}, (\mathsf{Time1} \Rightarrow \mathsf{?_}), 10)(\mathsf{t2}, (\mathsf{Time2} \Rightarrow \mathsf{?_}), 15) \rangle_{\mathsf{timers}} \\
&\quad \langle 0 \rangle_{\mathsf{priority}} \\
&\quad \langle \cdot \rangle_{\mathsf{stack}} \\
&\quad \langle \cdot \rangle_{\mathsf{interrupts}} \\
&\ \text{if } (\mathsf{N} > 0 \wedge \mathsf{Time1} > \mathsf{Time} \wedge \mathsf{Time2} > \mathsf{Time}) \wedge \\
&\quad (\mathsf{?Duration} = 3 * \mathsf{N} + 1 + max(0, 3 * \lceil \tfrac{\mathsf{?Duration} - \mathsf{Time1} + \mathsf{Time}}{10} \rceil) + max(0, 3 * \lceil \tfrac{\mathsf{?Duration} - \mathsf{Time2} + \mathsf{Time}}{15} \rceil) \wedge \mathsf{?Duration} > 0)
\end{aligned}$$

```
loop: sub r 0 , r 0 , #1
  bne loop , r 0 , #0
  halt
t1: add r 1 , r 1 , #1
  rfi
t2: add r 2 , r 2 , #1
  rfi
```

Figure 3: Assembly-like program with two timer interrupts. The invariant in the grey box proves how long the program takes to execute.

each in their own specific way. The first of these is given in Figure 2. It implements a simple polling loop which scans the external I/O for numbers and sums them up in a location in memory. We have proven this program correct in the sense that under certain conditions the polling loop will sum up numbers to the correct total. However, we do not go into the details here. Instead, we choose to take a close look at the verification of the next example.

2.2 Proving low-level programs correct

The second program we have proven involves timer interrupts, and how long a program containing periodic interrupts takes to execute. To go into more detail of how we prove things, the program as illustrated in Figure 3 contains a representation of the invariant used to prove it.

This program is one which is not trivial to analyze. By the nature of interrupts, once they have been scheduled they may occur at any time. A simplistic model of programming language semantics such as Hoare logic would struggle to successfully capture the semantics of interrupts, because it would need to account for the fact that an interrupt can occur before every single instruction when attempting to determine the effect any one instruction has on the pre- and post-conditions of the Hoare triple. However, an operational semantics handles this without issue, because it stores the current time in the configuration and automatically reroutes execution to the interrupt when specific time intervals have elapsed. Because of this, and because matching logic reachability works by means of partially executing an operational semantics, we can prove properties of the loop in this program correct in such a way as to be aware of the interrupts

which may occur at any time, without needing to perform any additional work. We simply assert in the precondition that interrupts will occur at some undetermined time in the future, and perform case analysis after each increment of the time to decide whether we need to consider the possibility of an interrupt occurring at that moment.

Before we are able to analyze the invariant and explain what it does, it is necessary to introduce some notation used to specify patterns. (1) While all specifications are reachability rules of the form $\varphi \Rightarrow \varphi'$, often φ and φ' share configuration context. In these case we mention the shared context only once and distribute "⇒" across the shared context only in those places where φ and φ' differ. (2) To avoid writing existential quantifiers, logical variables starting with "?" are assumed to be existentially quantified over the entire pattern. (3) To avoid writing portions of the pattern that are not of particular concern to us, we represent anonymous variables using the _ symbol and anonymous existentially quantified variables using the ?_ symbol. (4) In a cell in the configuration which is a collection (such as a map), we use ... to represent an implicit anonymous variable capturing the rest of the contents of the cell not explicitly denoted in the pattern. (5) While both φ and φ' contain logical formulas, each of which is separate from the other, we represent those formulas together as a single formula joined with a conjunction in the side condition of the invariant rule. The convention is that those conjuncts containing only variables present in the left hand side of the rule are part of φ, and those conjuncts containing variables that are only present in the right hand side of the pattern are part of φ'. In particular this is a safe and reasonable notation for representing the postcondition of the rule because since the variable valuation does not change across the left and right hand sides of the pattern, all logical formulas expressed in the precondition are necessarily true in the postcondition. So if desired, the entire side condition can also be viewed as the postcondition of the rule. And those portions of the side condition which are not part of the precondition are simply those for whom being part of the precondition would not make syntactic sense, since they refer to variables not yet bound.

To summarize the invariant as shown, and the way it is used to prove correct a property concerning execution of the loop and the two interrupts, if N iterations of the loop remain to be executed, registers 1 and 2 have values T1 and T2 respectively, an `add` instruction takes 1 cycle to execute and a `rfi` instruction two, two timer interrupts are scheduled as intervals of 10 and 15 cycles and will next execute at times Time1 and Time2 respectively, and we are currently at time Time, in non-interrupt code, and at the beginning of a loop iteration, then after the loop has finished executing, the first interrupt will have occurred an additional X times, the second interrupt an additional Y times, the entire loop will have executed for ?Duration cycles, and X, Y, and ?Duration are determined by solving the following system of equations:

$X = max(-1, \lceil \frac{?Duration - Time1 + Time}{10} \rceil)$

$Y = max(0, \lceil \frac{?Duration - Time2 + Time}{15} \rceil)$

$?Duration = 3 * N + 1 + 3 * X + 3 * Y$

This is consistent with the equations governing how long a task will take to execute when it has interference from higher-priority periodic tasks.

The proof is fully automated. We begin executing the left hand side of the rule symbolically, performing case analysis and circularity when needed. In order to perform circularity, we pass the current constraints to the Z3 SMT solver [2] and automatically determine whether they imply the left hand side of the circularity we want to use. When we reach a decision point in the operational semantics, case analysis breaks the configuration into two parts and executes each separately, removing cases as Z3 tells us that the path we are on is logically infeasible. Then, when we reach the right hand side of the reachability rule, we once again pass the constraints to Z3 and attempt to prove that the implication of the postcondition is valid. If the postcondition is valid for all feasible paths through the execution, we know we have proved the invariant correct. All of this is done automatically, and can be run in only a couple minutes, despite the scheduling of interrupts generating a large number of case analyses in even a very simple program.

We are encouraged by our results in using matching logic to verify programs written in the low-level language presented here. Although this are just preliminary results, we are confident that matching logic reachability and operational semantics based verification can handle low-level languages without any modification to the proof system or language-dependent reasoning.

References

[1] Dominique Clément, Joëlle Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural semantics on the computer. In *Proceedings of the France-Japan AI and CS Symposium*, pages 49–89. ICOT, Japan, 1986.

[2] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008.

[3] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *POPL*, pages 533–544. ACM, 2012.

[4] G.D. Plotkin. A structural approach to operational semantics. *Journal of Logic & Algebraic Programming*, 60-61:17–139, 2004.

[5] Grigore Roşu and Traian-Florin Şerbanuţă. An overview of the K semantic framework. *Journal of Logic & Algebraic Programming*, 79:397–434, 2010.

[6] Grigore Roşu and Andrei Ştefănescu. Checking reachability using matching logic. In *OOPSLA'12*, pages 555–574. ACM, 2012.

[7] Grigore Roşu and Andrei Ştefănescu. From Hoare logic to matching logic reachability. In *FM'12*, volume 7436 of *LNCS*, pages 387–402. Springer, 2012.

[8] Grigore Roşu and Andrei Ştefănescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP'12*, volume 7392 of *LNCS*, pages 351–363. Springer, 2012.

[9] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *LICS'13*. IEEE, 2013. To appear.

[10] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST*, volume 6486 of *LNCS*, pages 142–162, 2010.

[11] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information & Computation*, 115(1):38–94, 1994.