Proving Reachability-Logic Formulas Incrementally

Vlad Rusu[1](✉) and Andrei Arusoaie[1,2]

[1] Inria, Lille, France
Vlad.Rusu@inria.fr, Andrei.Arusoaie@inria.fr
[2] "Al. I. Cuza" University of Iaşi, Iaşi, Romania

Abstract. Reachability Logic (RL) is a formalism for defining the operational semantics of programming languages and for specifying program properties. As a program logic it can be seen as a language-independent alternative to Hoare Logics. Several verification techniques have been proposed for RL, all of which have a circular nature: the RL formula under proof can circularly be used as a hypothesis in the proof of another RL formula, or even in its own proof. This feature is essential for dealing with possibly unbounded repetitive behaviour (e.g., program loops). The downside of such approaches is that the verification of a set of RL formulas is monolithic, i.e., either all formulas in the set are proved valid, or nothing can be inferred about any of the formula's validity or invalidity. In this paper we propose a new, incremental method for proving a large class of RL formulas. The proposed method takes as input a given RL formula under proof (corresponding to a given program fragment), together with a (possibly empty) set of other valid RL formulas (e.g., already proved using our method), which specify sub-programs of the program fragment under verification. It then checks certain conditions are shown to be equivalent to the validity of the RL formula under proof. A newly proved formula can then be incrementally used in the proof of other RL formulas, corresponding to larger program fragments. The process is repeated until the whole program is proved. We illustrate our approach by verifying the nontrivial Knuth-Morris-Pratt string-matching program.

1 Introduction

Reachability Logic (RL) [1–4] is a language-independent logic for defining the operational semantics of programming languages and for specifying properties of programs. For instance, on the `sum` program in Fig. 1, the RL formula

$$\langle \texttt{sum}, \texttt{n}{\mapsto}a \rangle \wedge a \geq 0 \Rightarrow (\exists i, s)\langle \texttt{skip}, \texttt{n}{\mapsto}a\ \texttt{i}{\mapsto}i\ \texttt{s}{\mapsto}s\rangle \wedge s = sum(a) \qquad (1)$$

specifies that after the complete execution of the `sum` program from a configuration where the program variable `n` is bound to a non-negative value a, a configuration where `s` is bound to a value $s = sum(a)$ is reached. Here, $sum(a)$ is a mathematical definition of the sum of natural numbers up to a.

© Springer International Publishing Switzerland 2016
D. Lucanu (Ed.): WRLA 2016, LNCS 9942, pp. 134–151, 2016.
DOI: 10.1007/978-3-319-44802-2_8

```
i := 1;
s := 0;
while (i <= n) do
  s := s + i;
  i := i + 1
end
```

Fig. 1. Program `sum`.

Existing RL verification tools [1,2,4–6] would typically verify formula (1) as follows. First, they would consider (1) together with, e.g., the following formula (2), where `while` denotes the program fragment consisting of the while-loop in Fig. 1. The formula (2) is intended to specify the `while` loop, just like (1) specifies the whole program, and can be seen as encoding a loop invariant.

$$\begin{aligned}&\langle \texttt{while}, \texttt{n}{\mapsto}a\ \texttt{i}{\mapsto}i\ \texttt{s}{\mapsto}s\rangle \wedge 0 < i \leq a+1 \wedge s = \mathit{sum}(i-1) \\ &\Rightarrow (\exists i', s')\langle \texttt{skip}, \texttt{n}{\mapsto}a\ \texttt{i}{\mapsto}i'\ \texttt{s}{\mapsto}s'\rangle \wedge s' = \mathit{sum}(a)\end{aligned} \tag{2}$$

Then, the tool would symbolically execute at least one instruction in the programs in the left-hand side of both (1) and (2) using the semantics of the instructions of the language (assumed to be also expressed as RL formulas[1]), and then execute the remaining programs in the left-hand sides of the resulting formulas *as if both* (1) *and* (2) *became new semantical rules of the language*. For example, when the program executed in (1) reaches the `while` loop, the rule (2) can be applied instead of the rule defining the semantics of the `while` instruction - that is, when proving (1), (2) is assumed to hold. Similarly, when the program in (2) completes one loop iteration, the left-hand side of (2) contains again the same `while` loop as initially, with other values mapped to the variables. Then, (2) is applied instead of the rule defining the semantics of the `while` instruction. Thus, it is assumed that (2) holds after having completed one loop iteration.

The *circular* reasoning illustrated in the above example is sound, in the sense that if such a proof succeeds, all the formulas under proof are (semantically) valid. However, if the proof does not succeed, nothing can be said about the validity of the formulas. In our example, (1) or (2) (or both) could be invalid.

Contribution. In this paper we propose a new method for proving a significant subset of RL formulas, which, unlike existing verification methods, is *incremental*. In our example, the proposed method would first prove (2), and then would prove (1) knowing for a fact (i.e., not assuming) that (2) is valid. Thus, if the proof of (1) fails for some reason, the user still knows that (2) holds, and can take action for fixing the proof based on this knowledge. Of course, for a simple program such as the above example the advantage of incremental RL verification is not obvious, but it turns out to make quite a difference when verifying more challenging programs, such as the KMP program illustrated later in the paper.

[1] For the language of interest in this paper the rules are shown in Sect. 2.

We first establish an equivalence between the validity of RL formulas and two technical conditions (one condition is an invariance property, and the other one regards the so-called capturing of terminal configurations). Then we propose a graph-construction approach that takes a given RL formula under proof (corresponding to a given program fragment), together with a (possibly empty) set of other valid RL formulas (e.g., proved using a previous iteration of our approach, or by any other sound RL formula verification method). The latter formulas specify sub-programs of the program fragment currently under verification. The invariance and terminal-configuration capturing conditions are then checked on the graph, thus establishing the validity of the RL formula under proof. The newly proved formula can then be incrementally used in the proof of other RL formulas, corresponding to larger program fragments. The same process is then repeated until, eventually, the whole program is proved.

Of course, the proposed method has limitations, since verification of RL formulas is in general undecidable. The graph construction may not terminate, or the conditions to be checked on it may not hold. One situation that a purely incremental method cannot handle is mutually recursive function calls, in which none of the functions can be verified individually unless (coinductively) assuming that the other function's specifications hold. A natural solution here is to use an incremental method as much as possible, and to locally apply a circular approach only for subsets of formulas that the incremental method cannot handle.

In order to demonstrate the feasibility of our approach we illustrate it on the nontrivial Knuth-Morris-Pratt KMP string-matching program. The program is written in a simple imperative language, whose syntax and semantics is defined in Maude [7]. We chose Maude in order to benefit from its reflective capabilities, which turned out to be very useful for implementation purposes. We are using a specific version of Maude that has been interfaced with the Z3 solver [8], which is here used for simplifying conditions required for proving RL formula validity.

Paper Organisation. After this introduction we present in Sect. 2 the Maude-based definition of a simple imperative programming language IMP+ that includes assignments, conditions, loops, and simple procedures operating on global variables. In Sect. 3 we present background notions: Reachability Logic, and how the language definition from the previous section fits in this framework (Sect. 3.1); and language-parametric symbolic execution, together with its implementation by rewriting based on transforming the semantical rules of a language (Sect. 3.2). In Sect. 4 we present the incremental RL-formula verification method. In Sect. 5 we illustrated our method on the KMP string-matching algorithm, and in Sect. 6 we conclude and present related and future work. An extended version containing detailed proofs of technical results is available at https://hal.inria.fr/hal-01282379.

2 Defining a Simple Programming Language

In this section we define the language IMP+ in Maude. IMP+ is simple enough so that its Maude code is reasonably small (less than two hundred lines of code), yet expressive enough for programming algorithms on arrays such as the KMP. We assume Maude is familiar to readers; for details the standard reference is [7].

Datatypes. IMP+ computes over Booleans, integers, and integer arrays. We use the builtin Booleans and integers of Maude, and provide a standard algebraic definition of arrays. The constructor `array : Nat -> IntArray` creates an array of a given length. The operation `store : IntArray Nat Int -> IntArray` stores a given integer (third argument) at a given natural-number index. An operation `select : IntArray Nat -> Int` returns the element at the position given by the second argument. These functions are defined equationally. They return error values in case of attempts to access indices out of an array's bounds.

Syntax. The syntax on IMP+ consists of expressions (arithmetic and Boolean) and statements. Each of these syntactical categories is defined by a sort, i.e., `AExp`, `BExp`, and `Stmt`. Allowed arithmetical operations are addition, substraction, and array selector, denoted by `_++_`, `_--_`, and `_[_]` respectively, in order to avoid confusion with the corresponding Maude operations on the datatypes. In the same spirit, Boolean operations are less-or-equal-than (`_<==_`) and equality (`_===_`); negation `!`; and conjunction `_&&_`. Such expressions are built from identifiers (i.e., program variables) and constants (Maude integers and Booleans).

The statements of IMP+ are: assignments to integer variables and array elements (`_:=_`); conditional (`if_then_else_endif`); while loops (`while_do_end`); parameterless function declaration (`function_(){_}`) and call (`_()`); a `print` instruction; and finally, a sequencing `_;_` instruction that, for convenience, is declared associative with the "do-nothing" `skip` instruction as a neutral element.

Semantics. Semantical rules operate on *configurations*, which consist of a program to be executed, a mapping of integer variables to values and of function names to statements, and a list of integers denoting the output of the program. In Maude we write a constructor `<_,_,_,_> : Stmt Map Funs Ints -> Cfg`. Getters and setters for the `Map` and `Funs` maps are also equationally defined.

The semantics of IMP+ then consists in evaluating expressions (in a given map, assigning values to variables) and statements (in a given configuration, describing all the infrastructure required for statements to execute). Expressions are evaluated using equations, and statements are evaluated using rewrite rules.

Evaluating Expressions. This amounts to writing a function `eval` and equations:

```
op eval : AExp Map  -> Int .
eq eval(I, M) = I .
eq eval(X, (M (X -> J))) = J .
eq eval(X[E], (M (X -> A))) = select(A,eval(E,(M (X -> A)))) .
...
op eval : BExp Map -> Bool .
eq eval(B,M) = B .
eq eval(Cnd1 && Cnd2, M) = eval(Cnd1,M) and eval(Cnd2,M) .
...
```

That is, `eval` goes through the structure of an expression and evaluates it in a given mapping of values to variables. Here, e.g., `M (X -> J)` denotes an associative-commutative map, constructed as the anonymous juxtaposition operation `__` of a map variable `M` with a map of the identifier `X` to the integer `J`.

Evaluating Statements. This is performed by rewrite rules, some of which are:

```
rl [assign]: <((X := E) ; S), M, F, O > => < S, set(X, eval(E, M), M), F, O > .

crl [if-true]: <(if Cnd then S1 else S2 endif) ; S, M, F, O > => < S1 ; S, M, F, O >
               if  eval(Cnd,M) .

crl [if-false]: <(if Cnd then S1 else S2 endif) ; S, M, F, O > => < S2 ; S, M, F, O >
                if  not eval(Cnd,M) .

rl [while]: <(while Cnd do S1 end) ; S, M, F, O > =>
             <(if Cnd then S1 ; while Cnd do S1 end else skip endif) ; S, M, F, O > .

rl [print]: < (print E) ; S, M, F, O >   =>  < S, M, F, (O ; eval(E,M)) > .
```

The first rule deals with assigment to a program-variable `X` of an arithmetic expression `E`. It uses the `set` function on maps in order to update the map so that `X` is mapped to the value of `E`. Another rule, not shown here, deals with assignments to array elements. The following two rules describe the two possible outcomes of a conditional instruction, depending on the value of the condition. The rule for the while loop consists essentially in loop unrolling. The rule for the printing instruction appends the value of the instruction's argument to the list of integers (last argument of configurations) denoting the program's output.

3 Reachability Logic and Symbolic Execution

In this section we present background material used in the rest of the paper. We illustrate the concepts with examples from the IMP+ language.

3.1 Reachability Logic

Several versions of RL have been proposed in the last few years [1–4]. Moreover, RL is built on top of *Matching Logic* (ML), which also exists in several versions [9–11]. (The situation is somewhat similar to the relationship between rewriting logic and the equational logics underneath it.) We adopt the recent *all-paths* interpretation of RL [4], built upon a minimal ML that is enough to express typical practically-relevant properties about program configurations and is amenable to *symbolic execution* by rewriting, a key ingredient of our method.

The formulas of ML that we consider are called *patterns* and are defined as follows. Assume an algebraic signature Σ with a set S of sorts, including two distinguished sorts $Bool, Cfg \in S$. We write $T_{\Sigma,s}(Var)$ for the set of terms of sort s over a set Var of S-indexed variables and $T_{\Sigma,s}$ for the set of ground terms of sort s. We identify the $Bool$-sorted operations in Σ with a set Π of predicates.

Example 1. Consider the Maude definition of the IMP+ language. Then, Σ is the algebraic signature containing all the sorts and operations described in the previous section, including the `Bool` and `Cfg` sorts. The operation `eval : BExp Map -> Bool` has sort `Bool` and is thus identified with a predicate in the set Π. The sort `Cfg` has the constructor `<_,_,_,_> : Stmt Map Funs Ints -> Cfg`.

Definition 1 (Pattern). *A pattern is an expression of the form $(\exists X)\pi \wedge \phi$, where $X \subset Var$, $\pi \in T_{\Sigma,Cfg}(X)$ and ϕ is a FOL formula over the FOL signature (Σ, Π) with free variables in X.*

We often denote patterns by φ and write $\varphi \triangleq (\exists X)\pi \wedge \phi$ to emphasise its components: the quantified variables X, the *basic pattern* π, and ϕ, the *condition*. We let $FreeVars(\varphi)$ denote the set of variables freely occurring in a pattern φ, defined as usual (i.e., not under the incidence of a quantifier). We often identify basic patterns π with $(\exists\emptyset)\pi \wedge true$, and *elementary patterns* $\pi \wedge \phi$ with $(\exists\emptyset)\pi \wedge \phi$.

Example 2. The left and right-hand sides of the rules defining the semantics of IMP+ are basic patterns, `< S, M, F, O > /\ eval(true,M)` is an elementary pattern, and $(\exists$ `O`$)$ `< S, M, F, O > /\ eval(true,M)` is a pattern.

We now describe the semantics of patterns. We assume a model M of the algebraic signature Σ. In the case of the Maude specification of IMP+ the model M, M is the initial model induced by the specification's equations and axioms. For sorts $s \in S$ we write M_s for the interpretation (a.k.a. carrier set) of the sort s.

We call *valuations* the functions $\rho : Var \rightarrow M$ that assign to variables in Var a value in M of a corresponding sort, and *configurations* the elements in M_{Cfg}.

Definition 2 (Pattern Semantics). *Given a pattern $\varphi \triangleq (\exists X)\pi \wedge \phi$, $\gamma \in M_{Cfg}$ a configuration, and $\rho : Var \rightarrow M$ a valuation, the satisfaction relation $(\gamma, \rho) \models \varphi$ holds iff there exists a valuation ρ' with $\rho'|_{Var\setminus X} = \rho|_{Var\setminus X}$ such that $\gamma = \rho'(\pi)$ and $\rho' \models \phi$ (where the latter $\models$ denotes satisfaction in FOL, and $\rho_{|Var\setminus X}$ denotes the restriction of the valuation ρ to the set $Var \setminus X$).*

We let $[\![\varphi]\!]$ denote the set $\{\gamma \in M_{Cfg} \mid (\exists \rho : Var \rightarrow M)(\gamma, \rho) \models \varphi\}$. A formula φ is *valid in M*, denoted by $M \models \varphi$, if it is satisfied by all pairs (γ, ρ).

We now recall Reachability-Logic (RL) formulas, the transition systems that they induce, and their all-paths semantics [4] that we will be using in this paper.

Definition 3 (RL Formulas). *An* RL *formula is a pair of patterns $\varphi \Rightarrow \varphi'$.*

Examples of RL formulas were given in the introduction. The rules defining the semantics of IMP+ are also RL formulas (for the conditional rules, just assume that the expression following `if` is the condition of the rule's left-hand side).

Let $\mathcal{S}$ denote a fixed set of RL formulas, e.g., the semantics of a given language. We define the transition system defined by $\mathcal{S}$ together with some notions related to this transition system, and then the notion of validity for RL formulas.

Definition 4 (Transition System Defined by $\mathcal{S}$). *The* transition system *defined by $\mathcal{S}$ is $(M_{Cfg}, \Rightarrow_{\mathcal{S}})$, where $\Rightarrow_{\mathcal{S}} = \{(\gamma, \gamma') \mid (\exists \varphi \Rightarrow \varphi' \in \mathcal{S})(\exists \rho)(\gamma, \rho) \models \varphi \wedge (\gamma', \rho) \models \varphi'\}$. We write $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ for $(\gamma, \gamma') \in \Rightarrow_{\mathcal{S}}$. A state γ is* terminal *if there is no γ' such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$. A* path *is a sequence $\gamma_0 \cdots \gamma_n$ such that $\gamma_i \Rightarrow_{\mathcal{S}} \gamma_{i+1}$ for all $0 \le i \le n-1$. Such a path is* complete *if γ_n is terminal.*

An RL *formula $\varphi \Rightarrow \varphi'$ is valid, written $\mathcal{S} \models \varphi \Rightarrow \varphi'$, if for all pairs (γ_0, ρ) such that $(\gamma_0, \rho) \models \varphi$, and all complete paths $\gamma_0 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n$, there exists $0 \le i \le n$ such that $(\gamma_i, \rho) \models \varphi'$.*

Note that the validity of RL formulas is only determined by finite, complete paths. Infinite paths, induced by nonterminating programs, are not considered. Thus, termination is assumed: as a program logic, RL is a logic of partial correctness. We restrict our attention to RL formulas satisfying the following assumption:

Assumption 1. *RL formulas have the form $\pi_l \wedge \phi_l \Rightarrow (\exists Y)\pi_r \wedge \phi_r$ and satisfy $\mathit{FreeVars}(\pi_r) \subseteq \mathit{FreeVars}(\pi_l) \cup Y$, $\mathit{FreeVars}(\phi_r) \subseteq \mathit{FreeVars}(\pi_l) \cup \mathit{FreeVars}(\pi_r)$, and $\mathit{FreeVars}(\phi_l) \subseteq \mathit{FreeVars}(\pi_l)$.*

That is, the left-hand side is an elementary pattern, and the right hand side is a pattern, possibly with quantifiers. Such formulas are typically expressive enough for expressing language semantics (for this purpose, quantifiers are not even required)[2] and program properties. For program properties, existentially quantified variables in the right-hand side are useful to denote values computed by a given program, which are not known before the program computes them, such as s - the sum of natural numbers up to a given bound - in the formula (1).

3.2 Language-Parametric Symbolic Execution

We now briefly present symbolic execution, a well-known program analysis technique that consists in executing programs with symbolic input (e.g. a symbolic value x) instead of concrete input (e.g. 0). We reformulate the language-independent symbolic execution approach we already presented elsewhere [6], with some simplifications (e.g., unlike [6] we do not use coinduction). The approach consists in transforming the signature Σ and semantics $\mathcal{S}$ of a programming language so that, under reasonable restrictions, executing a program with the modified semantics amounts to executing the program symbolically.

Consider the signature Σ corresponding to a language definition. Let *Fol* be a new sort whose terms are all FOL formulas, including existential and universal quantifiers. Let *Id* and *IdSet* be new sorts denoting identifiers and sets of identifiers, with a union operation $\_,\_$. Let $\mathit{Cfg}^{\mathfrak{s}}$ be a new sort, with constructor $(\exists\_)\_\wedge\_ : \mathit{IdSet} \times \mathit{Cfg} \times \mathit{Fol} \to \mathit{Cfg}^{\mathfrak{s}}$. Thus, patterns $(\exists X)\pi \wedge \phi$ correspond to terms $(\exists X)\pi \wedge \phi$ of sort $\mathit{Cfg}^{\mathfrak{s}}$ in the enriched signature and reciprocally. Consider also the following set of RL formulas, called the *symbolic version of $\mathcal{S}$*:

$$\mathcal{S}^{\mathfrak{s}} \triangleq \{(\exists \mathcal{X})\pi_l \wedge \psi \Rightarrow (\exists \mathcal{X}, Y)\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r) | \pi_l \wedge \phi_l \Rightarrow (\exists Y)\pi_r \wedge \phi_r \in \mathcal{S}\}$$

with ψ a new variable of sort *Fol*, and $\mathcal{X}$ a new variable of sort *IdSet*.

[2] See, e.g., the languages defined in the $\mathbb{K}$ framework: http://k-framework.org.

Example 3. The following conditional rule is part of the semantics $\mathcal{S}$ of IMP+: `< if C then S1 else S2 endif ; S, M, F, O > => < S1 ; S, M, F, O > if eval(C,M)` Written as an RL formula (with patterns in left and right-hand sides) it becomes[3] `<if C then S1 else S2 endif ; S, M, F, O >` $\wedge$ `eval(C,M)=> < S1 ; S, M, F, O >` The corresponding rule in $\mathcal{S}^{\mathfrak{s}}$ becomes an unconditional rule: $(\exists\mathcal{X})$ `<if C then S1 else S2 endif ; S, M, F, O >` $\wedge\psi$ `=>` $(\exists\mathcal{X})$ `<S1 ; S, M, F, O >` $\wedge$ $(\psi \wedge$ `eval(C,M)`$)$.

The interest of the above nontrivial construction is that, under reasonable assumptions, stated below, rewriting with the rules in $\mathcal{S}^{\mathfrak{s}}$ achieves a *simulation* of rewriting with the rules in $\mathcal{S}$, which is a result that we need for our approach.

Assumption 2. *There exists a* builtin subsignature $\Sigma^b \subsetneq \Sigma$. *The sorts and operations in* Σ^b *are* builtin*, while all others are* non-builtin. *The sort Cfg is not builtin. Non-builtin operation symbols may only be subject to a (possibly empty) set of* linear, regular, and non-collapsing *axioms.*

We recall that an axiom $u = v$ is linear if both u, v are linear (a term is linear if any variable occurs in it at most once); it is regular if both u, v have the same set of variables; and it is non-collapsing if both u, v have non-builtin sorts.

Example 4. For the IMP+ language specification we assume that the non-builtin sorts are `Cfg`, `Stmt` (for statements), and `Funs` (which map function identifiers to statements). Statements were declared to be associative with unity, whereas maps of identifiers to statements were taken to be associative and commutative with unity. All these axioms have the properties requested by Assumption 2.

In order to formulate the simulation result we now define the transition relation generated by the set of symbolic RL rules $\mathcal{S}^{\mathfrak{s}}$. It is essentially rewriting modulo the congruence $\cong$ on $T_{\Sigma}(\mathit{Var})$ induced by the axioms in Assumption 2. Let $\mathit{Var}^b \subset \mathit{Var}$ be the set of variables of builtin sorts. We first need the following technical assumption, which does not restrict the generality of our approach:

Assumption 3. *For every* $\pi_l \wedge \phi_l \Rightarrow (\exists Y)\pi_r \wedge \phi_r \in \mathcal{S}$, $\pi_l \in T_{\Sigma\setminus\Sigma^b}(\mathit{Var})$, π_l *is linear, and* $Y \subseteq \mathit{Var}^b$.

The assumption can always be made to hold by replacing in π_l all non-variable terms in Σ^b and all duplicated variables by fresh variables, and by equating in the condition ϕ_l the new variables to the terms that they replaced.

For the sake of complying with the definition of rewriting we need to extend the congruence $\cong$ to terms of sort $\mathit{Cfg}^{\mathfrak{s}}$ by $(\exists X)\pi_1 \wedge \phi \cong (\exists X)\pi_2 \wedge \phi$ iff $\pi_1 \cong \pi_2$.

Definition 5 (Relation $\Rightarrow_{\alpha^{\mathfrak{s}}}$). *For* $\alpha^{\mathfrak{s}} \triangleq (\exists\mathcal{X})\pi_l \wedge \psi \Rightarrow (\exists\mathcal{X}, Y)\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r) \in \mathcal{S}^{\mathfrak{s}}$ *we write* $(\exists X)\pi \wedge \phi \Rightarrow_{\alpha^{\mathfrak{s}}} (\exists X, Y)\pi' \wedge \phi'$ whenever $(\exists X)\pi \wedge \phi$ $\alpha^{\mathfrak{s}}$ is rewritten by $\alpha^{\mathfrak{s}}$ to $(\exists X, Y)\pi' \wedge \phi'$, *i.e., there exists a substitution* σ' *on* $\mathit{Var} \cup \{\mathcal{X}, \psi\}$ *such that* $\sigma'((\exists\mathcal{X})\pi_l \wedge \psi) \cong (\exists X)\pi \wedge \phi$ *and* $\sigma'((\exists\mathcal{X}, Y)\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r) = (\exists X, Y)\pi' \wedge \phi'$.

[3] We liberally use a mixture of Maude and math notation for the sake of the example.

Lemma 1 ($\Rightarrow_{\alpha^s}$ Simulates $\Rightarrow_{\alpha}$). *For all $\gamma, \gamma' \in M_{Cfg}$, all patterns φ with $FreeVars(\varphi) \subseteq Var^b$, and all valuations ρ, if $(\gamma, \rho) \models \varphi$ and $\gamma \Rightarrow_{\alpha} \gamma'$ then there exists φ' with $FreeVars(\varphi') \subseteq Var^b$ such that $\varphi \Rightarrow_{\alpha^s} \varphi'$ and $(\gamma', \rho) \models \varphi'$.*

As a consequence, any concrete execution (following $\Rightarrow_{\mathcal{S}}$) such that the initial configuration satisfies a given initial pattern φ is simulated by a symbolic execution (following $\Rightarrow_{\mathcal{S}^s}$) starting in φ. We shall also use the following notion of *derivative*, which collects all the symbolic successors of a pattern by a rule:

Definition 6 (Derivatives). *$\Delta_{\alpha}(\varphi) = \{\varphi' \mid \varphi \Rightarrow_{\alpha^s} \varphi'\}$ for any $\alpha \in \mathcal{S}$.*

Since the symbolic successors are computed by rewriting, the derivative operation is computable and always returns a finite set of patterns.

4 Proving RL Formulas Incrementally

In this section we present an incremental method for proving RL formulas. We first state two technical conditions and prove that they are equivalent to RL formula validity. The equivalence works for so-called *terminal* formulas, whose right-hand side specifies a completed program; however, a generalisation to non-terminal formulas, required for incremental verification, is also given. Thus, RL formula verification amounts to checking the two above-mentioned conditions.

For this, we present a graph construction based on symbolic execution that, if it terminates successfully, ensures that the two conditions in question hold for a given RL formula. The graph construction is parameterised by a set of formulas that have already been proved valid (using the same method, or any other sound one). These formulas correspond to subprograms of the given program fragment that the current formula under proof specifies. The current formula, once proved, can then be used in proofs of formulas specifying larger program fragments.

We consider a fixed set $\mathcal{S}$ or RL formulas and their transition relation $\Rightarrow_{\mathcal{S}}$. The first of the two following definitions says that all terminal configurations reachable from a given pattern "end up" as instances of a quantified basic pattern:

Definition 7 (Capturing All Terminal Configurations). *We say that a pattern $(\exists Y)\pi'$* captures all terminal configurations *for a pattern φ if for all (γ, ρ) such that $(\gamma, \rho) \models \varphi$, and all complete paths $\gamma \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma'$, $(\gamma', \rho) \models (\exists Y)\pi'$.*

The second definition characterises FOL formulas that hold in a given quantified pattern, i.e., conditions satisfied by all configurations reachable from a given initial pattern whenever they "reach" the quantified pattern in question:

Definition 8 (Invariant at, Starting from). *We say that a FOL formula $(\exists Y)\phi'$ is* invariant at *a pattern $(\exists Y)\pi'$* starting from *a pattern φ if for all (γ, ρ) such that $(\gamma, \rho) \models \varphi$, all paths $\gamma \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma'$, and all valuations ρ' with $\rho'|_{Var \setminus Y} = \rho|_{Var \setminus Y}$, if $\gamma' = \rho'(\pi')$, then $\rho' \models \phi'$.*

Note that the *same* values of the variables Y were used for satisfying π' and ϕ'.

Definition 9. *A basic pattern π' is terminal if for all valuations ρ, $\rho(\pi')$ is a terminal configuration. A rule $\pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$ is terminal if π' is terminal.*

The following proposition characterises the validity of terminal RL formulas:

Proposition 1 (Equivalent Conditions for Terminal Formula Validity). *Consider a terminal formula $\pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$. Then $\mathcal{S} \models \pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$ iff*

1. *$(\exists Y)\phi'$ is invariant at $(\exists Y)\pi'$ starting from $\pi \wedge \phi$, and*
2. *$(\exists Y)\pi'$ captures all terminal configurations for $\pi \wedge \phi$.*

Remark 1. The ($\Leftarrow$) implication in Proposition 1 is the important one for the soundness of our method. Its proof naturally follows from definitions. For the reverse implication, the following assumption is required: for all right-hand sides $\varphi_r \triangleq (\exists Y)\pi_r \wedge \phi_r$ of rules in $\mathcal{S}$, if $\rho(\pi_r) = \rho'(\pi_r)$ then $\rho|_{\mathit{FreeVars}(\pi_r)} = \rho'|_{\mathit{FreeVars}(\pi_r)}$. The assumption does not restrict generality as it can always be made to hold, by replacing subterms of patterns by fresh variables (and adding equations to the condition) and by noting that the *Cfg* sort is interpreted syntactically in the model M. Then, $\pi_r \triangleq f(x_1, \ldots, x_n)$ where f is the constructor for the *Cfg* sort, and $\rho(f(x_1, \ldots, x_n)) = \rho'(f(x_1, \ldots, x_n))$ iff $\rho(x) = \rho'(x_i)$ for all variables x_i.

Remark 2. Proposition 1 works for terminal RL formulas. We shall need the following observation: assume that an RL formula of the following form $\langle P \ldots \rangle \wedge \phi \Rightarrow (\exists Y)\langle \mathit{skip} \ldots \rangle \wedge \phi'$ has been proved valid, where P is a program, *skip* denotes the empty program, and suspension dots denote the rest of the configurations (which depend on the programming language). Then, assuming a sequencing operation[4] denoted by semicolon, the following formula $\langle P; Q \ldots \rangle \wedge \phi \Rightarrow (\exists Y)\langle Q \ldots \rangle \wedge \phi'$ is also valid: if each terminal path executing P ended up in the empty program, then each path executing $P; Q$ still has Q to execute after having executed P. As shown later in this section, the validity of such "generalized" formulas enables us to incrementally use a proved-valid formula in the proofs of other formulas.

Proposition 1 is the basis for proving RL formulas, by checking the conditions (1) and (2). We now show how the conditions can be checked mechanically.

Symbolic Graph Construction. The graph-construction procedure in Fig. 2 uses symbolic execution and is used to check the conditions (1) and (2) in Proposition 1. Before we describe the procedure we introduce the components that it uses.

A Partial Order $<$ on $\mathcal{S}$. The procedure assumes a set of RL formulas $\mathcal{S}$, which consist of the semantical rules $\mathcal{S}_0$ of a programming language and a (possibly empty) set of RL formulas $\mathcal{G}$ that were already proved valid in an earlier step of our envisaged incremental verification method. Such formulas, sometimes called *circularities* in RL verification, specify subprograms of the program under verification, and are assumed here to have the form $\langle P; Q, \ldots \rangle \wedge \phi \Rightarrow (\exists Y)\langle Q, \ldots \rangle \wedge \phi'$

[4] "Sequencing" and "empty" do not need to be actual statements of the programming language; they can just be artifacts required by the language's operational semantics.

```
0: $G = (N \triangleq \{\pi \wedge \phi\}, E \triangleq \emptyset)$, $Failure \leftarrow false$, $New \leftarrow N$
1: while not $Failure$ and $New \neq \emptyset$
  2: choose $\varphi \triangleq (\exists X_n)\pi_n \wedge \phi_n \in New$; $New \leftarrow New \setminus \{\varphi\}$
  3: if $match_{\cong}(\pi_n, \pi') = \emptyset$ then
      4: if $\bigvee_{\alpha \in min(<)} inclusion(\varphi, lhs(\alpha)) = true$ then
          5: forall $\alpha \in min(<)$, forall $\varphi' \in \Delta_\alpha(\varphi)$
              6: if $inclusion(\varphi', \varphi)$ then $E \leftarrow E \cup \{\varphi \xrightarrow{\alpha} (\pi \wedge \phi\})$
              7: else $New \leftarrow New \cup \{\varphi'\}$; $E \leftarrow E \cup \{\varphi \xrightarrow{\alpha} \varphi'\}$ endif
          8: $N \leftarrow N \cup New$
      9: else $Failure \leftarrow true$ endif
  10: elseif not $inclusion(\varphi, (\exists Y)\pi' \wedge \phi')$ then $Failure \leftarrow true$ endif.
```

Fig. 2. Graph construction. $match_{\cong}()$ is matching modulo the non-bultin axioms (cf. Sect. 3.2), and *inclusion*() is the object of Definition 10.

(cf. Remark 2). During symbolic execution, circularities can be symbolically applied "in competition with" rules in the semantics (e.g., when the program to be executed is $P; Q$, the symbolic version of the above rule can be applied, but the symbolic version of the semantical rule for the first instruction of P can be applied as well). We solve the conflict between semantical rules and circularities by giving priority to the latter.

We use the following notations. Let $lhs(\alpha)$ denote the left-hand side of a formula α. Let $\mathcal{G} < \mathcal{S}_0$ denote the fact that for every $g \in \mathcal{G}$ and $\alpha \in \mathcal{S}_0$, $g < \alpha$. Let $\mathcal{S}_0 \models \mathcal{G}$ denote $\mathcal{S}_0 \models g$, for all $g \in \mathcal{G}$, and $min(<)$ denote the minimal elements of $<$.

Assumption 4. *We assume a partial order relation $<$ on $\mathcal{S} \triangleq \mathcal{S}_0 \cup \mathcal{G}$ satisfying: $\mathcal{G} < \mathcal{S}_0$, $\mathcal{S}_0 \models \mathcal{G}$, and for all $\alpha' \in \mathcal{S}$ and pairs (γ, ρ), if $(\gamma, \rho) \models lhs(\alpha')$ then there exists a rule $\alpha \in min(<)$ such that $(\gamma, \rho) \models lhs(\alpha)$.*

This assumption is satisfied by taking as minimal elements of $<$ previously proved circularities, which gives them priority over rules in the semantics that can be applied in competition with them. The other rules in the semantics, which are not in competition with circularities, are not related by $<$ with other formulas and are thus minimal by definition (and valid, since $\alpha \in \mathcal{S}$ implies $\mathcal{S} \models \alpha$).

Inclusion Between Patterns. The graph-construction procedure uses a test of inclusion between patterns, which satisfies the following definition.

Definition 10 (Inclusion). *An inclusion test is a function that, given patterns φ, φ', returns true if for all pairs (γ, ρ), if $(\gamma, \rho) \models \varphi$ then $(\gamma, \rho) \models \varphi'$.*

The Graph Construction. We are now ready to present the procedure in Fig. 2. The procedure takes as input an RL formula $\pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$ and a set $\mathcal{S}$ of RL formula with an order $<$ on $\mathcal{S}$ as discussed earlier in this section. It builds a graph (N, E) with N the set of nodes (initially, $\{\pi \wedge \phi\}$) and E the set of edges (initially empty). It uses two variables to control a while loop: a Boolean variable *Failure* (initially *false*) and a set of nodes *New* (initially equal to $N = \{\pi \wedge \phi\}$).

At each iteration of the while loop, a node $\varphi_n \triangleq (\exists X_n)\pi_n \wedge \phi_n$ is taken out from *New* (line 2) and checks whether there is a matcher modulo $\cong$ (cf. Sect. 3.2) of π' onto π_n (line 3). If this is the case, then π_n is an instance of the (terminal) basic pattern π', and the procedure goes to line 10 to check whether φ_n "as a whole" is included in $(\exists Y)\pi' \wedge \phi'$. If this is not the case, then, informally, this indicates a terminal path that does not satisfy the right-hand side of the formula under proof, i.e., of the fact that $(\exists Y)\phi'$ is not invariant at $(\exists Y)\pi'$, in contradiction with the first hypothesis of Proposition 1 that the procedure is checking; *Failure* is reported, which terminates the execution of the procedure. However, if the test at line 3 indicated that π_n is not an instance of the terminal pattern π', then another inclusion test is performed (line 4): whether there exists a minimal rule in $\mathcal{S}$ (i.e., a rule in the language's semantics, or a circularity already proved, as discussed earlier in this section) whose left-hand side includes φ_n. If this is not the case then, informally, this indicates a terminal configuration that is not an instance of $(\exists Y)\pi'$, which contradicts the second hypothesis of Proposition 1, making the procedure terminate again with *Failure* = *true*.

If, however, the inclusion test at line 4 succeeds then all symbolic successors φ'_n of φ_n by minimal rules α w.r.t. $<$ are computed. Each of these patterns is tested for inclusion in the initial node $\pi \wedge \phi$. If inclusion holds then an edge is added from φ_n to the initial node, labelled by the rule that generated the symbolic successor in question. Otherwise, a new node φ'_n is created, and an edge from the current node φ_n to the new node, labelled by the rule that generated it, is created, and the while loop proceeds to the next iteration.

The graph-construction procedure does not terminate in general, since the verification of RL formulas is undecidable. However, if it does terminate with *Failure* = *false* then the two conditions equivalent to the validity of the procedure's input $\pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$ hold, i.e., $\mathcal{S} \models \pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$, which is the desired conclusion. This is established by the results in the rest of this section.

The paths in the constructed graph simulate concrete execution paths whose transitions are given by rules from $\mathcal{S}_0$. This is formalised and used in the proof of the main theorem states that the hypotheses of Proposition 1, equivalent to RL formula validity, are checked by the graph-construction procedure.

Theorem 1. *If the procedure in Fig. 2 terminates with Failure = false on a terminal* RL *formula* $\pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$, *then* $(\exists Y)\phi'$ *is invariant at* $(\exists Y)\pi'$ *starting from* $\pi \wedge \phi$, *and* $(\exists Y)\pi'$ *captures all terminal configurations starting from* $\pi \wedge \phi$.

Theorem 1 uses the following (and last) assumptions on RL formulas:

Assumption 5. *All rules* $\varphi_l \Rightarrow \varphi_r \in \mathcal{S}$ *have the following properties:*

1. *for all pairs* (γ, ρ) *such that* $(\gamma, \rho) \models \varphi_l$ *there exists* γ' *such that* $(\gamma', \rho) \models \varphi_r$[5].
2. $[\![\varphi_l]\!] \cap [\![\varphi_r]\!] = \emptyset$.

The first of the above assumptions says that if the left-hand side of a rule matches a configuration then there is nothing in the right-hand side preventing the application. This property is called *weak well-definedness* in [4] and is shown there

[5] This property is called weak well-definedness in [4].

to be a necessary condition for obtaining a sound proof system for RL. The second condition just says that the left and right-hand sides of rules cannot share instances - such rules could generate self-loops on instances, which are useless. We then obtain as a corollary the soundness of our RL formula proof method:

Corollary 1 (Soundness). *If procedure in Fig. 2 terminates with Failure = false on a terminal* RL *formula* $\pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$ *then* $\mathcal{S} \models \pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$.

Incremental Verification. We are now ready to describe our incremental RL formula verification method. The method works in a setting where each formula has an associated *code* that it specifies, and that for a given RL formula f, $code(f)$ returns the given code. Considering the RL formulas (1) and (2) in Sect. 1, $code(1)$ is the `sum` program in Fig. 1 and $code(2)$ is the `while` subprogram.

The problem to be solved is: given two sets of formulas: $\mathcal{S}$ (the semantics of a language) and $\mathcal{G}$ (the specification of a given program and of some of its subprograms) prove *for all* $g \in \mathcal{G}$, $\mathcal{S} \models g$ (for short, $\mathcal{S} \models \mathcal{G}$).

We use partial orders $<$ on $\mathcal{S}$ (initially empty) and $\sqsubset$ on $\mathcal{G}$, defined by $g_1 \sqsubset g_2$ whenever $code(g_1)$ is a strict subprogram of $code(g_2)$. Without restriction of generality we take the formulas in $\mathcal{G}$ to be terminal (which is natural: a piece of code is specified by stating what the code "does" when it terminates). The verification consists repeatedly applying the following steps while $\mathcal{G} \neq \emptyset$:

- choose $g \in \mathcal{G}$ minimal w.r.t. $\sqsubset$ and prove it, based on Corollary 1;
- remove g from $\mathcal{G}$, transform g into a non-terminal formula (cf. Remark 2) and add the resulting formula g' to $\mathcal{S}$;
- extend $<$ on the newly obtained set $\mathcal{S}$ so that g' is smaller than any formula in $\mathcal{S}$ that can be applied concurrently with g'.

Example 5. Consider the `sum` program in Fig. 1. $\mathcal{S}$ consists of the semantical rules of IMP+, and $\mathcal{G}$ consists of formulas (1) and (2) in Sect. 1, with (2) $\sqsubset$ (1).

At the first iteration (2) is chosen. It is verified based on Corollary 1 (which builds the graph according to the procedure shown in Fig. 2), then transformed into a nonterminal formula, removed from $\mathcal{G}$ and added to $\mathcal{S}$. The relation $<$ is extended so that the newly added formula is smaller than the semantical rule for the `while` instruction, since the two rules can be applied concurrently.

At the second (and final) iteration, (1) is verified. The graph-construction procedure exploits the fact that (2) is minimal in $\mathcal{S}$ and thus it will be applied instead of the semantical rule for `while`, producing a finite graph by avoiding an infinite loop unfolding, and allowing Corollary 1 to establish that (1) is valid.

5 Incrementally Verifying the KMP Algorithm

The KMP (Knuth-Morris-Pratt) algorithm is a linear-time string-matching algorithm. The algorithm optimises the naive search of a pattern P into a text T by using some additional information collected from the pattern.

For instance, let us consider $T = \texttt{ABADABCDA}$ and $P = \texttt{ABAC}$. It can be easily observed that ABAC does not match ABADABCDA starting with the first position because there is a mismatch on the fourth position, namely $\texttt{C} \neq \texttt{D}$. A naive algorithm, after having detected this, would restart the matching process of P at the second position of T (which fails immediately) then at the third one, where it woud first match an A before detecting another mismatch (between B and D). The KMP optimises this by comparing directly the B and D, as it "already knows" that they are both preceded by A, thereby saving one redundant comparison.

The overall effect is that the worst-case complexity of KMP is determined by the sum of the lengths of P and T, whereas that of a naive algorithm is determined by the product of the two lengths.

The KMP algorithm pre-processes the pattern P by computing a so-called *prefix function* π. Let P_j denote the subpattern of P up to a position j. For such position j, $\pi(j)$ equals the *length of the longest proper prefix of* P_j*, which is also a suffix of* P_j. In the case of a mismatch between the position i in T and the position j in P, the algorithm proceeds with the comparison of the positions i and $\pi[j]$. This is why, in the above example, KMP direcly compared the B and D.

We prove that the KMP algorithm is correct, i.e., given a non-empty pattern P and a non-empty string T, the algorithm finds *all* the occurrences of P in T. We use the incremental method presented in Sect. 4 on an encoding of KMP in the IMP+ language formally defined in Maude (cf. Sect. 2).

The program is shown in Fig. 3. Its specification uses the following notions:

Definition 11. – *P_j denotes the prefix of P up to (and including) j. P_0 is the empty string ϵ. If a string P' is a strict suffix of P we write $P' \sqsupset P$.*
- *The prefix function for P is $\pi : \{1, \ldots, m\} \to \{0, \ldots, m-1\}$ defined by $\pi(i) = max\{j \mid 0 \leq j < i \wedge P_j \sqsupset P_i\}$. We let $\pi^*(q) = \{\pi(q), \pi(\pi(q)), \ldots\}$.*
- *Let T be a string of length n. We define $\theta : \{1, \ldots, n\} \to \{0, \ldots, m\}$ the function which, for a given $i \in \{1, \ldots, n\}$, returns the longest prefix of P which is a suffix of T_i: $\theta(i) = max\{j \mid 0 \leq j \leq m \wedge P_j \sqsupset T_i\}$.*
- *Let T be a string of length n and Out a list. The function $allOcc(Out, P, T, i)$ returns true iff the list Out contains all the occurrences of P in $T[1..i]$.*

The grey-text annotations, written as pre/post conditions and invariants, are syntactical sugar for RL formulas. The annotations are numbered (C_1 to C_6) according to the order in which the RL formulas are verified by our incremental method. So, for example, the annotation for the inner loop of the computePrefix function is the first to be verified, and corresponds to an RL formula for the form

$$\langle \texttt{while C do} \ldots \texttt{endwhile}, \ldots \rangle \wedge C \wedge C_1 \Rightarrow \langle \texttt{skip}, \ldots \rangle \wedge \neg C \wedge C_1$$

where while C do ... endwhile denotes the inner loop of computePrefix. Similarly, the specification of the KMP program is an RL formula of the form:

$$\langle \texttt{KMP}, \ldots, \texttt{.Ints} \rangle \wedge C_6 \Rightarrow \langle \texttt{skip}, \ldots, \texttt{Out} \rangle \wedge allOcc(\texttt{Out}, \ldots),$$

where KMP denotes the whole program, .Ints denotes an empty list of integers (cf. Sect. 2), Out is a list of integers denoting the program's output, and $allOcc(\texttt{Out}, \ldots)$ states that Out contains all positions of the pattern in the text.

The RL formulas corresponding to the annotations ($C_1 \ldots C_6$) were verified in the given order. Once a formula was verified, it was generalised (cf. Remark 2)

```
/*C3: m >= 1 */
function computePrefix(){
  k := 0;
  pi[1] := 0;
  q := 2;
  while (q <== m) do
  /*C2: 0 <= k /\ k < q /\ q <= m+1 /\
  (forall u:1..k)(p[u]=p[q-1-k+u]) /\
  (forall u:1..q-1)(pi[u]=Pi(u)) /\
  (forall j)((j > k /\ j in Pi*(q-1))
            implies p[j+1] != p[q]) /\
   k in Pi*(q-1) /\ Pi(q)<=k+1  */
    while !(k <== 0) &&
        !(p[k ++ 1] === p[q]) do
    /*C1: 0 <= k /\ k < q /\ q <= m /\
    (forall u:1..k)(p[u]=p[q-1-k+u]) /\
    (forall u:1..q-1)(pi[u]=Pi(u)) /\
    (forall j)((j > k /\ j in Pi*(q-1))
                implies p[j+1] != p[q]) /\
     k in Pi*(q-1) /\ Pi(q)<=k+1  */
     k := pi[k]
    endwhile
    if (p[k ++ 1] === p[q]) then
     k := k ++ 1
    else skip
    endif
    pi[q] := k;
    q := q ++ 1;
  endwhile}
/*(forall u:1..m)(pi[u]=Pi(u)) */
```

```
/*Main program  */
/*C6: m>=1 /\ n>=1 */
q := 0;
i := 1;
computePrefix();
while (i <== n) do
/C5: *1 <= m /\ 0 <=q <= m /\ 1 <= i<= n+1 /\
  (forall u:1..q)(p[u]=t[i-1-q+u]) /\
  (forall u:1..m)(pi[u]=Pi(u)) /\
  (exists v)((forall u:v+1..i-1) Theta(u)<m /\
                        allOcc(Out,p,t,v))/\
  Theta(i)<=q+1 */
  while !(q <== 0) && !(p[q ++ 1] === t[i]) do
  /*C4: 1<=m /\ 0<=q<m /\ 1 <= i <= n/\
    (forall u:1..q)(p[u]=t[i-1-q+u]) /\
    (forall u:1..m)(pi[u]=Pi(u)) /\
    (exists v)((forall u:v+1..i-1)
            Theta(u)<m /\ allOcc(Out,p,t,v))/\
    Theta(i)<=q+1 */
    q := pi[q]
  endwhile
  if (p[q ++ 1] === t[i]) then q := q ++ 1
  else skip endif;
  if (q === m) then print (i -- m) ; q := p[q]
  else skip endif;
  i := i ++ 1
endwhile
/*allOcc(Out, p, t, n) */
```

Fig. 3. The KMP algorithm in IMP+: prefix function (left) and the main program (right). Grey-text annotations are syntactic sugar for RL formulas. `Pi`, `Theta`, `allOcc`, and `Pi*` denote the functions π, θ, and *allOcc*, and the set π^* respectively (cf. Definition 11).

and added to the rules denoting the semantics of IMP+ as new, prioritary rules. Each rule verification follows the construction of a graph (cf. procedure in Fig. 2), performed by symbolic execution, implemented by rewriting as described in Sect. 3. For this purpose we have intensively use Maude's metalevel mechanisms in order to control the application of rewrite rules.

The main verification effort (besides coming up with the annotations $C_1 \dots C_6$) went into the inclusion test between patterns that occurs in our graph-construction procedure. For this purpose we have used certain properties of the π, π^*, and θ mathematical functions from [12], which we include in Maude as equations used for the purpose of simplification. Some elementary simplifications involving properties of integers and Booleans were performed via Maude's interface to the Z3 solver. Collectively, these properties can be seen as axioms that define the class of models in which the correctness of our KMP program holds.

Benefits of Incremental Verification. In earlier work [5] we attempted to verify KMP using a circular approach of the "all-or-nothing" variety. The main difficulty with such approaches is that, if verification fails, one is left with nothing: any of the formulas being (simultaneously) verified could be responsible for the failure.

The consequence was that (as we realised afterwards by revisiting the problem) our earlier verification was incorrect. We found some versions of the annotations $C_1 \dots C_6$, which, as RL formulas, would only hold under unrealistic assumptions about the problem-domain functions π, π^*, and θ.

We decided to redo the KMP verification incrementally, starting with smaller program fragments, and rigorously proving at each step the required facts about the problem domain. Our incremental approach was first a language-*dependent* one [12], as it was based on proving pre/post conditions of functions and loop invariants. Of course, not all languages have the same kinds of functions and loops; some lack such constructions altogether. The method proposed in this paper is (with some restrictions) both incremental and language-independent, is formally proved correct, and was instrumental in successfully proving the KMP program, this time, under valid assertions regarding the problem domain.

6 Conclusion, Related Work and Future Work

In this paper we propose an incremental method for proving a class of RL formulas useful in practical situations. Mainly, RL formula verification is reduced to checking two technical conditions: the first is an invariance property, while the second is related to the so-called capturing of terminal configurations. Formally, the conjunction of these conditions is shown to be equivalent to RL formula validity. We also present a graph construction procedure based on symbolic execution which, if it terminates successfully, ensures that these conditions hold for a given RL formula. The method is successfully applied on the nontrivial Knuth-Morris-Pratt algorithm for string matching, encoded in a simple imperative language. The syntax and the semantics of this language have been defined in Maude, whose reflective features were intensively used for implementation purposes.

Using the proposed approach RL formulas are proved in a systematic manner. One first proves formulas that specify sub-programs of the program under verification, and then exploits the newly proved formulas to (incrementally) prove other formulas that specify larger subprograms. By contrast, monolithic/circular approaches [1–4,6,13] attempt to prove all formulas at once, in no particular order. In case of failure, in a monolithic approach, *any* circularly dependent subset of formulas under proof might be responsible for the failure; whereas in an incremental approach, there is only one subset of formulas to consider (and to modify in order to progress in the proof): the formula currently under proof, together with some already proved valid formulas. Thus, an incremental method saves the user some effort in the trial-and-error process of program verification.

Related Work. Besides the already mentioned work on RL we cite some approaches in program verification; an exhaustive list is outside the scope of this paper.

Some approaches are based on exploring the state-space of a program, e.g., [14], in which software model checking is combined with symbolic execution and abstraction techniques to overcome state-space explosion. Our approach has some similarities with the above: we also use symbolic execution to construct a graph, which is an abstraction of the reachable state space of a program.

Some verification tools (e.g., Why3 [15]) are based on deductive methods. These tools use the program specifications (i.e., pre/post-conditions, invariants) to generate proof obligations, which are then discharged to external provers (e.g., COQ, Z3, ...). Similarly, our implementation uses a version of Maude which includes a connection to the Z3 SMT solver (used for simplifying conditions).

In the same spirit, compositional methods for the formal verification (e.g., [16]) shift the focus of verification from global to local level in order to reduce the complexity of the verification process.

Future Work. One issue that needs to be addressed is the handling of domain-specific properties. Each program makes computations over a certain domain (e.g., arrays), and in order to prove a program, certain properties of the underlying domain are required (e.g., relations between selecting and storing elements in an array). Currently, these properties are stated as axioms in Maude, and we are planning to connect Maude to an inductive prover in order to interactively prove the axioms in questions as properties satisfied by more basic definitions.

References

1. Roşu, G., Ştefănescu, A.: Towards a unified theory of operational and axiomatic semantics. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part II. LNCS, vol. 7392, pp. 351–363. Springer, Heidelberg (2012)
2. Roşu, G., Ştefănescu, A.: Checking reachability using matching logic. In: Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2012), pp. 555–574. ACM (2012)
3. Roşu, G., Ştefănescu, A., Ciobâcă, Ş., Moore, B.M.: One-path reachability logic. In: Proceedings of the 28th Symposium on Logic in Computer Science (LICS 2013), pp. 358–367. IEEE, June 2013
4. Ştefănescu, A., Ciobâcă, Ş., Mereuţă, R., Moore, B.M., Şerbănuţă, T.F., Roşu, G.: All-path reachability logic. In: Dowek, G. (ed.) RTA-TLCA 2014. LNCS, vol. 8560, pp. 425–440. Springer, Heidelberg (2014)
5. Arusoaie, A., Lucanu, D., Rusu, V.: A generic framework for symbolic execution: theory and applications. Research report RR-8189. Inria, September 2015
6. Arusoaie, A., Lucanu, D., Rusu, V.: A generic framework for symbolic execution. Research report RR-8189. Inria, September 2015. https://hal.inria.fr/hal-00766220
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
8. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. Roşu, G., Ellison, C., Schulte, W.: Matching logic: an alternative to Hoare/Floyd logic. In: Johnson, M., Pavlovic, D. (eds.) AMAST 2010. LNCS, vol. 6486, pp. 142–162. Springer, Heidelberg (2011)
10. Roşu, G., Ştefănescu, A.: Matching logic: a new program verification approach (NIER track). In: ICSE 2011: Proceedings of the 30th International Conference on Software Engineering, pp. 868–871. ACM (2011)

11. Roşu, G.: Matching logic — extended abstract. In: Proceedings of the 26th International Conference on Rewriting Techniques and Applications (RTA 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 36, pp. 5–21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, July 2015
12. Verification of the KMP algorithm. https://fmse.info.uaic.ro/imgs/kmp.pdf
13. Lucanu, D., Rusu, V., Arusoaie, A., Nowak, D.: Verifying reachability-logic properties on rewriting-logic specifications. In: Martí-Oliet, N., Ölveczky, P.C., Talcott, C. (eds.) Meseguer Festschrift. LNCS, vol. 9200, pp. 451–474. Springer, Heidelberg (2015)
14. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)
15. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013)
16. de Roever, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, Cambridge (2001)