Reachability Logic in $\mathbb{K}$

Andrei Ștefănescu[1], Ștefan Ciobâcă[2], Brandon M. Moore[1], Traian Florin Șerbănuță[3], and Grigore Roșu[1,2]

[1] University of Illinois at Urbana-Champaign, USA
{stefanes1,bmoore,grosu}@illinois.edu
[2] University "Alexandru Ioan Cuza", Romania, stefan.ciobaca@info.uaic.ro
[3] University of Bucharest, Romania

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

1 Introduction

Operational semantics are easy to define and understand. Giving a language an operational semantics can be regarded as "implementing" a formal interpreter. Operational semantics require little formal training, scale up well and, being executable, can be tested. Thus, operational semantics are typically used as trusted reference models for the defined languages. Despite these advantages, operational semantics are rarely used directly for program verification (i.e. verifying properties of a given program, rather than performing meta-reasoning about a given language), because such proofs tend to be low-level and tedious, as they involve formalizing and working directly with the corresponding transition system. Hoare or dynamic logics allow higher level reasoning at the cost of (re)defining the language as a set of abstract proof rules, which are harder to understand and trust. The state-of-the-art in mechanical program verification is to develop and prove such language-specific proof systems sound w.r.t to a trusted operational semantics [1,2,3], but that needs to be done for each language separately.

Defining more semantics for the same language and proving the soundness of one semantics in terms of another are highly uneconomical tasks when real programming languages are concerned, often taking several years to complete. Ideally, we would like to have only one semantics for a language, together with a generic theory and a set of generic tools and techniques allowing us to get all the benefits of any other semantics without paying the price of defining other semantics. Recent work [4,5,6,7] shows this *is possible*, by proposing a *language-independent proof system* which derives program properties directly from an operational semantics, *at the same proof granularity and compositionality* as a language-specific axiomatic semantics. Specifically, it introduces *(one-path) reachability rules*, which generalize both operational semantics reduction

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

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

Using *matching logic* [8] as a configuration specification formalism (Section 3), where a pattern φ specifies all program configurations that match it, we first introduce the novel notion of an *all-path reachability rule* $\varphi \Rightarrow^{\forall} \varphi'$ in Section 4, where φ and φ' are matching logic patterns. Rule $\varphi \Rightarrow^{\forall} \varphi'$ is valid iff any program configuration satisfying φ reaches, on any terminating execution path, some configuration satisfying φ'. This subsumes partial-correctness in non-deterministic languages. We then present a proof system for deriving an all-path $\varphi \Rightarrow^{\forall} \varphi'$ or one-path $\varphi \Rightarrow^{\exists} \varphi'$ reachability rule from a set $\mathcal{S}$ of semantics rules (Section 5). $\mathcal{S}$ consists of reduction rules $\varphi_l \Rightarrow^{\exists} \varphi_r$, where φ_l and φ_r are simple patterns as encountered in operational semantics (Section 2), which can be non-deterministic. The proof system derives more general sequents "$\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^Q \varphi'$", with $\mathcal{A}$ and C two sets of reachability rules and $Q \in \{\forall, \exists\}$. Intuitively, $\mathcal{A}$'s rules (*axioms*) are already established valid, and thus can be immediately used. Those in C (*circularities*) are only claimed valid, and can be used only after taking execution steps based on the rules in $\mathcal{S}$ or $\mathcal{A}$. The most important proof rules are

Step$\forall$:

$$\frac{\models \varphi \rightarrow \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists \mathit{FreeVars}(\varphi_l).\varphi_l \quad \models \exists c\, (\varphi[c/\Box] \wedge \varphi_l[c/\Box]) \wedge \varphi_r \rightarrow \varphi' \quad \text{for each } \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}}{\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{\forall} \varphi'}$$

Circularity :

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

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

We illustrate on examples how our proof system enables execution, state exploration, (similar to symbolic model-checking), and verification of program properties (Section 2). We also present an application to polymorphic type inference (Section 8). We show that our proof system is sound and relatively complete (Section 6).

Contributions This paper makes the following specific contributions:

1. A language-independent reachability logic for deriving all-path reachability, with proofs of its soundness and relative completeness; the soundness result has also been mechanized in Coq, to serve as a foundation for certifiable verification.
2. An implementation and preliminary evaluation of it as part of the $\mathbb{K}$ framework [9].

2 Execution and Verification using Reachability Logic in $\mathbb{K}$

$\mathbb{K}$ [9] is a modular semantic language design and analysis framework, which was used for teaching programming languages at several universities and to give semantics to languages such as C, Python, Java, etc. We refer the reader to `http://kframework.org` for $\mathbb{K}$'s implementation, links to language definitions, and a tutorial. We have reimplemented $\mathbb{K}$ as a faithful projection of reachability logic, specifically as a collection of heuristics and optimizations to achieve proof search within the sound and complete eight-rule proof system in Section 5. Section 7 discusses details of the new implementation. We here only focus on how to use $\mathbb{K}$, and how its commands reduce to reachability logic proof search. The role of this section is twofold: to motivate the subsequent theoretical developments, and to show that reachability logic is natural and practical.

In a nutshell, reachability logic can be thought of as generic symbolic execution combined with circular reasoning. Symbolic execution is achieved by rewriting modulo domain reasoning. In $\mathbb{K}$, the user provides both language definitions and proof tasks as sets of reachability rules. To simplify the writing of reachability rules and to increase modularity, $\mathbb{K}$ introduces a series of notations, briefly explained below. For domain reasoning, it uses a combination of matching, unification and SMT solving (Z3).

2.1 Defining $\mathbb{K}$ Semantics

We illustrate how to write $\mathbb{K}$ definitions by means of defining IMP++, a simple concurrent imperative language (also part of the $\mathbb{K}$ distribution and tutorial). Fig. 1 shows the IMP++ syntax and Fig. 2 the semantics. $\mathbb{K}$ definitions are structured using modules, which can contain imports statements to include other modules (line 31), syntax definitions (e.g., lines 2–27), configuration declarations (lines 33–44), and $\mathbb{K}$ rules.

Syntax definitions describe a CFG in a BNF-style extended with priorities and associativity filters used for disambiguation. Terminals are enclosed in quotes, > separates priority levels, and | separates productions with the same priority. Productions can have comma-separated tags; e.g., bracket on line 14 says that parentheses are to be used only for grouping purposes, and left on line 6 that addition is left associative. $\mathbb{K}$ generates parsers from syntax definitions, which are used to parse both programs and rules. IMP++ has arithmetic (AExp) and boolean (BExp) expressions, statements (Stmt), and blocks (Block). The builtin List construct (lines 25–27) specifies lists of elements of certain types (here Id, AExp, or Stmt) separated by a certain terminal (comma, for lines 25 and 26) or just whitespace (the empty terminal on line 27).

Computations extend syntax with a task sequentialization operation, "$\curvearrowright$" (having unit $\cdot$). A task can be either a fragment of syntax to be processed or a semantic task, such as the recovery of an environment. Most of the manipulation of the computation is abstracted away from the language designer via intuitive syntax annotations.

Strictness annotations. The strict tag specifies the evaluation strategy of the corresponding construct. E.g., strict on line 6 says that the arguments of + must be evaluated before + itself is evaluated, and strict (2) on line 8 says that only the second argument of the assignment is to be evaluated. $\mathbb{K}$ generates rules from these strictness annotations transforming the syntax into tasks (and back) to ensure the proper order of evaluation.

Configuration declarations describe the initial state of the execution environment as a nested multiset/bag of cells. The nested nature of cells resembles that of molecules and

```
1  module IMP-SYNTAX
2    syntax AExp ::= Int | String | Id
3                  | "++" Id
4                  | "read" "(" ")"
5                  > AExp "/" AExp                    [left , strict ]
6                  > AExp "+" AExp                    [left , strict ]
7                  > "spawn" Block
8                  > Id "=" AExp                        [strict (2)]
9                  | "(" AExp ")"                        [bracket]
10   syntax BExp ::= Bool
11                 | AExp "≤" AExp                      [strict ]
12                 | "!" BExp                           [strict ]
13                 > BExp "&&" BExp                  [left , strict (1)]
14                 | "(" BExp ")"                       [bracket]
15   syntax Block ::= "{" Stmts "}"
16   syntax Stmt  ::= Block
17                 | AExp ";"                           [strict ]
18                 | "if " "(" BExp ")" Block "else " Block  [strict (1)]
19                 | "while" "(" BExp ")" Block
20                 | "int " Ids ";"
21                 | "print " "(" AExps ")" ";"         [strict ]
22                 | "halt " ";"
23                 > "join " AExp ";"                   [strict ]
24
25   syntax Ids   ::= List {Id,","}                     [strict ]
26   syntax AExps ::= List {AExp,","}                   [strict ]
27   syntax Stmts ::= List {Stmt,""}
28 endmodule
```

Fig. 1. IMP++ language syntax

membranes in the CHAM soup [10], albeit our cells are named. Cells are written using an XML-like notation and can contain list/sets/maps/bags as well as computations.

The IMP++ configuration consists of a top cell T (lines 33–44), which contains a cell holding the execution threads (lines 34–40), the shared store (line 41), and cells for I/O (lines 42–43). The threads cell holds multiple thread cells, each containing a computation cell k (line 36), an environment (line 37) mapping local variables to store locations, and a thread identifier cell id. The k cell usually appears in all definitions and has a special status among cells, holding the computation and effectively directing the execution. Variables in the configuration declaration (e.g., $PGM:Exp on line 36) must be set by the $\mathbb{K}$ tool when initializing the execution environment (e.g., $PGM is initialized with the AST of the program to be executed). The cell attribute color is used for displaying purposes, the stream attribute links the contents of the cell to the specified buffer for interactive I/O, and the multiplicity attribute specifies that multiple instances of that cell can coexist. The IMP++ configuration is relatively simple, containing only 9 cells and three nesting levels. The configuration of C has 75 cells and 5 nesting levels [11].

$\mathbb{K}$ **rules** use configuration patterns with variables to describe transitions between configurations, together with an aggressive configuration abstraction mechanism to minimize the size of rules and to increase their modularity. First, rules not mentioning any

```
module IMP
  imports IMP-SYNTAX
  syntax KResult ::= Int | Bool | String
  configuration <T color="yellow">
                  <threads color="orange">
                    <thread multiplicity="*" color="blue">
                      <k color="green"> $PGM:Stmts </k>
                      <env color="LightSkyBlue"> .Map </env>
                      <id color="black"> 0 </id>
                    </thread>
                  </threads>
                  <store color="red"> .Map </store>
                  <in color="magenta" stream="stdin">.List </in>
                  <out color="Orchid" stream="stdout">.List </out>
                </T>
  rule <k> X:Id => I ...</k> <env>... X ↦N ...</env> <store>... N ↦I ...</store>
  rule <k> ++X => I +Int 1 ...</k>
       <env>... X ↦N ...</env> <store>... N ↦(I => I +Int 1) ...</store>
  rule <k> read() => I ...</k> <in> ListItem(I:Int) => . ...</in>
  rule I1:Int / I2:Int => I1 /Int I2 when I2 =/=Int 0
  rule I1:Int + I2:Int => I1 +Int I2
  rule Str1:String + Str2:String => Str1 +String Str2
  rule I1:Int ≤ I2:Int => I1 ≤Int I2
  rule ! T:Bool => notBool T
  rule true && B => B
  rule false && _ => false
  rule <k> {Ss} => Ss ↷Rho ...</k> <env> Rho </env>
  rule <k> Rho => . ...</k> <env> _ => Rho </env>
  rule _:Int ; => .
  rule <k> X = I:Int => I ...</k>
       <env>... X ↦N ...</env> <store>... N ↦(_ => I) ...</store>
  rule if (true) S else _ => S
  rule if (false) _ else S => S
  rule while(B) S => if(B) {S while(B) S} else {}
  rule <k> int (X:Id, Xs => Xs); ...</k>
       <env> Rho => Rho[N/X] </env> <store>... . => N ↦0 ...</store>
  rule int . Ids; => .
  syntax Printable ::= Int | String
  syntax AExp ::= Printable
  rule <k> print(P:Printable , AEs => AEs); ...</k> <out>... . => ListItem(P) </out>
  rule print (. AExps); => .
  rule <k> halt; ↷_ => . </k>
  rule <k> spawn S => T ...</k> <env> Rho </env>
    (. => <thread>... <id> T </id> <k> S </k> <env> Rho </env> ...</thread>)
  rule <k> join(T); => . ...</k> <thread>... <k>.</k> <id>T</id> ...</thread>
  rule . Stmts => .
  rule S Ss => S ↷Ss
endmodule
```

Fig. 2. IMP++ language semantics

cell are assumed to take place at the top of the k cell, modeling that evaluation takes place only at the current redex. Second, to account for the fact that most rules collect data from several cells but only make small changes, $\mathbb{K}$ rules specify the change *in-place* by defining the matching pattern and using the rewrite symbol => *inside* that pattern to locally specify what rewrites into what. This also enables a comprehension mechanism for cells: non-changing parts of cells are abstracted away using ellipses. For example, the rule on line 45 specifies the lookup of an identifier in the store: if X is the first task in the computation cell (... says there might be other subsequent tasks), and if X is mapped to a location N in the environment (the other mappings are abstracted by ...), and if N is mapped to a value I in the store, then X changes to I, leaving the rest unchanged. Finally, $\mathbb{K}$ allows users to only mention the relevant cells in each rule, the missing cells and cell fragments being automatically inferred from the fixed configuration structure. Hence, configuration abstraction allows existing rules to stay unchanged when the configuration is extended or reorganized to accommodate new language features.

The lookup rule was explained above, but now note that it makes full use of configuration abstraction: the k, env, and store cells reside at different levels in the configuration. Variable increment (lines 46–47) and assignment (lines 59–60) are similar, but note that each performs two local rewrites. The read rule on line 48 consumes one integer from the input stream and uses it as a value for the read. Rules on lines 49-55 define the semantics for arithmetic and boolean expressions. Note that + is defined for both integers and strings, and that && is short-circuited. The block rule (line 56) saves the environment on the computation stack, to be recovered upon executing the block statements by the rule on line 57. An expression statement is discarded once the expression was evaluated (line 58). Lines 61–62 define the conditional statement, and line 63 the semantics of while through loop unrolling. The semantics of variable declarations (lines 64–66) adds a new location (N) to the store for each variable, and the variable is to that location in the environment. Note that the N variable is unbound on the left-hand side of the rule, which means that a symbolic value of the specified type will be introduced. print (lines 67–70) appends printable items to the output stream cell (lines 69–70). Halt (line 71) simply voids the computation cell. Spawn creates a new thread holding the spawned statement, the parent's environment, and a fresh (integer) identifier which is also returned to the parent thread (lines 72–73). The join rule (line 74) dissolves the join (T) statement when the thread identified by T has an empty computation. Finally, the rules on lines 75–76 desugar statement sequences into task sequences.

$\mathbb{K}$ rules can introduce symbolic variables in the configuration to rewrite, and are *unconditional*, i.e., there are no premises that need to be recursively reduced to apply a rule. Boolean side conditions are allowed, like in the rule for division, but those are moved into constraints on the rule left and right patterns when regarding the $\mathbb{K}$ rule as a reachability logic rule (see Section 4) and are handled by the underlying SMT solver.

2.2 Executing and Verifying Programs

Here we show how $\mathbb{K}$ executes and proves programs correct using the proof system of reachability logic. We implemented a wrapper, called `krun`, which uses the proof system in Fig. 4 in different ways (details are given below for each example).

Consider the IMP++ program `SUM` (Fig. 3(a)). `krun` in default mode simply executes this program for concrete values of `n`. With the initialization "n = 100;",

```
1  s = 0;
2  while (n > 0) {
3    s = s + n;
4    n = n + -1
5  }
```
(a)

```
1  int x;
2  x = 0;
3  spawn {
4    x = x + 1;
5  };
6  x = x + 1;
```
(b)

```
1  int x, f0, f1, turn;
2  x = 0;
3  f0 = 0;
4  f1 = 0;
5  spawn {
6    f0 = 1;
7    turn = 1;
8    while(f1 == 1 && turn != 0) {}
9    x = x + 1;
10   f0 = 0;
11 }
12 f1 = 1;
13 turn = 0;
14 while(f0 == 1 && turn == 0) {}
15 x = x + 1;
16 f1 = 0;
```
(c)

Fig. 3. IMP++ programs: (a) sum.imp; (b) inc.imp; and (c) peterson.imp

```
krun sum.imp
```

yields 5050 for `s`. The default mode of `krun` uses the one-path version of the reachability proof system in Section 5, essentially executing the program along one path.

Now we can prove the functional correctness of the `SUM` program, by formally deriving the all-path reachability rule $\varphi \Rightarrow^{\forall} \varphi'$ capturing its behavior, namely

$$\begin{array}{l} \langle \text{k} \rangle \texttt{SUM} \ldots \langle / \text{k} \rangle \langle \text{env} \rangle \ldots \texttt{s} \mapsto l_s,\ \texttt{n} \mapsto l_n \ldots \langle / \text{env} \rangle \langle \text{store} \rangle \ldots l_s \mapsto 0,\ l_n \mapsto n \ldots \langle / \text{store} \rangle \wedge n \geq_{Int} 0 \\ \Rightarrow^{\forall} \ \langle \text{k} \rangle \cdot \ \ldots \langle / \text{k} \rangle \langle \text{env} \rangle \ldots \texttt{s} \mapsto l_s,\ \texttt{n} \mapsto l_n \ldots \langle / \text{env} \rangle \langle \text{store} \rangle \ldots l_s \mapsto n *_{Int} (n +_{Int} 1) /_{Int} 2,\ l_n \mapsto 0 \ \ldots \langle / \text{store} \rangle \end{array}$$

We used the $\mathbb{K}$ conventions of omitting cells which are not modified, and representing cell frames by "...". As expected, an "invariant rule" of the form $\varphi_{\text{inv}} \Rightarrow^{\forall} \varphi'$ is also needed, where φ_{inv} is the pattern (`LOOP` the while loop):

$$\begin{array}{l} \exists n' (\langle \text{k} \rangle \texttt{LOOP} \ldots \langle / \text{k} \rangle \langle \text{env} \rangle \ldots \ \texttt{s} \mapsto l_s,\ \texttt{n} \mapsto l_n \ \ldots \langle / \text{env} \rangle \\ \quad \langle \text{store} \rangle \ldots \ l_s \mapsto (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1) /_{Int} 2,\ l_n \mapsto n' \ \ldots \langle / \text{store} \rangle \wedge n' \geq_{Int} 0) \end{array}$$

We quoted "invariant rule" above because reachability logic has no specific support for pre/post conditions of functions, or loop invariants, or anything specific to any particular language. Such conventional program assertions uniformly desugar into reachability rules, which can be then derived with the reachability logic proof system. We can pass sets of such rules to `krun`, which proves each of them separately using the reachability logic proof system, allowing each of them to be used as a circularity in the Circularity proof rule. In this case, all we have to do it to put the two rules above in a file `sum_spec.k`, and then verify them both with the command

```
krun --prove sum_spec.k
```

The `--prove` option uses the all-path version of the reachability proof system in Section 5 to check that, for each rule in the file, on all terminating execution paths beginning with a configuration satisfying the left-hand side of the rule there exists some configuration satisfying the right-hand side of the rule. Since `SUM` is a deterministic program, one-path and all-path reachability are the same in this case.

Consider now the concurrent IMP++ program `INC` (Fig. 3(b)). We first show that our proof system enables exhaustive state exploration, similar to symbolic model-checking but based on the operational semantics. Although humans prefer to avoid such explicit proofs and instead methodologically use abstraction or compositional reasoning whenever possible, which can also be done with our proof system, a complete proof system must nevertheless support them. `INC` exhibits a race on `x`: its value increases by 1 when both reads happen before writes, and by 2 otherwise. `krun` can exhaustively search:

```
krun --search inc.imp
```

Two solutions are detected, where `x` increases by 1 or by 2. The search uses the one-path reachability proof system on all paths, that is, all execution paths end in one of the final states reported by the search. The all-path rule that captures the behavior of `INC` is

$$
\begin{array}{l}
\quad \langle \text{threads} \rangle \langle \text{thread} \rangle \ldots \langle \text{k} \rangle \texttt{INC} \langle / \text{k} \rangle \langle \text{env} \rangle \ldots \ \texttt{x} \mapsto l_x \ \ldots \langle / \text{env} \rangle \ \ldots \langle / \text{thread} \rangle \langle / \text{threads} \rangle \\
\quad \langle \text{store} \rangle \ldots \ l_x \mapsto m \ \ldots \langle / \text{store} \rangle \\
\Rightarrow^{\forall} \langle \text{threads} \rangle \langle \text{thread} \rangle \ldots \langle \text{k} \rangle \cdot \langle / \text{k} \rangle \langle \text{env} \rangle \ldots \ \texttt{x} \mapsto l_x \ \ldots \langle / \text{env} \rangle \ \ldots \langle / \text{thread} \rangle \\
\qquad\qquad \langle \text{thread} \rangle \ldots \langle \text{k} \rangle \cdot \langle / \text{k} \rangle \langle \text{env} \rangle \ldots \ \texttt{x} \mapsto l_x \ \ldots \langle / \text{env} \rangle \ \ldots \langle / \text{thread} \rangle \langle / \text{threads} \rangle \\
\quad \langle \text{store} \rangle \ldots \ l_x \mapsto n \ \ldots \langle / \text{store} \rangle \wedge (n = m +_{Int} 1 \vee n = m +_{Int} 2)
\end{array}
$$

As before, we verify the rule above with the command

```
krun --prove inc_spec.k
```

which uses the all-path reachability proof system.

Finally, we can use Peterson's algorithm for mutual exclusion to eliminate the race as shown in Fig. 3(c). The all-path rule that captures the new behavior is

$$
\begin{array}{l}
\quad \langle \text{threads} \rangle \langle \text{thread} \rangle \ldots \langle \text{k} \rangle \texttt{PETERSON} \langle / \text{k} \rangle \langle \text{env} \rangle \ldots \ \texttt{x} \mapsto l_x \ \ldots \langle / \text{env} \rangle \ldots \langle / \text{thread} \rangle \langle / \text{threads} \rangle \\
\quad \langle \text{store} \rangle \ldots \ l_x \mapsto m \ \ldots \langle / \text{store} \rangle \\
\Rightarrow^{\forall} \langle \text{threads} \rangle \langle \text{thread} \rangle \ldots \langle \text{k} \rangle \cdot \langle / \text{k} \rangle \langle \text{env} \rangle \ldots \ \texttt{x} \mapsto l_x \ \ldots \langle / \text{env} \rangle \ldots \langle / \text{thread} \rangle \\
\qquad\qquad \langle \text{thread} \rangle \ldots \langle \text{k} \rangle \cdot \langle / \text{k} \rangle \langle \text{env} \rangle \ldots \ \texttt{x} \mapsto l_x \ \ldots \langle / \text{env} \rangle \ldots \langle / \text{thread} \rangle \langle / \text{threads} \rangle \\
\quad \langle \text{store} \rangle \ldots \ l_x \mapsto m +_{Int} 2 \ \ldots \langle / \text{store} \rangle
\end{array}
$$

Although we here illustrated our semantics-based verification approach on a toy language for simplicity, we have experimented with instances of it in the context of more complex languages. We briefly mention MatchC [4], a one-path reachability verifier for a deterministic C fragment. Verification proceeds by applying the one-path proof system rules according to certain heuristics, deriving the given specifications from the operational semantics. The application of the proof system is implemented in Maude [12], while CVC3 [13] and Z3 [14] are used for domain reasoning (required by the Consequence rule—see Fig. 4). MatchC has efficiently verified the functional correctness of programs implementing sorting algorithms, AVL trees, the Schorr-Waite graph marking algorithm, etc., and involving arithmetic, heap data structures, I/O, etc. Users only provide program specifications and formalizations of mathematical domains used. The limitations of MatchC, which motivate this work, are that it only verifies one-path reachability and that it is specific to C; it cannot be used with another operational semantics. An online interface to MatchC, together with dozens of examples, can be found at `http://fsl.cs.illinois.edu/ml`.

3 Matching Logic

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

Matching logic is parametric in a signature and a model of configurations, making it a prime candidate for expressing program state properties in a language-independent verification framework. The configuration signature can be as simple as that of IMP in Section 2. It can also be as complex as that of the C language [11], which contains more than 70 semantic components.

We use basic concepts from multi-sorted first-order logic. Given a *signature* Σ which specifies the sorts and arities of the function symbols (constructors or operators) used in configurations, let $T_\Sigma(Var)$ 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 Var of variables, and (3) a Σ-algebra $\mathcal{T}$, the *configuration model*, which may but need not be a term algebra. As usual, $\mathcal{T}_{Cfg}$ denotes the elements of $\mathcal{T}$ of sort Cfg, called *configurations*.

Definition 1. *[8] A matching logic formula, or a **pattern**, is a first-order logic (FOL) formula which 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 $\exists$):*

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

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

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; in $(\gamma,\rho) \models \pi$ the ρ can be thought of as the "witness" of the matching, and can be further constrained in a pattern. For instance, the pattern from Section 2

$$\langle \text{k} \rangle \texttt{SUM} \ldots \langle / \text{k} \rangle \langle \text{env} \rangle \ldots \texttt{s} \mapsto l_s,\ \texttt{n} \mapsto l_n \ldots \langle / \text{env} \rangle \langle \text{store} \rangle \ldots l_s \mapsto 0,\ l_n \mapsto n \ldots \langle / \text{store} \rangle \wedge n \geq_{Int} 0$$

is matched by the configurations with code `SUM`, environment mapping program variables `s` and `n` into locations l_s and l_n, and store mapping locations l_s and l_n into integers 0 and n, with n being non-negative. Note that we use `typewriter` for program variables in *PVar* and *italic* for mathematical variables in *Var*. A pattern ψ is called *structureless* if it contains no basic patterns.

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

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

With the notation in Definition 2, $(\gamma, \rho) \models \varphi$ iff $\rho^{\gamma} \models \varphi^{\Box}$, and $\models \varphi$ iff $\mathcal{T} \models \varphi^{\Box}$. Thus, matching logic is a methodological fragment of the FOL theory of $\mathcal{T}$. We drop $\Box$ from $\varphi^{\Box}$ when it is clear in context that we mean the FOL formula instead of the matching logic pattern. It is often technically convenient to eliminate $\Box$ from φ, by replacing $\Box$ with a *Cfg* variable c and using $\varphi[c/\Box]$ instead of φ. We use the FOL representation in the Step proof rule in Fig. 4, and to establish relative completeness in Section 6.

4 Specifying Reachability

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

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

As discussed in Section 2, we assume an operational semantics is a set of (unconditional) reduction rules "$l \Rightarrow^{\exists} r$ when b", where $l, r \in T_{\Sigma,Cfg}(Var)$ and $b \in T_{\Sigma,Bool}(Var)$. Such a rule states that a ground configuration γ which is an instance of l and satisfies the Boolean condition b reduces to an instance γ' of r. Matching logic was designed to express terms with constraints: $l \wedge b$ is satisfied by exactly the γ above. Thus, we can regard such a semantics as a particular weakly well-defined reachability system $\mathcal{S}$ with rules of the form "$l \wedge b \Rightarrow^{\exists} r$". The weakly well-defined condition on $\mathcal{S}$ guarantees that if γ matches the left-hand-side of a rule in $\mathcal{S}$, then the respective rule induces an outgoing transition from γ. The transition system induced by $\mathcal{S}$ describes precisely the behavior of any program in any given state. In [6,5,4] we show that reachability rules capture one-path reachability properties and Hoare triples for deterministic languages.

Section 2 informally introduces one-path and all-path reachability rules. Formally, let us fix an operational semantics given as a reachability system $\mathcal{S}$. Then, we can specify reachability in the transition system induced by $\mathcal{S}$

Definition 4. *An **all-path reachability rule** is a pair $\varphi \Rightarrow^{\forall} \varphi'$ of patterns φ and φ'. Let $\mathcal{S}$ be a reachability system.*

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

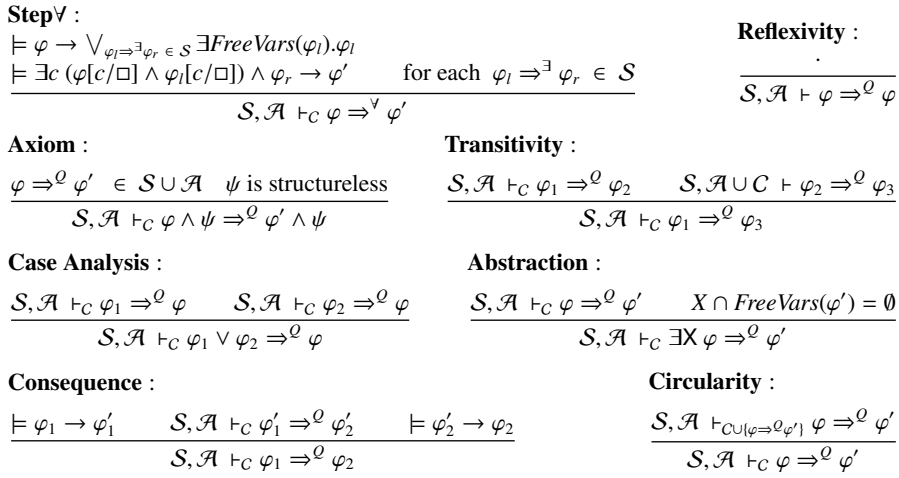

Fig. 4. Proof system for reachability. We make the standard assumption that the free variables of $\varphi_l \Rightarrow^{\exists} \varphi_r$ in the Step proof rule are fresh (e.g., disjoint from those of $\varphi \Rightarrow^{\forall} \varphi'$). Here $Q \in \{\forall, \exists\}$.

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

A Hoare triple describes the resulting state after execution finishes, so it corresponds to a reachability rule where the right side contains no remaining code. However, all-path reachability rules are strictly more expressive then Hoare triples, as they can also specify intermediate configurations (the code in the right-hand-side need not be empty) Reachability rules provide a unified representation for both language semantics and program specifications: $\varphi \Rightarrow^{\exists} \varphi'$ for semantics or one-path reachability, and $\varphi \Rightarrow^{\forall} \varphi'$ for all-path reachability specifications.

5 Reachability Proof System

Fig. 4 shows our proof system for reachability. The target language is given as a weakly well-defined reachability system $\mathcal{S}$. The soundness result (Thm. 1) guarantees that $\mathcal{S} \models \varphi \Rightarrow^{Q} \varphi'$ if $\mathcal{S} \vdash \varphi \Rightarrow^{Q} \varphi'$ is derivable. Note that the proof system derives more general sequents of the form $\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{Q} \varphi'$, where $\mathcal{A}$ and C are sets of reachability rules. The rules in $\mathcal{A}$ are called *axioms* and rules in C are called *circularities*. If either $\mathcal{A}$ or C does not appear in a sequent, it means it is empty: $\mathcal{S} \vdash_C \varphi \Rightarrow^{Q} \varphi'$ is a shorthand for $\mathcal{S}, \emptyset \vdash_C \varphi \Rightarrow^{Q} \varphi'$, and $\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^{Q} \varphi'$ is a shorthand for $\mathcal{S}, \mathcal{A} \vdash_\emptyset \varphi \Rightarrow^{Q} \varphi'$. Initially, both $\mathcal{A}$ and C are empty. During the proof, circularities can be added to C via Circularity, flushed into $\mathcal{A}$ by Transitivity, and used via Axiom.

The intuition is that the reachability rules in $\mathcal{A}$ can be assumed valid, while those in C have been postulated but not yet justified. After making progress it becomes valid

(coinductively) to rely on them. The desired semantics for sequent $\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^Q \varphi'$ (read "$\mathcal{S}$ with axioms $\mathcal{A}$ and circularities C proves $\varphi \Rightarrow^Q \varphi'$") is: $\varphi \Rightarrow^Q \varphi'$ holds if the rules in $\mathcal{A}$ hold and those in C hold after making progress, and if $C \neq \emptyset$ then φ reaches φ' after at least one transition on all complete paths when $Q = \forall$ and on at least one path when $Q = \exists$. We next discuss the proof rules.

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

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

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

Circularity has a coinductive nature, allowing us to make new circularity claims. We typically make such claims for code with repetitive behaviors, such as loops, recursive functions, jumps, etc. If 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.

One important aspect of concurrent program verification, which we do not address in this paper, is proof compositionality. Our focus here is limited to establishing a sound and complete language-independent proof system for both one-path and all-path reach-

ability rules, to serve as a foundation for further results and applications, and to discuss our current implementation of it. We only mention that we have already studied proof compositionality for earlier one-path variants of reachability logic [5], showing that there is a mechanical way to translate any Hoare logic proof derivation into a reachability proof of similar size and structure, but based entirely on the operational semantics of the language. The overall conclusion of our previous study, which we believe will carry over to all-path reachability, was that compositional reasoning can be achieved methodologically using our proof system, by proving and then using appropriate reachability rules as lemmas. However, note that this works only for well-behaved languages which enjoy a compositional semantics. For example, a language whose semantics assumes a bounded heap size, or which has constructs whose semantics involve the entire program, e.g., call/cc, will lack compositionality.

6 Soundness and Relative Completeness

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

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

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

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

This has been done for one-path reachability rules in [6,7], so we describe only the all-path case here. First we need to define a more general satisfaction relation than $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$. Let $\delta \in \{+, *\}$ be a flag and let $n \in \mathbb{N}$ be a natural number. We define a new satisfaction relation $\mathcal{S} \models^{\delta}_{n} \varphi \Rightarrow^{\forall} \varphi'$ that essentially restricts the transition system to paths of length at most n and requires φ to make progress (i.e. take steps) when $\delta = +$.

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

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

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

Lemma 1 is proved by induction on the derivation (with each induction hypotheses universally quantified over n). Consequence, Case Analysis, and Abstraction are easy.

$$step(c,\ c') \equiv \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\mu)\ (\varphi_l[c/\Box] \wedge \varphi_r[c'/\Box])$$

$$coreach(\varphi) \equiv \forall n \forall c_0 ... c_n \left(\Box = c_0 \rightarrow \bigwedge_{0 \leq i < n} step(c_i,\ c_{i+1}) \rightarrow \neg \exists c_{n+1}\ step(c_n,\ c_{n+1}) \rightarrow \bigvee_{0 \leq i \leq n} \varphi[c_i/\Box] \right)$$

Fig. 5. FOL encoding of one step transition relation and all-path reachability.

Axiom may only be used in cases where $\mathcal{S} \models^+_n \mathcal{A}$ includes $\mathcal{S} \models^+_n \varphi \Rightarrow^{\forall} \varphi'$ (as $\mathcal{S}$ contains only one-path rules). Reflexivity may only be used when C is empty, thus $\mathcal{S} \models^* \varphi \Rightarrow^{\forall} \varphi$ unconditionally. Step has premises which establish by direct domain reasoning that $\mathcal{S} \models^+ \varphi \Rightarrow^{\forall} \varphi'$. Transitivity requires considering execution paths more carefully. If C is empty, then the proof is trivial. Otherwise the induction hypothesis gives that $\varphi_1 \Rightarrow^{\forall} \varphi_2$ holds with progress. Therefore, when proving $\varphi_2 \Rightarrow^{\forall} \varphi_3$, the circularities are enabled soundly. Circularity proceeds by an inner well-founded induction on n. The induction hypothesis from the induction over the derivation requires the additional assumption that the desired rule $\varphi \Rightarrow^{\forall} \varphi'$ holds for any m strictly less than n, which is exactly the induction hypothesis provided by the induction on n. □

We next show relative completeness. The one-path case is covered in [4]. Here we prove the all-path case: any valid all-path reachability property of any program in any language with an operational semantics given as a reachability system $\mathcal{S}$ is derivable with the proof system in Fig. 4 from $\mathcal{S}$. As with Hoare and dynamic logics, "relative" means we assume an oracle capable of establishing validity in the first-order theory of the state, which here is the configuration model $\mathcal{T}$. An immediate consequence of relative completeness is that Circularity is sufficient to derive any repetitive behavior occurring in any program written in any language, and that Step is also sufficient to derive any non-deterministic behavior! We establish the relative completeness under the following assumptions: (1) $\mathcal{S}$ is finite; (2) the model $\mathcal{T}$ includes natural numbers with addition and multiplication; and (3) the set of configurations $\mathcal{T}_{Cfg}$ is countable (the model $\mathcal{T}$ includes some injective function $\alpha : \mathcal{T}_{Cfg} \rightarrow \mathbb{N}$). Assumption (1) ensures Step has a finite number of prerequisites. Assumption (2) is a standard assumption (also made by Hoare and dynamic logic completeness results) which allows the definition of Gödel's β predicate. Assumption (3) allows the encoding of a sequence of configurations into a sequence of natural numbers. We expect the operational semantics of any reasonable language to satisfy these conditions. Formally, we have the following

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

Proof (sketch — complete details in Appendix B). Our proof relies on the fact that pattern reasoning in first-order matching logic reduces to FOL reasoning in the model $\mathcal{T}$. A key component of the proof is defining the *coreach*(φ) predicate in plain FOL. This predicate holds when every complete $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path τ starting at c includes some configuration satisfying φ. We express *coreach*(φ) using auxiliary predicate *step*($c,\ c'$) which encodes the one step transition relation ($\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$). Fig. 5 shows both definitions. As it is,

coreach(φ) is not a proper FOL formula, as it quantifies over a sequence of configurations. This is addressed using the injective function α to encode universal quantification over a sequence of configurations into universal quantification over a sequence of integers, which is in turn encoded into quantification over two integer variables using Gödel's β predicate (encoding shown in the Appendix B.2).

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

7 Implementation

Here we briefly discuss our prototype implementation of reachability logic in $\mathbb{K}$. The prototype is implemented in Java, and uses Z3 [14] for domain reasoning. As seen in Section 2, the $\mathbb{K}$ tool has three modes: execution, search, and prove. Execution uses the one-path version of Reflexivity, Axiom, and Transitivity to explore one possible path, while search uses the same proof rules to exhaustively explore all paths. Prove uses the entire all-path proof system to ensure that an all-path reachability rule holds. We restrict the $\mathbb{K}$ tool to work only with rules between patterns of the form $\exists X(\pi \wedge \psi)$, with X a set of variables, π a basic pattern and ψ a structureless formula. In practice, this is enough to specify both the operational semantics and the program reachability specifications.

In prove mode, $\mathbb{K}$ accepts a set of rules to prove together. For each rule it searches starting from the left-hand side for formulae which imply the right-hand side, starting with $\mathcal{S}$ the semantics and C all the rules it attempts to prove. By a derived rule called *Set Circularity*, this suffices to show that each rule is valid. As an optimization, Axiom is given priority over Step (using specifications rather than stepping into the code).

Most work goes into implementing the Step proof rule, and in particular calculating how $\rho \models \exists c\ (\varphi[c/\Box] \wedge \varphi_l[c/\Box])$ can be satisfied. This holds when $\rho^{\gamma} \models \varphi$ and $\rho^{\gamma} \models \varphi_l$, which can be checked with unification modulo theories. The $\mathbb{K}$ tool distinguishes a number of mathematical domains (e.g. booleans, integers, sets, maps) which the underlying SMT solver can reason about. The $\mathbb{K}$ tool begins unifying π (the basic pattern of φ) and π_l (the basic pattern of φ_l) as usual. When $\mathbb{K}$ encounters corresponding subpatterns (π' in π and π'_l in π_l) which are both in one of the domains above, it records an equality $\pi' = \pi'_l$ rather than exploring the subpatterns further (if only one is in a domain, unification fails). If this stage of unification is successful, we have a conjunction ψ_u of constraints, some with one side a variable and some with both sides in one of the domains. Satisfiability of $\psi_u \wedge \psi \wedge \psi_l$ is then checked by the SMT solver. In order to be able to use Step in an automated way, the $\mathbb{K}$ tool constructs the φ' for a given φ as a disjunction of $\varphi_r \wedge \psi_u \wedge \psi \wedge \psi_l$ over each rule $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ and each way ψ_u of unifying φ with φ_l. As discussed in Section 5, in general this disjunction may not be finite, but it is sufficient for the examples that we considered.

The Consequence proof rule also requires unification modulo theories, to check validity of the implication hypothesis $\models \varphi_1 \rightarrow \varphi_1'$. The main difference from Step is that the free variables of φ' become universality quantified when sending the query to the SMT solver. The implementation of the other proof rules is straight-forward.

8 Polymorphic Type Inference using Reachability Logic in $\mathbb{K}$

Here we use the Hindley-Milner let-polymorphic type system to show that the reachability logic proof system can also be used to type programs correctly and relatively efficiently. Our "small-step" approach to defining a type system is not common, but similar definitions were proposed in the literature, e.g., in [15,16]. This section has a dual purpose: first, it shows that the language-independent implementation of reachability logic pays off, because it can be used with any rewriting definitions, even ones which reachability logic was not initially meant to handle; and second, it suggests that reachability logic has the potential to be implemented efficiently enough so that formal definitions, which are correct by construction, can also serve as implementations.

8.1 Defining the Hindley-Milner Type System in $\mathbb{K}$

Fig. 6 depicts our $\mathbb{K}$ definition of the Hindley-Milner type system, called LAMBDA.

Lines 2–14 define the syntax: one syntactic category, Exp, including integers, Bools, identifiers, the application construct, arithmetic operations, the conditional construct, λ and mu abstractions, and the let/letrec constructs. Line 15 desugars letrec into let and mu. Lines 17–22 introduce Types and TypeSchemas, extending the syntax of expressions to include types (line 23), and declaring that types are computation results (line 24).

The configuration on lines 24–27 contains a cell T holding two other cells, k (holding the computation) and tenv. The tenv cell is used in the definition of LAMBDA to map type variables to their corresponding types.

The definition rewrites builtins to their types (lines 28–29), ids to their corresponding type (line 30) or to a fresh instance of their type schema (line 31), and the λ application (line 32), arithmetic expressions (lines 33–37), and the conditional (line 38) in the most straightforward way. The type of the λ X.E abstraction (lines 39–40) is obtained by assigning a new type variable T for X and using the type environment updated with this new binding to evaluate E to its type within the function type T -> E (note that -> is strict in the second argument). Similarly to IMP++, upon evaluating E, the type environment needs to be restored. Since variable T appears only in the right-hand-side of the rewrite rule, it will be kept as a variable in the configuration term upon applying the rule, and then further constrained using unification when applying other rules.

The type of mu X.E (lines 41–42) is obtained by typing E in a type environment mapping X to a fresh type variable T, which is further constrained to also be the type of E. This post-evaluation constraint is added by transforming mu X.E to (T -> T) E, which (using the rule on line 32) evaluates to T while constraining the type of E to also be T.

The let X = T in E rule (lines 43–44) is the core of the Hindley-Milner type system, achieving polymorphism by associating a type schema to the type of X, universally quantified in all the free variables of T; these variables will be instantiated with fresh variables each time the lookup rule on line 31 applies to X.

```
module LAMBDA
  syntax Exp ::= Int | Bool | Id
               | "(" Exp ")"                          [bracket]
               | Exp Exp                              [left, strict]
               > Exp "*" Exp                          [left, strict]
               | Exp "/" Exp                          [left, strict]
               > Exp "+" Exp                          [left, strict]
               > Exp "-" Exp                          [left, strict]
               > Exp "≤" Exp                          [strict]
               > "if " Exp "then" Exp "else " Exp     [strict]
               | "λ" Id "." Exp
               | "mu" Id "." Exp
               | "let " Id "=" Exp "in" Exp           [strict (2)]
               | "letrec " Id Id "=" Exp "in" Exp
  rule letrec F:Id X:Id = E:Exp in E':Exp => let F = mu F . λ X . E in E'

  syntax Type ::= "int " | "bool"
                | Type "->" Type                      [strict (2)]
                | "(" Type ")"                        [bracket]
  syntax TypeSchema ::= "∀" Set "." Type
  syntax Exp ::= Type
  syntax KResult ::= Type

  configuration <T color="yellow">
                  <k color="green"> $PGM:Exp </k>
                  <tenv color="red"> .Map </tenv>
                </T>
  rule I:Int => int
  rule B:Bool => bool
  rule <k> X:Id => T ...</k> <tenv>... X ↦T:Type ...</tenv>
  rule <k> X:Id => #rename(T, S) ...</k> <tenv>... X ↦∀S:Set . T:Type ...</tenv>
  rule (T1:Type -> T2:Type) T1 => T2
  rule int * int => int
  rule int / int => int
  rule int + int => int
  rule int - int => int
  rule int ≤ int => bool
  rule if bool then T:Type else T => T
  rule <k> λX:Id . E:Exp => (T:Type -> E) ↷TEnv ...</k>
       <tenv> TEnv:Map => TEnv[X <- T] </tenv>
  rule <k> mu X:Id . E:Exp => (T:Type -> T) E ↷TEnv ...</k>
       <tenv> TEnv:Map => TEnv[X <- T] </tenv>
  rule <k> let X:Id = T:Type in E:Exp => E ↷TEnv ...</k>
       <tenv> TEnv:Map => TEnv[X <- ∀(#variables(T) -Set #variables (TEnv)) . T] </tenv>

  rule <k> T:Type ↷(TEnv:Map => .) ...</k>
       <tenv> _:Map => TEnv </tenv>
endmodule
```

Fig. 6. Hindley-Milner type system for lambda calculus

```
1  let f0 = λ x . λ y . x in
2    let f1 = λ x . f0 (f0 x) in
3      let f2 = λ x . f1 (f1 x) in
4        f2
```

(a)

	Time(s) for Size				
System	12	13	14	15	16
K	1.972	3.863	8.476	20.289	47.132
`ocamlc`	0.390	1.446	5.565	21.754	85.465
`ocamlc.opt`	0.060	0.254	0.908	3.161	9.845
`ghc`	0.318	0.661	1.367	2.910	5.890
`mlton`	2.657	2.929	4.565	10.065	38.859

(c)

Fig. 7. Typechecking times for Hindley-Milner example

8.2 Type Inference By Executing the Definition

Consider the sample program `exponential_type.lambda` in Fig. 7 We can use the default mode of `krun`

```
krun exponential_type.lambda
```

to infer the polymorphic type of the program, namely

$$t_0 \to (t_1 \to (t_2 \to (t_3 \to (t_4 \to t_0))))$$

The size of the type grows exponentially with the number of nested let constructs. Recall that, by default, `krun` uses the *one-path* version of the reachability proof system in Fig. 4, that is, it evaluates the program along one of the (multiple) execution paths. In this case, there is only one execution path on which the program evaluates to a type. This is not ordinary rewriting: notice that typing rules for λ and μ (lines 47-52 in Fig. 6) introduce fresh variables in the right-hand side. The intuition is that λ types to some function ($\to$) type, and further typing rules narrow the type. In the end, a program evaluates to the most general type (if it has a type) or does not evaluate to a type at all.

To characterize the performance of our implementation of reachability logic for type checking, we consider variants of the program above for different numbers of nested `let` constructs. Table 7 shows times to type-check such programs using reachability logic as implemented in $\mathbb{K}$, and a variety of other compilers. Versions used are $\mathbb{K}$'s SVN revision r10310, GHC 7.4.1, OCaml 3.12.1, and MLton 20100608. Times were measured on a lightly loaded server with an Intel Xeon E5-1660 processor. The times for $\mathbb{K}$ are its reported rewriting time, GHC's type-checking time isolated using the debugging flag `-dshow-passes`. The times reported for OCaml are compilation to bytecode, where `ocamlc` runs the compiler as interpreted bytecode, and `ocamlc.opt` runs the compiler as optimized native code (though still generating bytecode). $\mathbb{K}$ currently interprets rewrite rules. MLton time is compilation to an executable, as we didn't find suitable compiler options to isolate type-checking time. We observe at most 8x slowdown compared to `ghc`, which is the most efficient in the list, and similar performance compared to `ocamlc` and `mlton`. In general type checking/inference is not the dominant time in the compilation process, so we believe that the advantages of having a simple and straight forward type system may overweight the slowdown.

9 Related Work

We fully share the goal of the unified theory of programming initiative [17] and of the mechanical verification community to reduce the correctness of program verification to a trusted formal semantics of the target language although our methods are different. Instead of a framework to ease the task of giving multiple semantics of the same language and proving systematic relationships between them, we advocate developing frameworks which eliminate the task by requiring only one semantics (which is operational), and offering an underlying theory with the necessary machinery to achieve the benefits of multiple semantics without the costs.

Using Hoare logic [18] to prove concurrent programs correct dates back to Owicki and Gries [19]. In the rely-guarantee method proposed by Jones [20] each thread relies on some properties being satisfied by the other threads, and in its turn, offers some guarantees on which the other threads can rely. O'Hearn [21] advances a Separation Hypothesis in the context of separation logic [22] to achieve compositionality: the state can be partitioned into separate portions for each process and relevant resources, respectively, satisfying certain invariants. More recent research focuses on improvements over both of the above methods and even combinations of them (e.g., [23,24,25,26]).

The satisfaction of all-path-reachability rules can also be understood intuitively in the context of temporal logics. Matching logic formulae can be thought of as state formulae, and reachability rules as temporal formulae. Assuming CTL* on finite traces, the semantics rule $\varphi \Rightarrow \varphi'$ can be expressed as $\varphi \rightarrow E \bigcirc \varphi'$, while an all-path reachability rule $\varphi \Rightarrow^{\forall} \varphi'$ can be expressed as $\varphi \rightarrow A\Diamond\varphi'$. However, unlike in CTL*, reachability rules $\varphi \Rightarrow \varphi'$ (and all path reachability rules $\varphi \Rightarrow^{\forall} \varphi'$) share the free variables of the matching logic formulae φ and φ'. Therefore, existing proof systems for temporal logics such as the one for CTL* defined by Pnueli and Kesten are not directly comparable with our approach: CTL* formulae only have atomic predicates, while in our approach we consider only a specific temporal structure ($\varphi \rightarrow A\Diamond\varphi'$).

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

10 Conclusion and Future Work

This paper introduces a sound and (relatively) complete language-independent proof system which derives program properties holding along all execution paths (capturing partial correctness for non-deterministic programs), directly from an operational semantics. Extending this result to operational semantics given with conditional rules is left as future work. The proof system separates reasoning about deterministic language features (via the operational semantics) from reasoning about non-determinism (via the proof system). Thus, we believe that existing techniques such as rely-guarantee and concurrent separation logic could be used in conjunction with our proof system to achieve semantically grounded and compositional verification.

References

1. Nipkow, T.: Winskel is (almost) right: Towards a mechanized semantics textbook. Formal Aspects of Computing **10** (1998) 171–186
2. Jacobs, B.: Weakest pre-condition reasoning for Java programs with JML annotations. J. Logic and Algebraic Programming **58**(1-2) (2004) 61–88
3. Appel, A.W.: Verified software toolchain. In: ESOP. Volume 6602 of LNCS. (2011) 1–17
4. Roşu, G., Ştefănescu, A.: Checking reachability using matching logic. In: OOPSLA, ACM (2012) 555–574
5. Roşu, G., Ştefănescu, A.: From Hoare logic to matching logic reachability. In: FM. Volume 7436 of LNCS. (2012) 387–402
6. Roşu, G., Ştefănescu, A.: Towards a unified theory of operational and axiomatic semantics. In: ICALP. Volume 7392 of LNCS. (2012) 351–363
7. Roşu, G., Ştefănescu, A., Ciobâcă, c., Moore, B.M.: One-path reachability logic. In: LICS'13, IEEE (2013)
8. Roşu, G., Ellison, C., Schulte, W.: Matching logic: An alternative to Hoare/Floyd logic. In: AMAST. Volume 6486 of LNCS. (2010) 142–162
9. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. J. Logic and Algebraic Programming **79**(6) (2010) 397–434
10. Berry, G., Boudol, G.: The chemical abstract machine. Theoretical Computer Science **96**(1) (1992) 217–248
11. Ellison, C., Roşu, G.: An executable formal semantics of C with applications. In: POPL, ACM (2012) 533–544
12. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude. Volume 4350 of LNCS. Springer (2007)
13. Barrett, C., Tinelli, C.: CVC3. In: CAV. (2007) 298–302
14. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. Volume 4963 of LNCS. (2008)
15. Kuan, G., MacQueen, D., Findler, R.B.: A rewriting semantics for type inference. In: ESOP. Volume 4421 of Lecture Notes in Computer Science. (2007) 426–440
16. Ellison, C., Şerbănuţă, T.F., Roşu, G.: A rewriting logic approach to type inference. In: WADT. (2008) 135–151
17. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall (1998)
18. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–580
19. Owicki, S.S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. Communications of the ACM **19**(5) (1976) 279–285
20. Jones, C.B.: Specification and design of (parallel) programs. In Mason, R.E.A., ed.: Information Processing 1983: World Congress Proceedings, Elsevier (1984) 321–332
21. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theoretical Computer Science **375**(1-3) (2007) 271–307
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, IEEE (2002) 55–74
23. Feng, X.: Local rely-guarantee reasoning. In: POPL, ACM (2009) 315–327
24. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. Volume 4703 of LNCS. (2007) 256–271
25. Reddy, U.S., Reynolds, J.C.: Syntactic control of interference for separation logic. In: POPL, ACM (2012) 323–336
26. Hayman, J.: Granularity and concurrent separation logic. In: CONCUR. Volume 6901 of LNCS. (2011) 219–234

A Soundness

Definition 5. *Let $\mathcal{S}$ be a reachability system, $\varphi \Rightarrow^{Q} \varphi'$ (with $Q \in \{\forall, \exists\}$) a reachability rule and $n \in \mathbb{N}$ a natural number.*

- *We write $\mathcal{S} \models^{*}_{n} \varphi \Rightarrow^{\exists} \varphi'$ (respectively $\mathcal{S} \models^{+}_{n} \varphi \Rightarrow^{\exists} \varphi'$) when for any γ and ρ such that $(\gamma, \rho) \models \varphi$ and γ terminates in strictly less than n steps in $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$, we have that there exists γ' such that $\gamma \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'$ (respectively $\gamma \Rightarrow^{+\mathcal{T}}_{\mathcal{S}} \gamma'$) and $(\gamma', \rho) \models \varphi'$.*
- *We write $\mathcal{S} \models^{*}_{n} \varphi \Rightarrow^{\forall} \varphi'$ (respectively $\mathcal{S} \models^{+}_{n} \varphi \Rightarrow^{\forall} \varphi'$) when for any complete path $\tau = \gamma_1 ... \gamma_k$ of length $k \leq n$ and for any $\rho : Var \to \mathcal{T}$ such that $(\gamma_1, \rho) \models \varphi$ there exists $i \in \{1, ..., n\}$ (resp. $i \in \{2, ..., n\}$) such that $(\gamma_i, \rho) \models \varphi$.*

We extend the previous notations to sets of formulae: for any $\delta \in \{'+', '*'\}$, we write $S \models^{\delta}_{n} D$ to mean $S \models^{\delta}_{n} \varphi \Rightarrow^{Q} \varphi'$ for all $\varphi \Rightarrow^{Q} \varphi' \in D$.

If C is a set of reachability rules, we write "Δ_C" for "+" when C is not empty and for "$*$" when C is empty. Therefore, the relation $\models^{\Delta_C}_{n}$ should be understood as $\models^{*}_{n}$ when C is empty and $\models^{+}_{n}$ when C is not empty.

Proposition 1. *We have that $\mathcal{S} \models \varphi \Rightarrow^{Q} \varphi'$ iff for all n, $\mathcal{S} \models^{*}_{n} \varphi \Rightarrow^{Q} \varphi'$.*

Proof. By case analysis.

Lemma 2. *For any derivation tree, for any sets $\mathcal{A}$ and C of reachability rules, if the sequent $\mathcal{S}, \mathcal{A} \vdash_{C} \varphi \Rightarrow^{Q} \varphi'$ is the last sequent in the tree then for any natural number $n \in \mathbb{N} \setminus \{0\}$, if $\mathcal{S} \models^{+}_{n} \mathcal{A}$ and $\mathcal{S} \models^{+}_{n-1} C$, then $\mathcal{S} \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$.*

Proof. By induction on the proof tree for $\mathcal{S}, \mathcal{A} \vdash_{C} \varphi \Rightarrow^{Q} \varphi'$:

1. **Step**$\forall$.
   If the last rule in the proof tree is **Step**$\forall$, then Q must be $\forall$.
   Let $n \in \mathbb{N} \in \{0\}$ be an arbitrary positive natural number such that $\mathcal{S} \models^{+}_{n} \mathcal{A}$ and $\mathcal{S} \models^{+}_{n-1} C$. We assume that $\models \varphi \to \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists \mathit{FreeVars}(\varphi_l)\varphi_l$, that $\models \exists c\, (\varphi[c/\Box] \wedge \varphi_l[c/\Box]) \wedge \varphi_r \to \varphi'$ for each $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ and we show that $\mathcal{S} \models^{\Delta_C}_{n} \varphi \Rightarrow^{\forall} \varphi'$.
   Let $\tau = \gamma_1 ... \gamma_k$ be a complete path of length $k \leq n$ and let $\rho : Var \to \mathcal{T}$ be a valuation such that $(\gamma_1, \rho) \models \varphi$. We show that there exists $i \in \{1, ..., n\}$ such that $(\gamma_i, \rho) \models \varphi$.
   As we have $\models \varphi \to \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists \mathit{FreeVars}(\varphi_l)\varphi_l$ it follows that $\tau = \gamma_1$ is not a complete $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$-path and therefore $n \neq 1$. Therefore $i = 2 \in \{1, ..., n\}$ and $\gamma_2 \in \tau$. We will show that $(\gamma_2, \rho) \models \varphi'$.
   By the definition of τ, we have that $\gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_2$. By the definition of $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$, there exists $\varphi_l \Rightarrow^{\exists} \varphi_r \in S$ and a valuation $\rho' : Var \to \mathcal{T}$ such that $(\gamma_1, \rho') \models \varphi_l$ and $(\gamma_2, \rho') \models \varphi_r$.
   We have that $(\gamma_1, \rho) \models \varphi$, $(\gamma_1, \rho') \models \varphi_l$ and $(\gamma_2, \rho') \models \varphi_r$. Let $X = \mathit{FreeVars}(\varphi, \varphi')$ and $Y = \mathit{FreeVars}(\varphi_l, \varphi_r)$. We assume without loss of generality that $X \cap Y = \emptyset$. We have that $(\gamma_1, \rho[X]) \models \varphi$, $(\gamma_1, \rho'[Y]) \models \varphi_l$. Therefore $(\gamma_1, \rho[X] \uplus \rho'[Y]) \models \varphi \wedge \varphi_l$. Therefore we have that for all γ_0, $(\gamma_0, \rho[X] \uplus \rho'[Y]) \models \exists c.(\varphi[c/\Box] \wedge \varphi_l[x/\Box])$. But $(\gamma_2, \rho[X] \uplus \rho'[Y]) \models \varphi_r$ and therefore $(\gamma_2, \rho[X] \uplus \rho'[Y]) \models \exists c.(\varphi[c/\Box] \wedge \varphi_l[x/\Box]) \wedge \varphi_r$. But $\models \exists c\, (\varphi[c/\Box] \wedge \varphi_l[c/\Box]) \wedge \varphi_r \to \varphi'$ and therefore $(\gamma_2, \rho[X] \uplus \rho'[Y]) \models \varphi'$. But $\mathit{FreeVars}(\varphi') \subseteq X$ and therefore $(\gamma_2, \rho) \models \varphi'$. But this is exactly what we had to prove.

2. Axiom.
   We have that $\varphi \Rightarrow^{Q} \varphi' \in \mathcal{A}$. Let $n \in \mathbb{N} \setminus \{0\}$ be an arbitrary positive natural number such that $\mathcal{S} \models^{+}_{n} \mathcal{A}$ and $\mathcal{S} \models^{+}_{n-1} C$. We prove that $\mathcal{S} \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$.
   As $\varphi \Rightarrow^{Q} \varphi' \in \mathcal{A}$, it follows that $\mathcal{S} \models^{+}_{n} \varphi \Rightarrow^{Q} \varphi'$. But $\mathcal{S} \models^{+}_{n} \varphi \Rightarrow^{Q} \varphi'$ implies $\mathcal{S} \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$ independently of whether Δ_C is $+$ or $*$. Therefore $\mathcal{S} \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$, which is what we had to prove.
3. Reflexivity.
   Note that C is empty here. We trivially have that $\mathcal{S} \models^{*}_{n} \varphi \Rightarrow^{Q} \varphi$.
4. Transitivity.
   Let $n \in \mathbb{N} \setminus \{0\}$ be an arbitrary positive natural number such that $S \models^{+}_{n} \mathcal{A}$ and that $S \models^{+}_{n-1} C$. We prove that $S \models^{\Delta_C}_{n} \varphi_1 \Rightarrow^{Q} \varphi_3$. We distinguish on whether C is empty or not:

   – when C is empty, we have that $\Delta_C =' *'$.
     By the induction hypothesis, we have that $S \models^{*}_{n} \varphi_1 \Rightarrow^{Q} \varphi_2$ and that $S \models^{*}_{n} \varphi_2 \Rightarrow^{Q} \varphi_3$. This trivially implies that $S \models^{*}_{n} \varphi_1 \Rightarrow^{Q} \varphi_3$.
   – when C is not empty, we have that $\Delta_C =' +'$.
     By the induction hypothesis (for the first condition in the proof rule), we have that

     $$S \models^{+}_{n} \varphi_1 \Rightarrow^{Q} \varphi_2. \tag{1}$$

     By the hypothesis, we have that $S \models^{+}_{n-1} \mathcal{A} \cup C$. By the induction hypothesis (for the second condition in the proof rule) we have that

     $$S \models^{*}_{n-1} \varphi_2 \Rightarrow^{Q} \varphi_3. \tag{2}$$

     From Equation (1) and Equation (2), we immediately obtain that $S \models^{+}_{n} \varphi_1 \Rightarrow^{Q} \varphi_3$.

   In either case, we have obtained that $S \models^{\Delta_C}_{n} \varphi_1 \Rightarrow^{Q} \varphi_3$, which is what we had to prove.
5. Consequence.
   Let $n \in \mathbb{N} \setminus \{0\}$ be an arbitrary configuration such that $S \models^{+}_{n} \mathcal{A}$, that $S \models^{+}_{n-1} C$, that $\models \varphi_1 \rightarrow \varphi'_1$, that $\models \varphi'_2 \rightarrow \varphi_2$ and that $S \models^{\Delta_C}_{n} \varphi'_1 \Rightarrow^{Q} \varphi'_2$. We prove that $S \models^{\Delta_C}_{n} \varphi_1 \Rightarrow^{Q} \varphi_2$.
   We distinguish on Q:

   (a) $Q = \forall$.
   Let $\tau = \gamma_1 ... \gamma_k$ be an arbitrary complete path of length $k \leq n$ and let $\rho : Var \rightarrow \mathcal{T}$ be an arbitrary valuation such that $(\gamma_1, \rho) \models \varphi_1$. We will show that there exists $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_i, \rho) \models \varphi_2$.
   As $(\gamma_1, \rho) \models \varphi_1$ and $\models \varphi_1 \rightarrow \varphi'_1$, it follows that $(\gamma_1, \rho) \models \varphi'_1$. But we have that $S \models^{\Delta_C}_{n} \varphi'_1 \Rightarrow^{\forall} \varphi'_2$ and therefore, there exists $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_i, \rho) \models \varphi'_2$. But $\varphi'_2 \rightarrow \varphi_2$ and therefore$(\gamma_i, \rho) \models \varphi_2$, which is exactly what we had to prove.

(b) $Q = \exists$.
We will show that $S \models_n^{\Delta_C} \varphi_1 \Rightarrow^{\exists} \varphi_2$. Let $\gamma \in \mathcal{T}_{Cfg}$ be an arbitrary configuration that terminates in strictly less than n steps and let $\rho : Var \to \mathcal{T}$ be a valuation such that $(\gamma, \rho) \models \varphi_1$. As $\models \varphi_1 \to \varphi_1'$, it follows that $(\gamma, \rho) \models \varphi_1'$. But $S \models_n^{\Delta_C} \varphi_1' \Rightarrow^{\exists} \varphi_2'$ and therefore there exists $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_S^{\star_{\mathcal{T}}} \gamma'$ when $C = \emptyset$ (respectively $\gamma \Rightarrow_S^{+_{\mathcal{T}}} \gamma'$ when $C \neq emptyset$) and $(\gamma', \rho) \models \varphi_2'$. But $\models \varphi_2' \to \varphi_2$ and therefore $(\gamma', \rho) \models \varphi_2$. But this is exactly what we had to prove.

We have shown in any case that $S \models_n^{\Delta_C} \varphi_1 \Rightarrow^{Q} \varphi_2$, which is what we had to prove.

6. Case Analysis.
Let $n \in \mathbb{N} \setminus \{0\}$ be an arbitrary configuration such that $S \models_n^{+} \mathcal{A}$, that $S \models_{n-1}^{+} C$, that $S \models_n^{\Delta_C} \varphi_1 \Rightarrow^{Q} \varphi$ and that $S \models_n^{\Delta_C} \varphi_2 \Rightarrow^{Q} \varphi$. We show that $S \models_n^{\Delta_C} \varphi_1 \vee \varphi_2 \Rightarrow^{Q} \varphi$.
We distinguish two cases:
   (a) $Q = \forall$. Let $\tau = \gamma_1 ... \gamma_k$ be an arbitrary complete path of length $k \leq n$ and let $\rho : Var \to \mathcal{T}$ be an arbitrary valuation such that $(\gamma_1, \rho) \models \varphi_1 \vee \varphi_2$. We will show that there exists some $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_i, \rho) \models \varphi$.
   As $(\gamma_1, \rho) \models \varphi_1 \vee \varphi_2$, there exists $j \in \{1, 2\}$ such that $(\gamma_1, \rho) \models \varphi_j$. But by hypothesis we have that $S \models_n^{\Delta_C} \varphi_j \Rightarrow^{\forall} \varphi$. Therefore there exists $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_2, \rho) \models \varphi$, which is what we had to prove.
   (b) $Q = \exists$. Let $\gamma \in \mathcal{T}_{Cfg}$ be a configuration that terminates in strictly less than n steps and $\rho : Var \to \mathcal{T}$ a valuation such that $(\gamma, \rho) \models \varphi_1 \vee \varphi_2$. We will show that there exists some $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_S^{\star_{\mathcal{T}}} \gamma'$ when $C = \emptyset$ (respectively $\gamma \Rightarrow_S^{+_{\mathcal{T}}} \gamma'$ when $C \neq \emptyset$) such that $(\gamma', \rho) \models \varphi$.
   As $(\gamma, \rho) \models \varphi_1 \vee \varphi_2$, there exists $j \in \{1, 2\}$ such that $(\gamma, \rho) \models \varphi_j$. But $S \models_n^{\Delta_C} \varphi_j \Rightarrow^{\exists} \varphi$ and therefore there exists $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_S^{\star_{\mathcal{T}}} \gamma'$ when $C = \emptyset$ (respectively $\gamma \Rightarrow_S^{+_{\mathcal{T}}} \gamma'$ when $C \neq \emptyset$) and $(\gamma', \rho) \models \varphi$. But this is what we had to prove.

   In both cases, we have shown that $S \models_n^{\Delta_C} \varphi_1 \vee \varphi_2 \Rightarrow^{Q} \varphi$, which is what we had to prove.

7. Abstraction.
Let $n \in \mathbb{N}$ be an arbitrary positive natural number such that $S \models_n^{+} \mathcal{A}$, that $S \models_{n-1}^{*} C$, that $S \models_n^{\Delta_C} \varphi \Rightarrow^{Q} \varphi'$ and that X is a set of variables such that $X \cap FreeVars(\varphi') = \emptyset$. We show that $S \models_n^{\Delta_C} \exists X.\varphi \Rightarrow^{Q} \varphi'$.
We distinguish two cases:
   (a) $Q = \forall$. Let $\tau = \gamma_1 ... \gamma_k$ be an arbitrary complete path of length $k \leq n$ and let $\rho : Var \to \mathcal{T}$ be an arbitrary valuation such that $(\gamma_1, \rho) \models \exists X.\varphi$. We will show that there exists $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_i, \rho) \models \varphi'$.
   Because $(\gamma_1, \rho) \models \exists X.\varphi$, we have that there exists $\rho' : Var \to \mathcal{T}$ such that ρ and ρ' agree on $Var \setminus X$ and $(\gamma_1, \rho') \models \varphi$. But $S \models_n^{\Delta_C} \varphi \Rightarrow^{\forall} \varphi'$ and therefore there exists $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_i', \rho') \models \varphi'$. As ρ and ρ' agree on $Var \setminus X$ and $X \cap FreeVars(\varphi') = \emptyset$, we obtain $(\gamma_i, \rho) \models \varphi'$, which is what we had to prove.

(b) $Q = \exists$. Let $\gamma \in \mathcal{T}_{Cfg}$ be a configuration that terminates in strictly less than n steps and $\rho : Var \to \mathcal{T}$ a valuation such that $(\gamma, \rho) \models \exists X.\varphi$. We will show that there exists some $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ when $C = \emptyset$ (respectively $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ when $C \neq \emptyset$) such that $(\gamma', \rho) \models \varphi'$.
Because $(\gamma, \rho) \models \exists X.\varphi$, we have that there exists $\rho' : Var \to \mathcal{T}$ such that ρ and ρ' agree on $Var \setminus X$ and $(\gamma, \rho') \models \varphi$. But $S \models_n^{\Delta c} \varphi \Rightarrow^{\forall} \varphi'$ and therefore there exists $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ when $C = \emptyset$ (respectively $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$ when $C \neq \emptyset$) and $(\gamma', \rho') \models \varphi'$. As ρ and ρ' agree on $Var \setminus X$ and $X \cap FreeVars(\varphi') = \emptyset$, we obtain $(\gamma', \rho) \models \varphi'$, which is what we had to prove.

In both cases, we have shown that $S \models_n^{\Delta c} \exists X.\varphi \Rightarrow^{Q} \varphi'$, which is what we had to prove.

8. Circularity.
By the induction hypothesis we know that for all positive naturals $m \in \mathbb{N} \setminus \{0\}$,

$$\text{if } S \models_m^{+} \mathcal{A} \text{ and } S \models_{m-1}^{+} C \cup \{\varphi \Rightarrow^{Q} \varphi'\} \text{ then } S \models_m^{+} \varphi \Rightarrow^{Q} \varphi'. \tag{3}$$

We prove that for all positive naturals $n \in \mathbb{N} \setminus \{0\}$, $S \models_n^{+} \mathcal{A}$ and $S \models_{n-1}^{+} C$ implies $S \models_n^{\Delta c} \varphi \Rightarrow^{Q} \varphi'$, by induction on n.

(a) if $n = 1$, we trivially have that $S \models_{n-1}^{\Delta c} \varphi \Rightarrow^{Q} \varphi'$. Therefore $S \models_{n-1}^{\Delta c} C \cup \{\varphi \Rightarrow^{Q} \varphi'\}$. Applying Equation (3) with $m = n = 1$, we obtain that $S \models_n^{\Delta c} \varphi \Rightarrow^{Q} \varphi'$, what we had to show.

(b) if $n > 1$, we have that $S \models_{n-1}^{\Delta c} \varphi \Rightarrow^{Q} \varphi'$ by the inner induction hypothesis. We also have that $S \models_{n-1}^{\Delta c} C$ and therefore $S \models_{n-1}^{\Delta c} C \cup \{\varphi \Rightarrow^{Q} \varphi'\}$.
Let $m = n$ in Equation (3). We obtain that $S \models_n^{\Delta c} \varphi \Rightarrow^{Q} \varphi'$, what we had to show.

We have shown in any of the cases that for any natural number $n \in \mathbb{N} \setminus \{0\}$, if $\mathcal{S} \models_n^{+} \mathcal{A}$ and $\mathcal{S} \models_{n-1}^{+} C$, then $\mathcal{S} \models_n^{\Delta c} \varphi \Rightarrow^{Q} \varphi'$, which concludes our proof.

B Relative Completeness

Here we prove the relative completeness of the proof system in Fig. 4. Before we begin, let us recall that the relative completeness result makes the following assumptions:

Framework:
The semantics reachability system $\mathcal{S}$ is
— finite;
The configuration signature Σ has
— a sort $\mathbb{N}$;
— constant symbols 0 and 1 of $\mathbb{N}$;
— binary operation symbols + and $\times$ on $\mathbb{N}$;
— an operation symbol $\alpha : Cfg \to \mathbb{N}$.
The configuration model $\mathcal{T}$ interprets
— $\mathbb{N}$ as the natural numbers;
— operation symbols on $\mathbb{N}$ as corresponding operations;
— $\alpha : Cfg \to \mathbb{N}$ as an injective function.

Also recall that, as discussed in Section 3, matching logic is a methodological fragment of the FOL theory of the model $\mathcal{T}$. For technical convenience, in this section we work with the FOL translations $\varphi^{\square}$ instead of the matching logic formulae φ. We mention that in all the formulae used in this section, $\square$ only occurs in the context $\square = t$, thus we stay inside the methodological fragment. For the duration of the proof, we let $c, c', c_0, ..., c_n$ be distinct variables of sort *Cfg* which do not appear free in the rules in $\mathcal{S}$). We also let $\gamma, \gamma', \gamma_0, ..., \gamma_n$ range over (not necessarily distinct) configurations in the model $\mathcal{T}$, that is, over elements in $\mathcal{T}_{Cfg}$, and let ρ, ρ' range over valuations $Var \to \mathcal{T}$.

B.1 Encoding Transition System Operations in FOL

Fig. 5 shows the definition of the one step transition relation ($\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$) and of the configurations that reach φ on all and complete paths. The former is a (proper) FOL formula, while the later is not, as it quantifies over a sequence of configuration. In Section B.2 we use Gödel's β predicate to define $\overline{coreach}(\varphi)$, a FOL formula equivalent to $coreach(\varphi)$.

First, we establish the following general purpose lemma

Lemma 3. $(\rho(c), \rho) \models \varphi^{\square}$ *iff* $\rho \models \varphi^{\square}[c/\square]$.

Proof. With the notation in Definition 2, $(\rho(c), \rho) \models \varphi^{\square}$ iff $\rho^{\rho(c)} \models \varphi^{\square}$. Notice that if a valuation agrees on two variables, then it satisfies a formula iff it satisfies the formula obtained by substituting one of the two variables for the other. In particular, since $\rho^{\rho(c)}(\square) = \rho^{\rho(c)}(c)$, it follows that $\rho^{\rho(c)} \models \varphi^{\square}$ iff $\rho^{\rho(c)} \models \varphi^{\square}[c/\square]$. We notice that $\square$ does not occur in $\varphi^{\square}[c/\square]$, thus $\rho^{\rho(c)} \models \varphi^{\square}[c/\square]$ iff $\rho \models \varphi^{\square}[c/\square]$, and we are done.

The following lemma states that $step(c,\ c')$ actually has the semantic properties its name suggests.

Lemma 4. $\rho \models step(c,\ c')$ *iff* $\rho(c) \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \rho(c')$.

Proof. Assume $\rho \models step(c,\ c')$. Then, by the definition of $step(c,\ c')$, there exists some rule $\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ such that $\rho \models \exists FreeVars(\mu)\ (\varphi_l[c/\square] \wedge \varphi_r[c'/\square])$. Further, since c and c' do not occur in μ, there exists some ρ' which agrees with ρ on c and c' such that $\rho' \models \varphi_l[c/\square]$ and $\rho' \models \varphi_r[c'/\square]$. By Lemma 3, $\rho' \models \varphi_l[c/\square]$ iff $(\rho'(c), \rho') \models \varphi_l$ and $\rho' \models \varphi_r[c'/\square]$ iff $(\rho'(c'), \rho') \models \varphi_r$, so $(\rho'(c), \rho') \models \varphi_l$ and $(\rho'(c'), \rho') \models \varphi_r$. Since ρ and ρ' agree on c and c', it follows that $(\rho(c), \rho') \models \varphi_l$ and $(\rho(c'), \rho') \models \varphi_r$. By Definition 3, we conclude $\rho(c) \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \rho(c')$.

Conversely, assume $\rho(c) \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \rho(c')$. Then, by Definition 3, there exist some rule $\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ and some ρ' for which $(\rho(c), \rho') \models \varphi_l$ and $(\rho(c'), \rho') \models \varphi_r$. Further, since c and c' do not occur in μ, we can choose ρ' to agree with ρ on c and c'. Hence, $(\rho'(c), \rho') \models \varphi_l$ and $(\rho'(c'), \rho') \models \varphi_r$. By Lemma 3, $(\rho'(c), \rho') \models \varphi_l$ iff $\rho' \models \varphi_l[c/\square]$ and $(\rho'(c'), \rho') \models \varphi_r$ iff $\rho' \models \varphi_r[c'/\square]$, so $\rho' \models \varphi_l[c/\square]$ and $\rho' \models \varphi_r[c'/\square]$. Since the free variables occurring in $\varphi_l[c/\square] \wedge \varphi_r[c'/\square]$ are $FreeVars(\mu) \cup \{c, c'\}$ and ρ and ρ' agree on c and c', it follows that $\rho \models \exists FreeVars(\mu)\ (\varphi_l[c/\square] \wedge \varphi_r[c'/\square])$. By the definition of $step(c,\ c')$, we conclude $\rho \models step(c,\ c')$.

The following lemma introduces a formula encoding a complete path of fixed length.

Lemma 5. $\rho \models \bigwedge_{0 \leq i < n} step(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ step(c_n,\ c_{n+1})$ *iff* $\rho(c_0), ..., \rho(c_{n+1})$ *is a complete* $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$*-path.*

Proof. By Lemma 4, we have that $\rho(c_i) \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \rho(c_{i+1})$ iff $\rho' \models step(c_i,\ c_{i+1})$, for each $0 \leq i < n$. Further, $\rho(c_0), ..., \rho(c_{n+1})$ is complete, iff there does not exist γ such that $\rho(c_n) \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma$. Again, by Lemma 4, that is iff $\rho \models \nexists c_{n+1}\ step(c_n,\ c_{n+1})$. We conclude that $\rho \models \bigwedge_{0 \leq i < n} step(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ step(c_n,\ c_{n+1})$ iff $\rho(c_0), ..., \rho(c_{n+1})$ is a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path, and we are done.

The following lemma states that *coreach*(φ) actually has the semantic properties its name suggests.

Lemma 6. $(\gamma, \rho) \models coreach(\varphi)$ *iff for all complete* $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$*-paths* τ *starting with* γ *it is the case that* $(\gamma', \rho) \models \varphi$ *for some* $\gamma' \in \tau$*.*

Proof. First we prove the direct implication. Assume $(\gamma, \rho) \models coreach(\varphi)$, and let $\tau \equiv \gamma_0, ..., \gamma_n$ be a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path starting with γ. Then let ρ' agree with ρ on *FreeVars*(φ) such that $\rho'(n) = n$ and $\rho'(c_i) = \gamma_i$ for each $0 \leq i \leq n$. According to the definition of *coreach*(φ), we have that

$$(\gamma, \rho') \models \Box = c_0 \wedge \bigwedge_{0 \leq i < n} step(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ step(c_n,\ c_{n+1}) \rightarrow \bigvee_{0 \leq i \leq n} \varphi[c_i/\Box]$$

Since, $\gamma = \gamma_0$ and $\rho'(c_0) = \gamma_0$, it follows that $\rho' \models \Box = c_0$. Further, by Lemma 5, since $\rho'(c_0), ..., \rho'(c_n)$ is a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path, it must be the case that

$$\rho' \models \bigwedge_{0 \leq i < n} step(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ step(c_n,\ c_{n+1})$$

Thus, as $\Box$ does not occur in any $\varphi[c_i/\Box]$, we conclude that $\rho' \models \bigvee_{0 \leq i \leq n} \varphi[c_i/\Box]$, that is, $\rho' \models \varphi[c_i/\Box]$ for some $0 \leq i \leq n$. By Lemma 3, $\rho' \models \varphi[c_i/\Box]$ iff $(\gamma_i, \rho') \models \varphi$. Since ρ agrees with ρ' on *FreeVars*(φ), we conclude that $(\gamma_i, \rho) \models \varphi$.

Conversely, assume that if τ is a finite and complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path starting with γ. Then $(\gamma', \rho) \models \varphi$ for some $\gamma' \in \tau$. Let ρ' agree with ρ on *FreeVars*(φ). Then we prove that

$$(\gamma, \rho') \models \Box = c_0 \wedge \bigwedge_{0 \leq i < n} step(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ step(c_n,\ c_{n+1}) \rightarrow \bigvee_{0 \leq i \leq n} \varphi[c_i/\Box]$$

Specifically, assume $(\gamma, \rho') \models \Box = c_0 \wedge \bigwedge_{0 \leq i < n} step(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ step(c_n,\ c_{n+1})$. As $\Box$ does not occur in any cc_ic_{i+1}, by Lemma 5, it follows that $\rho'(c_0), ..., \rho'(c_n)$ is a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path. Further, $(\gamma, \rho') \models \Box = c$, implies that $\rho'(c_0), ..., \rho'(c_n)$ starts with γ. Thus, there exists some $0 \leq i \leq n$ such that $(\rho'(c_i), \rho) \models \varphi$, or equivalently, since ρ and ρ' agree on *FreeVars*(φ), such that $(\rho'(c_i), \rho') \models \varphi$. By Lemma 3, $(\rho'(c_i), \rho') \models \varphi$ iff $\rho' \models \varphi[c_i/\Box]$. Therefore, we have that $(\gamma, \rho') \models \bigvee_{0 \leq i \leq n} \varphi[c_i/\Box]$. Finally, since ρ' is an arbitrary valuation which agrees with ρ on *FreeVars*(φ), by the definition of *coreach*(φ) we can conclude that $(\gamma, \rho) \models coreach(\varphi)$, and we are done.

The following lemma established a useful property of $\mathit{coreach}(\varphi)$.

Lemma 7.

$$\models \mathit{coreach}(\varphi) \rightarrow \varphi \vee (\exists c'\ \mathit{step}(c,\ c') \wedge \forall c'\ (\mathit{step}(c,\ c') \rightarrow \mathit{coreach}(\varphi)[c'/\Box]))$$

Proof. We prove that if $(\gamma, \rho) \models \mathit{coreach}(\varphi)$ then

$$(\gamma, \rho) \models \varphi \vee (\exists c'\ \mathit{step}(c,\ c') \wedge \forall c'\ (\mathit{step}(c,\ c') \rightarrow \mathit{coreach}(\varphi)[c'/\Box]))$$

By Lemma 6, we have that for all complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-paths τ starting with γ it is the case that $(\gamma'', \rho) \models \varphi$ for some $\gamma'' \in \tau$. We distinguish two cases

- $(\gamma, \rho) \models \varphi$. We are trivially done.
- $(\gamma, \rho) \not\models \varphi$. Then γ must have $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-successors. Indeed, assume the contrary. Then $\tau \equiv \gamma$ is a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path. It follows that $(\gamma, \rho) \models \varphi$, which is a contradiction. Thus, there exists some γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$. By Lemma 4, that is iff $\rho \models \exists c'\ \mathit{step}(c,\ c')$. Further, let γ' be a $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-successor of γ and τ' a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path starting with γ'. Then, $\gamma\tau$ is a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path starting with γ. Thus, there exists some $\gamma'' \in \gamma\tau'$ such that $(\gamma'', \rho) \models \varphi$. Since $(\gamma, \rho) \not\models \varphi$, it follows that $\gamma'' \in \tau'$. Notice that γ' is an arbitrary configuration and τ' an arbitrary $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path, therefore by Lemma 6 and Lemma 3, we can conclude that $\rho \models \forall c'\ (\mathit{step}(c,\ c') \rightarrow \mathit{coreach}(\varphi)[c'/\Box]))$.

B.2 Formula Gödelization

$$\begin{aligned}\overline{\mathit{coreach}}(\varphi) \equiv\ & \forall n \forall a \forall b\ (\exists c\ (\beta(a, b, 0, \alpha(c)) \wedge \Box = c) \\ & \wedge \forall i\ (0 \leq i \wedge i < n \rightarrow \exists c \exists c'\ (\beta(a, b, i, \alpha(c)) \wedge \beta(a, b, i+1, \alpha(c')) \wedge \mathit{step}(c,\ c'))) \\ & \wedge \exists c\ (\beta(a, b, n, \alpha(c)) \wedge \nexists c'\ \mathit{step}(c,\ c')) \\ & \rightarrow \exists i\ (0 \leq i \wedge i \leq n \wedge \exists c\ (\beta(a, b, i, \alpha(c)) \wedge \varphi[c/\Box])))\end{aligned}$$

Fig. 8. FOL definition of $\mathit{coreach}(\varphi)$

Fig. 8 defines $\overline{\mathit{coreach}}(\varphi)$, the FOL equivalent of $\mathit{coreach}(\varphi)$ using Gödel's β predicate. Formally, we have the following

Lemma 8. $\models \mathit{coreach}(\varphi) \leftrightarrow \overline{\mathit{coreach}}(\varphi)$.

Proof. Let us choose some arbitrary but fixed values for n, a and b, and let ρ' such that ρ and ρ' agree on $\mathit{Var} \setminus \{n, a, b\}$ and $\rho'(n) = n$ and $\rho'(a) = a$ and $\rho'(b) = b$. According to the definition of the β predicate, there exists a unique integer sequence $j_0, ..., j_n$ such that $\beta(a, b, i, j_i)$ holds for each $0 \leq i \leq n$. Since α is injective, we distinguish two cases

- there exists some $0 \leq i \leq n$ such that there is not any γ_i with $\alpha(\gamma_i) = j_i$
- there exists a unique sequence $\gamma_0, ..., \gamma_n$ such that $\alpha(\gamma_i) = j_i$ for each $0 \leq i \leq n$.

In the former case, if $i = n$ we get that $\rho' \not\models \exists c\ (\beta(a, b, n, \alpha(c)) \wedge \nexists c'\ step(c,\ c')$ while if $0 \leq i < n$ we get that $\rho' \not\models \exists c \exists c'\ (\beta(a, b, i, \alpha(c)) \wedge \beta(a, b, i+1, \alpha(c')) \wedge step(c,\ c'))$ as in both cases we can not pick a value for c. Thus, (γ, ρ') does not satisfy left-hand-side of the implication in $\overline{coreach}(\varphi)$, and we conclude that (γ, ρ') satisfies the implication.

In the later case, we have that there is a unique way of instantiating the existentially quantified variables c and c' in each sub-formula in which they appear, as they are always arguments of the β predicate. Thus, $(\gamma, \rho') \models \exists c\ (\beta(a, b, 0, \alpha(c)) \wedge \Box = c)$ iff $\gamma = \gamma_0$. By Lemma 5, we have that

$$\begin{aligned}\rho' \models \forall i\ (0 \leq i \wedge i < n \rightarrow \exists c \exists c'\ (\beta(a, b, i, \alpha(c)) \wedge \beta(a, b, i+1, \alpha(c')) \wedge step(c,\ c'))) \\ \wedge \exists c\ (\beta(a, b, n, \alpha(c)) \wedge \nexists c'\ step(c,\ c'))\end{aligned}$$

iff $\gamma_0 ... \gamma_n$ is a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path. Finally, by Lemma 3

$$\rho' \models \exists i\ (0 \leq i \wedge i \leq n \wedge \exists c\ (\beta(a, b, i, \alpha(c)) \wedge \varphi[c/\Box]))$$

iff $(\gamma_i, \rho') \models \varphi$ for some $0 \leq i \leq n$.

We conclude that (γ, ρ') satisfies the implication in $\overline{coreach}(\varphi)$ iff

- there is no sequence $\gamma_0, ..., \gamma_n$ such that $\alpha(\gamma_i) = j_i$ for each $0 \leq i \leq n$
- the unique sequence $\gamma_0, ..., \gamma_n$ such that $\alpha(\gamma_i) = j_i$ for each $0 \leq i \leq n$ is either not starting at γ, not a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path or contains some γ' such that $(\gamma', \rho) \models \varphi$, as ρ and ρ' agree on $Var \setminus \{n, a, b\}$.

According to the property of β, for each sequence $j_0, ..., j_n$ there exist some values for a and b. Since n, a and b are chosen arbitrary, we conclude that $(\gamma, \rho) \models \overline{coreach}(\varphi)$ iff for all complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-paths τ starting at γ, there exists some $\gamma' \in \tau$ such that $(\gamma', \rho) \models \varphi$. By Lemma 6, we have that the above iff $(\gamma, \rho) \models coreach(\varphi)$, and we are done.

B.3 Encoding Semantic Validity in FOL

Lemma 9. *If* $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$ *then* $\models \varphi \rightarrow coreach(\varphi')$.

Proof. Follows from the definition of semantic validity of $\varphi \Rightarrow^{\forall} \varphi'$ and Lemma 6.

B.4 Relative Completeness

A matching logic formula ψ is *patternless* iff $\Box$ does not occur in ψ. Then we have the following lemma stating that we can derive on step on all paths

Lemma 10. $\mathcal{S}, \mathcal{A} \vdash_C \Box = c \wedge \exists c'\ step(c,\ c') \wedge \psi \Rightarrow^{\forall} \exists c'\ (\Box = c' \wedge step(c,\ c')) \wedge \psi$ *where* ψ *is a patternless formula.*

Proof. We derive the rule by applying the Step proof rule with the following prerequisites

$$\models \Box = c \wedge \exists c'\ step(c,\ c') \wedge \psi \rightarrow \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\varphi_l)\ \varphi_l$$

and for each $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ (since $\Box$ does not occur in ψ)

$$\models \exists c''\ (c'' = c \wedge \exists c'\ step(c,\ c') \wedge \varphi_l[c''/\Box]) \wedge \varphi_r \wedge \psi \rightarrow \exists c'\ (\Box = c' \wedge step(c,\ c')) \wedge \psi$$

For the first prerequisite, we have the following (using the definition of $step(c,\ c')$)

$$\begin{aligned}
& \Box = c \wedge \exists c'\ step(c,\ c') \wedge \psi \\
\rightarrow\ & \Box = c \wedge \exists c'\ step(c,\ c') \\
\leftrightarrow\ & \Box = c \wedge \exists c' \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\mu)\ (\varphi_l[c/\Box] \wedge \varphi_r[c'/\Box]) \\
\rightarrow\ & \Box = c \wedge \exists c' \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\mu)\ \varphi_l[c/\Box] \\
\rightarrow\ & \Box = c \wedge \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\varphi_l)\ \varphi_l[c/\Box] \\
\rightarrow\ & \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\varphi_l)\ \varphi_l
\end{aligned}$$

For the second prerequisite, let $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$. Then we have that

$$\begin{aligned}
& \exists c''\ (c'' = c \wedge \exists c'\ step(c,\ c') \wedge \varphi_l[c''/\Box]) \wedge \varphi_r \wedge \psi \\
\rightarrow\ & \varphi_l[c/\Box] \wedge \varphi_r \wedge \psi \\
\rightarrow\ & \exists c'\ (\Box = c' \wedge \varphi_l[c/\Box] \wedge \varphi_r[c'/\Box]) \wedge \psi \\
\rightarrow\ & \exists c'\ (\Box = c' \wedge \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} (\varphi_l[c/\Box] \wedge \varphi_r[c'/\Box])) \wedge \psi \\
\rightarrow\ & \exists c'\ (\Box = c' \wedge \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\mu)\ (\varphi_l[c/\Box] \wedge \varphi_r[c'/\Box])) \wedge \psi \\
\rightarrow\ & \exists c'\ (\Box = c' \wedge step(c,\ c')) \wedge \psi
\end{aligned}$$

and we are done.

The following three lemmas show that we can derive a rule stating that all the configurations reaching φ in the transition system actually reach φ.

Lemma 11. *If*

$$\mathcal{S}, \mathcal{A} \vdash \Box = c \wedge \exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\Box]) \Rightarrow^{\forall} \varphi$$

then $\mathcal{S}, \mathcal{A} \vdash coreach(\varphi) \Rightarrow^{\forall} \varphi$.

Proof. By Lemma 7

$$\models coreach(\varphi) \leftrightarrow \varphi \vee (\exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\Box]))$$

Thus, by Consequence and Case Analysis, it suffices to derive

$$\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^{\forall} \varphi$$
$$\mathcal{S}, \mathcal{A} \vdash \Box = c \wedge \exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\Box]) \Rightarrow^{\forall} \varphi$$

The first sequent follows by Reflexivity. The second sequent is part of the hypothesis, and we are done.

Lemma 12.

$$\mathcal{S}, \mathcal{A} \vdash \Box = c \wedge \exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\Box]) \Rightarrow^{\forall} \varphi$$

Proof. Let μ be the rule we want to derive, namely

$$\Box = c \wedge \exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\Box]) \Rightarrow^{\forall} \varphi$$

Then $\mathcal{S}, \mathcal{A} \vdash \mu$ follows by Circularity from $\mathcal{S}, \mathcal{A} \vdash_{\{\mu\}} \mu$. Hence, by Transitivity, it suffices to derive the two sequents below

$$\mathcal{S}, \mathcal{A} \vdash_{\{\mu\}} \Box = c \wedge \exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\Box]) \Rightarrow^{\forall} \varphi'$$
$$\mathcal{S}, \mathcal{A} \cup \{\mu\} \vdash \varphi' \Rightarrow^{\forall} \varphi$$

where $\varphi' \equiv \exists c'\ (\Box = c' \wedge step(c,\ c')) \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\Box])$. The first sequent follows by Lemma 10 with $\psi \equiv \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\Box])$. For the second sequent, by Abstraction with $\{c'\}$ and Consequence with

$$\models \Box = c' \wedge step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\Box]) \rightarrow coreach(\varphi)$$

it suffices to derive $\mathcal{S}, \mathcal{A} \cup \{\mu\} \vdash coreach(\varphi) \Rightarrow^{\forall} \varphi$. Then, by Lemma 11, we are left to derive

$$\mathcal{S}, \mathcal{A} \cup \{\mu\} \vdash \Box = c \wedge \exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\Box]) \Rightarrow^{\forall} \varphi$$

that is, $\mathcal{S}, \mathcal{A} \cup \{\mu\} \vdash \mu$, which trivially follows by Axiom and we are done.

Lemma 13. $\mathcal{S}, \mathcal{A} \vdash coreach(\varphi) \Rightarrow^{\forall} \varphi$.

Proof. By Lemma 11, it suffices to derive

$$\mathcal{S}, \mathcal{A} \vdash \Box = c \wedge \exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)) \Rightarrow^{\forall} \varphi$$

which follows by Lemma 12.

Theorem 3 (Relative Completeness). *If* $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$ *then* $\mathcal{S} \vdash \varphi \Rightarrow^{\forall} \varphi'$.

Proof. By Lemma 9, we have that $\models \varphi \rightarrow coreach(\varphi')$. Further, by Lemma 13, we have that $\mathcal{S} \vdash coreach(\varphi') \Rightarrow^{\forall} \varphi'$. Then the theorem follows by Consequence.