Collecting Semantics under Predicate Abstraction in the K Framework *

Irina Măriuca Asăvoae and Mihail Asăvoae

Faculty of Computer Science
Alexandru Ioan Cuza University, Iaşi, Romania
{mariuca.asavoae,mihail.asavoae}@info.uaic.ro

Abstract. The K framework is a specialization of rewriting logic for defining programming language semantics. This paper introduces the model checking with predicate abstraction technique into the K framework. To express this technique in K, we go to the foundations of predicate abstraction, that is abstract interpretation, and use its collecting semantics. As such, we propose a suitable description in K for collecting semantics under predicate abstraction of a simple imperative language. Next, we prove that our K specification for collecting semantics is a sound approximation of the K specification for concrete semantics. This work makes a further step towards the development of program verification methodologies in rewriting logic semantics project in general and the K framework in particular.

1 Introduction

Programs are expected to work correctly with respect to certain requirements. To ensure their desired behavior, one needs to be able to formally reason about programs and about programming languages. Existing formal approaches range from manually-constructed proofs to highly automated techniques. The latter includes model checking [2] and static analysis [15] as methods of ensuring correctness and finding certain classes of bugs.

In the context of software model checking, a program is translated, via convenient abstractions, into a state transition system. Abstraction helps to reduce the space size and therefore to improve the chances of a runnable/terminating verification process. However, the reduction in the number of states introduces additional behaviors, and leads to an over-approximation of the initial program.

Abstract interpretation [3] makes precise the fact that formal verification of the concrete program is reduced to verification of the simplified, abstract program, if the abstraction is sound. Several program reasoning methods make use of collecting semantics. Essentially, collecting semantics [3] abstracts each program point by set of states, and stores the collected information according to the property of interest.

* This work has been supported by Project POSDRU/88/1.5/S/47646 and by Contract ANCS POS-CCE 62 (DAK).

Predicate abstraction [6] is a popular technique to build abstract models for programs which relies on defining a set of predicates over program variables. Valid executions in the abstract representation correspond to valid executions in the original program, whereas invalid runs in the abstract semantics need to be checked for feasibility in the concrete counterpart.

The K framework [17, 19] proposes a rewrite logic-based approach specialized for the design and analysis of programming languages. A definition (or specification) of a programming language in K consists of a multiset of cells called program configuration, together with semantic sentences. A program configuration represents the structural support to define program executions. Semantic sentences include equations and rewrite rules, with equations controlling the abstraction degree and with rewrite rules controlling the observability degree of a K definition. The resulting specification is modular, semantics-based and executable. Therefore, K permits, in a unified framework, "evaluations" of both programs written in a defined programming language and the corresponding reasoning tools developed for the particular language.

In this paper we explore the potential of the K framework to define program reasoning methods. More specifically, we use K to define model checking with predicate abstraction. Because of the semantics-based characteristic of K specifications, we use collecting semantics as a means of delivering the work. As such, we propose a K description for the collecting semantics under predicate abstraction for a simple imperative language. In order to check the consistency of this K specification, we go along the lines of abstract interpretation standards, and prove that the defined programs' abstract executions are a sound approximation of the programs' concrete executions (the latter is also specified in K [19]). The present work is meant to be the incipient part of a larger project which aims to define K specifications for program analysis and verification.

Even though we frame our work in this paper within K for notational convenience, since K can be "desugared" to a large extent into rewriting logic (see [17, 19]), the results in this paper apply very well also to rewriting logic. In fact, we fully adhere to the rewriting logic project [14, 21].

Related work. There is extensive work in *software model checking*, and most of it spawned from abstract interpretation [3] and model checking [2]. If we follow the line of *predicate abstraction* that emerged in [6], we could mention only a few and important forward steps in improving the technique with counterexample-based refinement [4], localized, on-demand abstraction refinement [7], and then optimization of abstract computation using Craig interpolants [10].

Rewriting logic [12, 9] *theories* allow for nondeterminism and concurrency, while a LTL Model Checker for Maude is described in [5], and used in [1] for Java programs. A methodology for equational abstraction in the context of rewrite logic, with direct application to the Maude model checker, is proposed in [13]. An alternative (to ours) *predicate abstraction* approach is introduced to model checking under rewriting logic in [16]. A comparative study of various program semantics defined in the context of rewriting logic can be found in [21].

The K framework is extensively described in [17, 19], and it is used to define a series of languages, such as Scheme in [11], and a non-trivial object oriented language called KOOL in [8], as well as type systems, explicit state model checkers, and Hoare style program verifier [18]. The latest development within the K framework is K-Maude, a rewriting based tool for semantics of programing languages, introduced in [20].

Having the above brief history map, let us pinpoint a few correlations with the current work. In the K framework area, this paper contributes with incorporation of model checking under predicate abstraction as program meta-executions, showing that the K definitional style for concrete semantics of programming languages can be consistently used for program verification methods. However, since K is a rewrite-based framework, a question arises about how is our work positioned with respect to previously described abstractions in rewriting logic systems. Firstly, the equational abstraction creates the abstract state space via equivalence classes, which inherently introduces the overhead of equivalence checking in the infrastructure. In our case, the abstract state is denoted and calculated in the traditional predicate abstraction style (as predicates conjunction), the overhead being transferred to the specialized SMT-solver for the calculation of the abstract transition. Here we need to boast only the potential benefits our approach could bring into rewriting logic from state of the art abstraction based model checking techniques. Secondly, predicate abstraction support is already introduced in rewriting logic systems by [16]. There, the concrete transitional system is provided as a rewrite theory, which is injected with the abstraction predicates to produce the rewrite theory for the abstract transitional system. Then, model checking is performed on the latter theory. There is an important aspect of our approach, which does not seem to be easy to address with the technique in [16]: we are able to also obtain the inverse transformation, from abstract to concrete. This aspect is particularly important when the model checking in the abstract system fails to verify the property, and a refinement of the abstraction needs to be performed.

Outline of the paper. The structure of the paper is as follows: Section 2 introduces *the K framework* by defining concrete semantics for a simple imperative programming language, *SIMP*. Section 3 defines a *collecting semantics under predicate abstraction* for *SIMP*, which can be used to reason about program correctness. Section 4 states the formal correspondence between the concrete semantics and the collecting semantics of *SIMP*, as defined in K. Finally, in Section 5 we draw conclusions and present directions for further work.

2 Preliminaries

K is a rewrite logic-based framework for design and analysis of programming languages. A K specification consists of *configurations* and *rules*. The configurations, formed of K cells, are (potentially) labeled and nested structures that represent program states. The rules in K are divided into two classes: *computational rules*, that may be interpreted as transitions in a program execution,

and *structural rules*, that modify a term to enable the application of a computational rule. The K framework allows one to define modular and executable programming language semantics.

We present the K framework by means of an example - a simple imperative language *SIMP* with simple integer arithmetic, basic boolean expressions, assignments, if statements, while statements, sequential composition, and blocks. For this purpose we rely extensively on [19].

The K syntax with annotations and semantics of *SIMP* is given in Fig. 1. The left column states the *SIMP* abstract syntax, the middle column introduces a special K notation, called strictness attribute, and the right column presents the K rules for *SIMP* language semantics. Because the abstract syntax is given in a standard way, we proceed explaining, via an example, the strictness attribute called *seqstrict* (denoted here as *sq* for space efficiency). The strictness attribute that corresponds to the addition rule *Aexp* + *Aexp* is translated into the set of heating/cooling rule pairs: $a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$ and $i_1 + a_2 \rightleftharpoons a_2 \curvearrowright i_1 + \square$. These structural rules state how an arithmetic expression with addition is evaluated sequentially: first the lefthand side term (here a_1) is reduced to an integer, and only then the righthand side term a_2 is reduced to some integer. The resulted integers are added using the internal operation of integer addition $+_{Int}$ as represented by the rule $i_1 + i_2 \rightarrow i_1 +_{Int} i_2$ in the *SIMP* semantics (right column). The assignment statement has the attribute $sq(2)$ which means that strictness attribute *seqtrict* is applied only to the second argument.

The K modeling of a program configuration is a wrapped multiset of cells written $\langle c \rangle_l$, where c is the multiset of cells and l is the cell label. Examples of labels include: top $\top$, current computation k, store, call stack, output, formal analysis results, etc. The *SIMP* program configuration is:

$$\mathit{Configuration} \equiv \langle \langle K \rangle_{\mathsf{k}} \langle \mathsf{Map}[\mathit{Var} \mapsto \mathit{Int}] \rangle_{\mathsf{state}} \rangle_{\top}$$

where the top cell $\langle \ldots \rangle_{\top}$ contains two other cells: the computation $\langle K \rangle_{\mathsf{k}}$ and the store $\langle \mathsf{Map}[\mathit{Var} \mapsto \mathit{Int}] \rangle_{\mathsf{state}}$. The k cell has a special meaning in K, maintaining computational contents, much as programs or fragments of programs. The computations, i.e. terms of special sort k, are nested list structures of computational tasks. Elements of such a list are separated by an associative operator "$\curvearrowright$", as in $s_1 \curvearrowright s_2$, and are processed sequentially: s_2 is computed after s_1. The "$\cdot$" is the identity of "$\curvearrowright$". The contents of state cell is an element from $\mathsf{Map}[\mathit{Var} \mapsto \mathit{Int}]$, namely a mapping from program variables to integer values (maps are easy to define algebraically and, like lists and sets, they are considered builtins in K).

The third column in Fig. 1 contains the semantic rules of *SIMP*. The K rules generalize the usual rewrite rules, namely K rules manipulate parts of the rewrite term in different ways: write, read, and don't care. This special type of rewrite rule is conveniently represented in a bidimensional form. In this notation, the lefthand side of the rewrite rule is placed above a horizontal line and the righthand side is placed below. The bidirectional notation is flexible and concise, one could underline only the parts of the term that are to be modified. Ordinary rewrite rules are a special case of K rules, when the entire term is replaced; in this case, the standard notation $\mathit{left} \rightarrow \mathit{right}$ is used.

$AExp ::= Var \mid Int$		$\langle \underline{x}\ \_ \rangle_{\mathsf{k}}\ \langle \sigma \rangle_{\mathsf{state}}$ $\sigma[x]$
$\mid AExp + AExp \mid \ldots$	sq	$i_1 + i_2 \to i_1 +_{Int} i_2$
$BExp ::= AExp <= AExp$	sq	$i_1 <= i_2 \to i_1 \leq_{Int} i_2$
$\mid AExp = AExp$	sq	$i_1 = i_2 \to i_1 =_{Int} i_2$
$\mid$ `not` $BExp$	sq	`not` $t \to \neg_{Bool}\ t$
$\mid BExp$ `and` $BExp$	$sq(1)$	$true$ `and` $b \to b$
		$false$ `and` $b \to false$
$Stmt ::=$ `skip`		`skip` $\to \cdot$
$\mid Var := AExp$	$sq(2)$	$\langle \underline{x := i}\ \_ \rangle_{\mathsf{k}}\ \langle \_\ x \mapsto \underline{\_}\ \_ \rangle_{\mathsf{state}}$ $\cdot$ $\quad i$
$\mid Stmt\ ;\ Stmt$		$s_1; s_2 \curvearrowright s_1 \curvearrowright s_2$
$\mid \{Stmt\}$		$\{s\} \curvearrowright s$
$\mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$	$sq(1)$	`if` $true$ `then` s_1 `else` $s_2 \to s_1$
		`if` $false$ `then` s_1 `else` $s_2 \to s_2$
$\mid$ `while` $BExp$ `do` $Stmt$		$\langle$ `while` b `do` s $\rangle_{\mathsf{k}}$ `if` b `then` $\{s;$ `while` b `do` $s\}$ `else` $\cdot$
$Pgm ::=$ `vars` $\mathsf{Set}[Var];\ Stmt$		$\langle$ `vars` $xs; s \rangle_{\mathsf{k}}\ \langle \cdot \rangle_{\mathsf{state}}$ s $\quad xs \mapsto is$

Fig. 1. K syntax of *SIMP* (left) with annotations (middle) and semantics (right) with $x \in Var$, $xs \in \mathsf{Set}[Var]$, $i, i_1, i_2 \in Int$, $is \in \mathsf{Set}[Int]$, $b \in BExp$, $s, s_1, s_2 \in Stmt$

The first K rule is a *computational rule* using bidimensional notation to describe the variable lookup operation. The underlined term x in cell k means that x is a "write" term, and is to be replaced by $\sigma[x]$ from the state cell. The absence of the horizontal line under σ indicates that this is a "read" term and the state remains unchanged. The notation "$\langle\, x\ \_ \rangle$" in cell k says that x is placed in the beginning of the term contained by this cell.

The assignment rule has the statement $x := i$ as the current computation task (the first element in cell k), with a "don't care" value "$\_$" for x *somewhere* in the store (as shown by the notation $\langle \_\ x\ \_ \rangle_{\mathsf{state}}$). The value of x is updated with i in the store and the assignment is replaced by the empty computation.

The variable lookup and assignment rules are computational rules. Recall that structural rules are used to only rearrange the term to enable the application of the computational rules. One such example is the "while" rule from the right column in Fig. 1, which unfolds one step of a while-loop statement into a conditional statement. The structural transformation is represented with a dotted line to convey the idea that this transformation is lighter-weight than in computational rules. We recall that the usual rewrite rules are special cases of K rules and the K framework proposes "$\to$" for computational rules, and "$\curvearrowright$" for structural rules. The former notation is used for "if" rules, while the latter is used for the sequential composition.

The application of the initialization rule of a program (last rule in the right column in Fig. 1) leaves the computation cell containing the entire set of state-

ments, and the memory cell containing an initial mapping of program variables xs into integers. The program terminates when computation is completely consumed, meaning when the computation cell is $\langle \cdot \rangle_{\mathsf{k}}$.

```
vars x, y, err;
x := 0;   err := x;
while (y <= 0) do {
  x := x + 1;   y := y + x;   x := −1 + x;
  if not (x = 0) then err := 1 else skip;
}
```

Fig. 2. Example of a *SIMP* program

Example 1. A *SIMP* program *pgmX* is given in Fig. 2 as `vars x, y, err;` sX, where sX denotes the statements of the program. In a concrete execution, initialized with $\langle \langle sX \rangle_{\mathsf{k}} \ \langle \_\mathtt{y} \mapsto -3 \_ \rangle_{\mathsf{state}} \rangle_{\top}$, the first computational rule applied is the rule for assignments, such that the state cell becomes $\langle \_\mathtt{y} \mapsto -3, \mathtt{x} \mapsto 0 \_ \rangle_{\mathsf{state}}$. This execution terminates with $\langle \mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0, \mathtt{err} \mapsto 0 \rangle_{\mathsf{state}}$. However, if the while condition in the program is (`0 <= y`) and the program is initialized with $\langle \langle sX \rangle_{\mathsf{k}} \ \langle \_\mathtt{y} \mapsto 3 \_ \rangle_{\mathsf{state}} \rangle_{\top}$, then the execution does not terminate.

3 Collecting Semantics under Predicate Abstraction

Collecting semantics defines the set of program executions from the property of interest point of view and has several instantiations: computation traces, transitive closure of the program transition relation, reachable states, and so on. In this work we *collect forward abstract computation traces* using predicate abstraction. We describe next the details of this setting.

First, we recall the notion of abstract computation in the predicate abstraction environment. Abstraction is a mapping from a set of concrete states to an abstract state. An abstract transition between two abstract states exists if there is at least a concrete transition between concrete states from the preimage of each abstract state, respectively. In predicate abstraction, an abstract state is represented by a predicate φ. The formal definition for φ is $\varphi ::= p \in \Pi \mid \neg p \mid \varphi \wedge \varphi \mid \mathit{true} \mid \mathit{false}$, where Π is a finite set of predicates from *AtomPreds* - all atomic predicates of interest over program variables. In other words, $\varphi \in \mathcal{L}(\Pi)$ = the lattice generated by the atomic predicates from Π. Formally, this lattice is defined as $\langle \mathcal{L}(\Pi), \sqcap, \sqcup, \bot, \top \rangle$ where $\sqcap$ stands for the logic operator $\vee$, $\sqcup$ stands for the logic operator $\wedge$, while $\bot$ and $\top$ stand for *true* and *false*, respectively (more details on this can be found in [15]). Intuitively, an abstract state φ corresponds to the set of concrete states for which the values of the program variables make the formula φ true. The correspondence between the concrete states and the predicates φ is provided by a Galois connection from the powerset of all concrete states to $\mathcal{L}(\Pi)$. (Additional details on this Galois

connection are given in Section 4, where are of use.) We denote a transition in predicate abstraction by a function $post^\sharp$ standing for the abstract transition $\varphi \xrightarrow{s} post^\sharp(\varphi, s)$. The formal definition of $post^\sharp$ is in Fig. 4. We do not elaborate on it now, since it makes use of notations introduced later in this section. Of importance here is to have an understanding of the abstract computation with predicate abstraction, as this is a component of the collecting semantics.

We proceed to define in K the program meta-executions using collecting semantics under a fixed predicate abstraction. The finite set of predicates Π is given, as well as the property of interest AGϕ. (We refer meta-executions also as abstract executions.)

The *abstract configuration* in K is defined as:

$$\mathit{Configuration}^\sharp \equiv \langle \langle \langle K^\sharp \rangle_{\mathsf{k}^\sharp} \langle \mathit{State}^\sharp \rangle_{\mathsf{state}^\sharp} \langle \mathsf{List}[\mathit{Label}] \rangle_{\mathsf{path}} \rangle_{\mathsf{trace}^\sharp *} \langle \mathit{Store}^\sharp \rangle_{\mathsf{store}^\sharp} \langle \Phi \rangle_{\mathsf{inv}} \rangle_{\top^\sharp}$$

In order to have the intuition behind $\mathit{Configuration}^\sharp$, imagine that $PdcT$ is a *p*arallel *d*ivide and *c*onquer algorithm performing the *t*raversal of a digraph. $PdcT$ traverses the digraph in a standard fashion but, when it encounters a node with more than one neighbor, it is going to clone itself on each neighboring direction. The instances of $PdcT$ communicate via a shared memory where everyone deposits its own visited nodes. When an instance of $PdcT$ encounters a node existing in the shared memory, it terminates its job, as that part of the digraph is already under the administration of another instance. $\mathit{Configuration}^\sharp$ is similar to a state in the running of $PdcT$. Namely, a $\mathsf{trace}^\sharp$ cell resembles with the state of an instance of $PdcT$, and the $\mathsf{store}^\sharp$ cell resembles with the state of the shared memory. Moreover, the rules for collecting semantics under predicate abstraction, from Fig. 6, are similar with the transitions between states of $PdcT$.

Next, we provide detailed description of each cell of the K configuration for collecting semantics under predicate abstraction.

The $\mathsf{k}^\sharp$ cell maintains the "abstract" computation of the program to be verified. This is in essence the control flow graph of a program, or of a program fragment. More formally, we provide the definition of an abstract computation $K^\sharp$ as follows:

$$\langle K^\sharp \rangle_{\mathsf{k}^\sharp} \begin{cases} \mathit{Label} ::= \text{positive integers representing program points} \\ \mathit{Var} ::= \text{symbols denoting program variables} \\ \mathit{Asg} ::= \mathit{asg}(\mathit{Var}, \mathit{AExp}) \\ \mathit{Cnd} ::= \mathit{cnd}(\mathit{BExp}) \\ \mathit{TransAsg} ::= \mathit{Label} : \mathit{Asg} \\ \mathit{TransCnd} ::= \mathit{Label} : \mathit{Cnd} \\ \mathit{Trans} ::= \mathit{TransAsg} \mid \mathit{TransCnd} \\ \mathit{Ks} ::= \mathit{Trans} \mid \mathit{if}(\mathit{Ks}, \mathit{Ks}) \mid \mathit{while}(\mathit{TransCnd}, \mathit{Ks}) \mid \mathit{skip} \mid \mathsf{List}_{\curvearrowright}[\mathit{Ks}] \\ K^\sharp ::= \mathit{Ks} \curvearrowright \lfloor \mathit{Label} \rfloor \end{cases}$$

There is no obvious novelty in the abstract computation $\mathsf{k}^\sharp$ besides adding labels to each basic statement - assignments and conditions. However, a closer look shows that we categorize the statements into basic statements (*Trans*, involved in computational rules), and composed statements (*ifs* and *whiles*, involved in

structural rules). Moreover, note that after constructing the list of abstract computational tasks Ks we finalize by tailing $\lfloor Label \rfloor$ to it ($\lfloor Label \rfloor$ marks the end of the program, and provides base case computational rules). Then, this is delivered as $K^\sharp$ - the content of the abstract computation cell $\mathsf{k}^\sharp$.

$$
\begin{array}{ll}
stmt^\sharp : Label \times Stmt \to LabelK^\sharp Pair & k^\sharp : Stmt \to K^\sharp \\
|\_, \_| : K^\sharp \times Label \to LabelK^\sharp Pair & \lfloor \_ \rfloor : Label \to K^\sharp \\
\_ : \_ \ : Label \times (Asg \cup Cnd) \to Trans & \_ \curvearrowright \_ : K^\sharp \times K^\sharp \to K^\sharp
\end{array}
$$

$$
\begin{array}{l}
stmt^\sharp(\ell_{in},\ \cdot) = |\cdot,\ \ell_{in}| \\
stmt^\sharp(\ell_{in},\ \texttt{skip}; S) = stmt^\sharp(\ell_{in},\ S) \\
stmt^\sharp(\ell_{in},\ X := A; S) = |\ell_{in} : asg(X, A) \curvearrowright K,\ \ell_{fin}| \\
\qquad \textbf{if}\ |K,\ \ell_{fin}| := stmt^\sharp(\ell_{in} + 1,\ S) \\
stmt^\sharp(\ell_{in},\ S_1;\ S_2) = |K_1 \curvearrowright K_2,\ \ell_{fin}| \\
\qquad \textbf{if}\ |K_1,\ \ell_{aux}| := stmt^\sharp(\ell_{in},\ S_1)\ \text{and}\ |K_2,\ \ell_{fin}| := stmt^\sharp(\ell_{aux},\ S_2) \\
stmt^\sharp(\ell_{in},\ \{S\}) = stmt^\sharp(\ell_{in},\ S) \\
stmt^\sharp(\ell_{in},\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = |\ell_{in} : cnd(B) \curvearrowright if(K_1, K_2),\ \ell_{fin}| \\
\qquad \textbf{if}\ |K_1,\ \ell_{aux}| := stmt^\sharp(\ell_{in} + 1,\ S_1)\ \text{and}\ |K_2,\ \ell_{fin}| := stmt^\sharp(\ell_{aux},\ S_2) \\
stmt^\sharp(\ell_{in},\ \texttt{while}\ B\ \texttt{do}\ S) = |while(\ \ell_{in} : cnd(B), K),\ \ell_{fin}| \\
\qquad \textbf{if}\ |K,\ \ell_{fin}| := stmt^\sharp(\ell_{in} + 1,\ S)\ ) \\
\\
k^\sharp(S) = K \curvearrowright \lfloor \ell_{fin} \rfloor \quad \textbf{if}\ |K,\ \ell_{fin}| := stmt^\sharp(1, S)
\end{array}
$$

Fig. 3. The rewrite-based rules for abstract computation $k^\sharp$ of a *SIMP* program

In Fig. 3 we give an explicit rewrite-based method for labeling a program, and its transformation into an abstract computation (the last rule).

Example 2. The abstract computation for the program in Fig. 2 is:

$1 : asg(\mathtt{x}, 0) \curvearrowright 2 : asg(\mathtt{err}, \mathtt{x}) \curvearrowright while(3 : cnd(\mathtt{y} \leq 0), 4 : asg(\mathtt{x}, \mathtt{x}{+}1) \curvearrowright 5 : asg(\mathtt{y}, \mathtt{y}{+}\mathtt{x})$
$\curvearrowright 6 : asg(\mathtt{x}, -1{+}\mathtt{x}) \curvearrowright 7 : cnd(\neg(\mathtt{x} = 0)) \curvearrowright if(8 : asg(\mathtt{err}, 1), \cdot)) \curvearrowright \lfloor 9 \rfloor$

A $\mathsf{state}^\sharp$ cell is an abstract state which actually stands for a subset of states in the concrete execution. Since we use predicate abstraction with atomic predicates Π, the abstract state is a formula $\varphi \in \mathcal{L}(\Pi)$. However, here we prefer an equivalent representation which writes a formula in $\varphi \in \mathcal{L}(\Pi) - \{\top\}$ as $\wedge_{p \in \Pi} op(\varphi, p)$, where op is defined as:

$$
op(\varphi, p) = \begin{cases} p, \text{ if } \varphi \Rightarrow p \\ \neg p, \text{ if } \varphi \Rightarrow \neg p \\ \bot_p, \text{ otherwise} \end{cases}
$$

Obviously, *true* is $(\bot_p)_{p \in \Pi}$, and for example, if the set of atomic predicates Π is $\{\mathtt{x} \geq 0,\ \mathtt{x} = 0,\ \mathtt{y} = 1\}$, then the formula $\varphi := \mathtt{x} > 0$ is defined with the above representation as $\langle (\mathtt{x} \geq 0)\ \neg(\mathtt{x} = 0)\ \bot_{(\mathtt{y}=1)} \rangle_{\mathsf{state}^\sharp}$. Also, we recall that this abstract state corresponds to the set of concrete states which map $\mathtt{x}$ to a positive integer. Hence, we use the equivalent representation of an abstract state:

$$\langle \mathit{State}^\sharp \rangle_{\mathsf{state}^\sharp} \quad \begin{cases} \mathit{State}^\sharp ::= \mathit{Valid} \mid \mathit{False} \\ \mathit{Valid} ::= \mathsf{Map}[\Pi \mapsto \{(), \neg(), \bot_{()}\}] \end{cases}$$

In this representation, the set of predicates Π defining the abstraction is implicitly contained in a $\mathsf{state}^\sharp$ cell. Note that *False* stands for the top element of $\mathcal{L}(\Pi)$, and is actually the abstract state corresponding to the empty set of concrete states. Meanwhile, *Valid* stands for any element from $\mathcal{L}(\Pi) - \{\top\}$. Because it represents a nonempty set of concrete states, we say that $\Gamma \in \mathit{Valid}$ is a "valid" abstract state. Moreover, we make the implicit assumption that $\mathit{false} \in \mathit{False}$ is always differentiated from a valid abstract state Γ, as if *false* and Γ are cells with distinct labels (while $\mathsf{state}^\sharp$ cell can contain either of them). However, for simplicity, we do not embellish the notation any further. Finally, in Fig. 4 we define $post^\sharp$, the update operator for the abstract state.

$$post^\sharp_\phi : \mathit{Valid} \times (\mathit{Asg} \cup \mathit{Cnd}) \to \mathit{State}^\sharp \text{ is defined as follows:}$$

$$post^\sharp_\phi(\Gamma, s) = \begin{cases} \wedge_{p \in \Pi}\ post_{(\Gamma,s)}(p), \text{ if } \wedge_{p \in \Pi}\ post_{(\Gamma,s)}(p) \Rightarrow \phi \\ \mathit{false}, \text{ otherwise} \end{cases}$$

$$\text{with } \ post_{(\Gamma,cnd(b))}(p) = \begin{cases} \mathit{false}, \text{ if } \wedge_{p \in \Pi} \Gamma(p) \wedge b \Rightarrow \mathit{false} \\ p, \text{ if } \wedge_{p \in \Pi} \Gamma(p) \wedge b \Rightarrow p \\ \neg p, \text{ if } \wedge_{p \in \Pi} \Gamma(p) \wedge b \Rightarrow \neg p \\ \bot_p, \text{ otherwise} \end{cases}$$

$$\text{and } \ post_{(\Gamma,asg(x,a))}(p) = \begin{cases} p, \text{ if } \wedge_{p \in \Pi} \Gamma(p) \wedge (x' = a) \Rightarrow p[x'/x] \\ \neg p, \text{ if } \wedge_{p \in \Pi} \Gamma(p) \wedge (x' = x) \Rightarrow \neg p[x'/x] \\ \bot_p, \text{ otherwise} \end{cases}$$

Fig. 4. The update operator $post^\sharp$ for the abstract state cell $\mathsf{state}^\sharp$

A cell of type path is a list of labels which represents a trace of a possible abstract execution. Note that we refer to this as *trace* because many details are cut out from the abstract execution. Instead, we keep as representative the program points where the abstract execution took place, in their order of appearance.

We finalize the description of the $\mathsf{trace}^\sharp$ cell with the observation that $\mathsf{trace}^\sharp$ models a *forward abstract computation trace*. Namely, the cells $\mathsf{k}^\sharp$ and $\mathsf{state}^\sharp$ capture the *abstract computation*, the cell path stands for *trace*, while *forward* comes from $post^\sharp$, the abstract state update operator. Note that $\mathsf{trace}^\sharp *$ in $\mathit{Configuration}^\sharp$ indicates the existence of many $\mathsf{trace}^\sharp$ cells, and this encapsulates the *collecting* attribute of the semantics.

The content of a $\mathsf{store}^\sharp$ cell, denoted as Σ, is a set of pairs $(\!|\ell, \Gamma|\!)$ of labels from the abstract computation and elements from $\mathcal{L}(\Pi) - \{\top\}$, formally defined as $\mathit{Store}^\sharp ::= \mathsf{Set}[\mathsf{Pair}(\mathit{Label},\ \mathit{Valid})]$. The abstract store update is defined as $\Sigma[\ell \rightarrowtail \Gamma] = \Sigma \cup \{(\!|\ell, \Gamma|\!)\}$. A more standard definition for the abstract store would involve a mapping, such as $\mathit{Store}^\sharp ::= \mathsf{Map}[\mathit{Label} \mapsto \mathsf{Set}[\mathit{Valid}]]$. Note that

the two representations are equivalent. However, we prefer the former one in order to suggest the *collecting* nature of the current semantics.

An inv cell maintains the formula to be validated. In this work we restrict this formula to invariants AGϕ, where $\phi ::= p \in AtomPreds \mid \neg p \mid \phi \wedge \phi \mid false \mid true$, and A, G are the CTL operators "always" and "general". On short, AGϕ is translated as "formula ϕ is satisfied in any (abstract) state, on any computational path". Φ is defined similarly with $State^\sharp$ (Π is replaced by the set of atomic predicates from ϕ). Usually, Π includes the atomic predicates from ϕ.

$$Initialization^\sharp \equiv pgm^{\phi}_{\Pi} \rightharpoonup \langle\langle\langle k^\sharp(s)\rangle_{\mathsf{k}^\sharp}\langle \sqcap\{\varphi{\in}\mathcal{L}(\Pi)|\phi \Rightarrow \varphi\}\rangle_{\mathsf{state}^\sharp}\langle\cdot\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}\langle\cdot\rangle_{\mathsf{store}^\sharp}\langle\phi\rangle_{\mathsf{inv}}\rangle_{\top^\sharp}$$

$$Termination^\sharp \equiv \begin{cases} \langle\_\ \langle\langle\cdot\rangle_{\mathsf{k}^\sharp}\ \langle\cdot\rangle_{\mathsf{state}^\sharp}\ \langle P\rangle_{\mathsf{path}}\ \rangle_{\mathsf{trace}^\sharp}\ \_\rangle_{\top^\sharp} \rightharpoonup \langle P\rangle_{\mathsf{CE}} \\ \langle\langle\_\rangle_{\mathsf{store}^\sharp}\ \langle\phi\rangle_{\mathsf{inv}}\ \rangle_{\top^\sharp} \rightharpoonup \langle\cdot\rangle_{\mathsf{CE}} \end{cases}$$

Fig. 5. Initialization and termination for K abstract executions of *SIMP*

Fig. 5 provides the K structural rules for initialization and termination of the abstract executions, where pgm^{ϕ}_{Π} is a shorthand for the input cell containing the program $pgm = \texttt{vars}\ xs;\ s$, the abstraction predicates set Π, and the formula to verify ϕ. The initialization of an execution in collecting semantics for the program *pgm* has one $\mathsf{trace}^\sharp$ cell containing the abstract computation of the program, an initial abstract state corresponding to the best over-approximation of the property ϕ in the lattice $\mathcal{L}(\Pi)$, an empty path, an empty abstract store, and the property to be verified upon the program. (Note that we consider "$\cdot$" as the unit element for any cell.) The choice of the initial $\mathsf{state}^\sharp$ cell, $\sqcap\{\varphi \in \Pi |\ \phi \Rightarrow \varphi\}$, is the abstract representation of all concrete states σ_0, where ϕ is true (i.e. $\{\sigma_0 \mid \sigma_0 \vDash \phi\}$). As a matter of fact, we could as well generalize the initial abstract state cell $\mathsf{state}^\sharp$ to contain any element from the lattice $\mathcal{L}(\Pi)$. The termination of an execution in collecting semantics is expected to provide a path representing a potential counterexample to the validity of the property ϕ for *pgm*. In the case when there is no counterexample, the property is valid in the abstract model, and also in the program. Otherwise, no conclusion could be derived with respect to the validity of property ϕ for the given program.

The semantic rules for an execution with collecting semantics under predicate abstraction are described in Fig. 6. Note that in these rules the cells are considered to appear in the inner most environment wrapping them, according with the *locality principle* [19]. Next, we explain in details each of these rules.

The first two rules, (R1-2), deal with the case when the abstract computation reaches the final label of the program either with a valid abstract state Γ or with *false*. If the abstract state is valid then its containing $\mathsf{trace}^\sharp$ cell is voided (because there is no abstract computation left for it, and along the current abstract trace only valid states were encountered, meaning that the property ϕ is satisfied in any abstract state along this trace). If the abstract state is *false* then an error is found just before the end of the program. Whenever an error is found, meaning an abstract state where ϕ is not valid, we end the abstract execution from that

$(\mathrm{R1})_{\Box}$:
$\langle\, \langle\, \lfloor \ell \rfloor \,\rangle_{\mathsf{k}^\sharp}\ \ \langle\, \Gamma \,\rangle_{\mathsf{state}^\sharp}\ \ \langle\, \_ \,\rangle_{\mathsf{path}}\, \rangle_{\mathsf{trace}^\sharp}\ \ \rightarrow\ \cdot$

$(\mathrm{R2})_{\boxtimes}$:
$\langle\, \underline{\lfloor \ell \rfloor}_{\cdot} \,\rangle_{\mathsf{k}^\sharp}\ \ \langle\, \underline{\mathit{false}}_{\cdot} \,\rangle_{\mathsf{state}^\sharp}\ \ \langle\, \_\ \underline{\cdot}_{\ell} \,\rangle_{\mathsf{path}}$

$(\mathrm{R3})_{\Box\checkmark}$:
$\underline{\langle\, \langle\, \ell : \_\ \curvearrowright\ \_ \,\rangle_{\mathsf{k}^\sharp}\ \ \langle\, \Gamma \,\rangle_{\mathsf{state}^\sharp}\ \ \langle\, \_ \,\rangle_{\mathsf{path}}\, \rangle_{\mathsf{trace}^\sharp}\ \ \langle\, \_\ (\!|\ell, \Gamma|\!)\ \_ \,\rangle_{\mathsf{store}^\sharp}}_{\cdot}$

$(\mathrm{R4})_{\boxtimes}$:
$\langle\, \underline{\ell : \_ \curvearrowright \_}_{\cdot} \,\rangle_{\mathsf{k}^\sharp}\ \ \langle\, \underline{\mathit{false}}_{\cdot} \,\rangle_{\mathsf{state}^\sharp}\ \ \langle\, \_\ \underline{\cdot}_{\ell} \,\rangle_{\mathsf{path}}$

$(\mathrm{R5})_{\rightarrowtail}$:
$\langle\, \underline{\ell : \mathit{asg}(x, a)}_{\cdot} \,\rangle_{\mathsf{k}^\sharp}\ \ \langle\, \underline{\qquad\qquad \Gamma \qquad\qquad}_{\mathit{post}^\sharp_\phi(\Gamma, \mathit{asg}(x,a))} \,\rangle_{\mathsf{state}^\sharp}\ \ \langle\, \_\ \underline{\cdot}_{\ell} \,\rangle_{\mathsf{path}}\ \ \langle\, \underline{\qquad \Sigma \qquad}_{\Sigma[\ell \rightarrowtail \Gamma]} \,\rangle_{\mathsf{store}^\sharp}\ \ \langle \phi \rangle_{\mathsf{inv}}$

if $(\!|\ell, \Gamma|\!) \notin \Sigma$

$(\mathrm{R6})_{\rightarrowtail}$:
$$\frac{\langle\, \underline{\ell : \mathit{cnd}(b) \curvearrowright \mathit{if}(K_1,\ K_2)}_{\mathit{skip} \curvearrowright K_1} \curvearrowright K \,\rangle_{\mathsf{k}^\sharp}\ \ \langle\, \underline{\qquad\qquad \Gamma \qquad\qquad}_{\mathit{post}^\sharp_\phi(\Gamma, \mathit{cnd}(b))} \,\rangle_{\mathsf{state}^\sharp}\ \ \langle\, \underline{P}_{P, \ell} \,\rangle_{\mathsf{path}}}{\langle\, \langle \mathit{skip} \curvearrowright K_2 \curvearrowright K \rangle_{\mathsf{k}^\sharp}\ \ \langle \mathit{post}^\sharp_\phi(\Gamma, \mathit{cnd}(\neg b)) \rangle_{\mathsf{state}^\sharp}\ \ \langle P,\ \ell \rangle_{\mathsf{path}} \,\rangle_{\mathsf{trace}^\sharp}}\ \ \langle\, \underline{\qquad \Sigma \qquad}_{\Sigma[\ell \rightarrowtail \Gamma]} \,\rangle_{\mathsf{store}^\sharp}\ \ \langle \phi \rangle_{\mathsf{inv}}$$
if $(\!|\ell, \Gamma|\!) \notin \Sigma$

$(\mathrm{R7})_{\curvearrowright}$:
$\langle\, \langle\, \mathit{skip} \curvearrowright \_ \,\rangle_{\mathsf{k}^\sharp}\ \langle \mathit{false} \rangle_{\mathsf{state}^\sharp}\ \langle\, \_ \,\rangle_{\mathsf{path}}\, \rangle_{\mathsf{trace}^\sharp}\ \ \curvearrowright\ \cdot$

$(\mathrm{R8})_{\curvearrowright}$:
$\langle\, \underline{\mathit{skip}}_{\cdot} \,\rangle_{\mathsf{k}^\sharp}\ \ \langle\, \Gamma \,\rangle_{\mathsf{state}^\sharp}$

$(\mathrm{R9})_{\multimap}$:
$$\langle\, \underline{\qquad\qquad \mathit{while}(\ell : \mathit{cnd}(b),\ K) \qquad\qquad}_{\ell : \mathit{cnd}(b) \curvearrowright \mathit{if}(K \curvearrowright \mathit{while}(\ell : \mathit{cnd}(b), K),\ \cdot)} \,\rangle_{\mathsf{k}^\sharp}$$

Fig. 6. K rules for collecting semantics under predicate abstraction of *SIMP*

particular $\mathsf{trace}^\sharp$ cell and keep its representation in the path cell as a witness to the potential discovery of a bug (so called counterexample). This happens in the rules annotated with $\boxtimes$. (Note that $\square$ annotation stands for "good" termination.)

The rules (R3-4) present two other base cases, when the statement labeled ℓ is at the top of the abstract computation (i.e. $\langle \ell : \_ \_ \rangle_{\mathsf{k}^\sharp}$). The rule (R3) covers the case when a particular program point is reached again, with the same abstract state Γ. This is expressed by the fact that the abstract store, $\mathsf{store}^\sharp$ cell, contains the pair $(\!|\ell, \Gamma|\!)$. We can void the current $\mathsf{trace}^\sharp$ cell, because this particular abstract trace will not increment the $\mathsf{store}^\sharp$ cell any further. However, if a particular program point is reached with the *false* abstract state, as in (R4), we maintain the path as a counterexample.

Rule $(\mathrm{R5})_{\rightharpoonup}$ performs an abstract execution of an assignment statement encountered at the top of the abstract computation, $\mathsf{k}^\sharp$ cell. This means that the abstract state is updated by the abstract postcondition $post^\sharp$, while the current abstract state is used to update $\mathsf{store}^\sharp$, by adding the pair $(\!|\ell, \Gamma|\!)$ to it. Note that this addition is made only if the pair is not already in the abstract store, according to the definition of the $\mathsf{store}^\sharp$ update.

The rules $(\mathrm{R7\text{-}8})_\curlyvee$, containing *skip* at the top of the abstract computation, are both following a branching rule $(\mathrm{R6})_{\rightharpoonup}$. When the abstract execution encounters a branching condition, denoted by $\ell : cnd(b) \curvearrowright if(K_1, K_2)$, the rule $(\mathrm{R6})_{\rightharpoonup}$ spawns another abstract trace $newt^\sharp$. In this way, the current abstract trace maintains the "then" branch, with the boolean condition b, while $newt^\sharp$ maintains the "else" branch, with the boolean condition $\neg b$. However, it might be the case that not both branches are possible executions (e.g. if the boolean condition is *false*, then only the "else" branch is feasible). In order to filter these cases, when spawning the two traces, we also add a *skip* flag at the top of the abstract computation. The structural rules $(\mathrm{R7\text{-}8})_\curlyvee$ filter the *skip* flag: if the abstract state obtained by adding the conditional evaluates to *false*, then we remove this $\mathsf{trace}^\sharp$ cell, otherwise we continue the execution removing the *skip* flag.

The last rule, $(\mathrm{R9})_{\multimap}$ unfolds the while statement once. Note that the last three rules, (R7-9), are structural rules that transform the abstract computation. Also, we emphasize the R's annotations provide additional rules' classification.

Example 3. For the program in Fig. 2 and the property $\mathrm{AG}(\mathtt{err} = 0)$ the abstract execution with the predicate abstraction given by $\Pi = \{\mathtt{err} = 0\}$ terminates with $\langle 1, 2, 3 \rangle_{\mathsf{CE}}$, while if $\Pi = \{\mathtt{err} = 0, \mathtt{x} = 0\}$ the abstract execution terminates with $\langle 1, 2, 3, 4, 5, 6, 7, 8, 3 \rangle_{\mathsf{CE}}$. However, with the predicate abstraction given by $\Pi = \{\mathtt{err} = 0, \mathtt{x} = 0, \mathtt{x} = 1\}$ the abstract execution ends with $\langle \cdot \rangle_{\mathsf{CE}}$.
The abstract execution with $\Pi = \{\mathtt{err} = 0, \mathtt{x} = 0\}$ starts with the abstract computation described in Example 2, and proceeds as described in Fig. 7.

4 Correspondence between Concrete and Collecting Semantics

In this part we focus on proving the correctness of the K definition of *SIMP* collecting semantics under predicate abstraction with respect to the K definition

$$\langle\langle\langle 1 : asg(\mathtt{x},0)\ \_\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)\bot_{(\mathtt{x}=0)}\rangle_{\mathsf{state}^\sharp}\ \langle\cdot\rangle_{\mathsf{path}}\ \rangle_{\mathsf{trace}^\sharp}\ \langle\cdot\rangle_{\mathsf{store}^\sharp}\ \langle\mathtt{err}=0\rangle_{\mathsf{inv}}\ \rangle_{\top^\sharp}$$

$$\xrightarrow{R5}\ \langle\langle\langle 2 : asg(\mathtt{err},\mathtt{x})\ \_\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1\rangle_{\mathsf{path}}\ \rangle_{\mathsf{trace}^\sharp}$$
$$\langle(\!|1,(\mathtt{err}=0)\bot_{(\mathtt{x}=0)}|\!)\rangle_{\mathsf{store}^\sharp}\ \langle\mathtt{err}=0\rangle_{\mathsf{inv}}\ \rangle_{\top^\sharp}$$

$$\xrightarrow{R5}\ \langle\langle\langle while(3 : cnd(\mathtt{y}\le 0),K)\ \_\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1,2\rangle_{\mathsf{path}}\ \rangle_{\mathsf{trace}^\sharp}$$
$$\langle(\!|1,(\mathtt{err}=0)\bot_{(\mathtt{x}=0)}|\!),(\!|2,(\mathtt{err}=0)(\mathtt{x}=0)|\!)\rangle_{\mathsf{store}^\sharp}\ \_\rangle_{\top^\sharp}$$

where K is $4 : asg(\mathtt{x},\mathtt{x}+1) \curvearrowright 5 : asg(\mathtt{y},\mathtt{y}+\mathtt{x}) \curvearrowright 6 : asg(\mathtt{x},-1+\mathtt{x})$
$\curvearrowright 7 : cnd(\neg(\mathtt{x}=0)) \curvearrowright if(8 : asg(\mathtt{err},1),\cdot)$

$$\overset{R9}{\rightharpoonup}\ \langle\langle\langle 3 : cnd(\mathtt{y}\le 0)\curvearrowright if(K\curvearrowright while(3 : cnd(\mathtt{y}\le 0),K),\cdot)\ \_\rangle_{\mathsf{k}^\sharp}\ \_\rangle_{\mathsf{trace}^\sharp}$$
$$\langle(\!|1,(\mathtt{err}=0)\bot_{(\mathtt{x}=0)}|\!),(\!|2,(\mathtt{err}=0)(\mathtt{x}=0)|\!)\rangle_{\mathsf{store}^\sharp}\ \_\rangle_{\top^\sharp}$$

$$\xrightarrow{R6}\ \langle\langle\langle skip\curvearrowright\ K\ \_\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1,2,3\rangle_{\mathsf{path}}\ \rangle_{\mathsf{trace}^\sharp}$$
$$\langle\langle skip\curvearrowright\cdot\curvearrowright\lfloor 9\rfloor\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1,2,3\rangle_{\mathsf{path}}\ \rangle_{\mathsf{trace}^\sharp}$$
$$\langle(\!|1,(\mathtt{err}=0)\bot_{(\mathtt{x}=0)}|\!),(\!|2,(\mathtt{err}=0)(\mathtt{x}=0)|\!),(\!|3,(\mathtt{err}=0)(\mathtt{x}=0)|\!)\rangle_{\mathsf{store}^\sharp}\ \_\rangle_{\top^\sharp}$$

$$\overset{R8}{\rightharpoonup}\ \langle\langle\langle K\ \_\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1,2,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle\langle skip\curvearrowright\lfloor 9\rfloor\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1,2,3\rangle_{\mathsf{path}}\ \rangle_{\mathsf{trace}^\sharp}\ \ \_\rangle_{\top^\sharp}$$

$$\overset{R8}{\rightharpoonup}\ \langle\langle\langle 4 : asg(\mathtt{err},\mathtt{x})\ \_\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1,2,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle\langle\lfloor 9\rfloor\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1,2,3\rangle_{\mathsf{path}}\ \rangle_{\mathsf{trace}^\sharp}\ \ \_\rangle_{\top^\sharp}$$

$$\xrightarrow{R1}\ \langle\langle\langle 4 : asg(\mathtt{x},\mathtt{x}+1)\ \_\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1,2,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle(\!|1,(\mathtt{err}=0)\bot_{(\mathtt{x}=0)}|\!),(\!|2,(\mathtt{err}=0)(\mathtt{x}=0)|\!),(\!|3,(\mathtt{err}=0)(\mathtt{x}=0)|\!)\rangle_{\mathsf{store}^\sharp}\ \_\rangle_{\top^\sharp}$$

$$\xrightarrow{R5}\ \langle\langle\langle 5 : asg(\mathtt{y},\mathtt{y}+\mathtt{x})\ \_\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)\ \neg(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1,2,3,4\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle\_(\!|4,(\mathtt{err}=0)(\mathtt{x}=0)|\!)\rangle_{\mathsf{store}^\sharp}\ \_\rangle_{\top^\sharp}$$

$$\xrightarrow{R5}\ \langle\langle\langle 6 : asg(\mathtt{x},-\mathtt{1}+\mathtt{x})\ \_\rangle_{\mathsf{k}^\sharp}\ \langle(\mathtt{err}=0)\ \neg(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\ \langle 1,2,3,4,5\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle\_(\!|5,(\mathtt{err}=0)\ \neg(\mathtt{x}=0)|\!)\rangle_{\mathsf{store}^\sharp}\ \_\rangle_{\top^\sharp}$$

$$\xrightarrow{R5}\ \langle\langle\langle 7 : cnd(\neg(\mathtt{x}=0))\curvearrowright if(8 : asg(\mathtt{err},1),\cdot)\ \_\rangle_{\mathsf{k}^\sharp}\langle(\mathtt{err}=0)\bot_{(\mathtt{x}=0)}\rangle_{\mathsf{state}^\sharp}\langle\_\,6\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle\_(\!|6,(\mathtt{err}=0)\ \neg(\mathtt{x}=0)|\!)\rangle_{\mathsf{store}^\sharp}\ \_\rangle_{\top^\sharp}$$

$$\xrightarrow{R6}\ \langle\langle\langle skip\curvearrowright 8 : asg(\mathtt{err},1)\ \_\rangle_{\mathsf{k}^\sharp}\langle(\mathtt{err}=0)\ \neg(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\langle\_\,7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle\langle skip\curvearrowright\cdot\curvearrowright while(3 : cnd(\mathtt{y}\le 0),K)\ \_\rangle_{\mathsf{k}^\sharp}\langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\langle\_\,7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle\_(\!|7,(\mathtt{err}=0)\ \bot_{(\mathtt{x}=0)}|\!)\rangle_{\mathsf{store}^\sharp}\ \_\rangle_{\top^\sharp}$$

$$\overset{R8}{\rightharpoonup}\ \langle\langle skip\curvearrowright 8 : asg(\mathtt{err},1)\ \_\rangle_{\mathsf{k}^\sharp}\langle(\mathtt{err}=0)\ \neg(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\langle\_\,7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle\langle while(3 : cnd(\mathtt{y}\le 0),K)\ \_\rangle_{\mathsf{k}^\sharp}\langle(\mathtt{err}=0)(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\langle\_\,7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle\_(\!|3,(\mathtt{err}=0)(\mathtt{x}=0)|\!)\ \_\rangle_{\mathsf{store}^\sharp}\ \ \_\rangle_{\top^\sharp}$$

$$\xrightarrow{R3}\ \langle\langle skip\curvearrowright 8 : asg(\mathtt{err},1)\ \_\rangle_{\mathsf{k}^\sharp}\langle(\mathtt{err}=0)\ \neg(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\langle\_\,7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}\ \ \_\rangle_{\top^\sharp}$$

$$\overset{R8}{\rightharpoonup}\ \langle\langle 8 : asg(\mathtt{err},1)\ \_\rangle_{\mathsf{k}^\sharp}\langle(\mathtt{err}=0)\ \neg(\mathtt{x}=0)\rangle_{\mathsf{state}^\sharp}\langle\_\,7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}\ \ \_\rangle_{\top^\sharp}$$

$$\xrightarrow{R5}\ \langle\langle while(3 : cnd(\mathtt{y}\le 0),K)\ \_\rangle_{\mathsf{k}^\sharp}\langle false\rangle_{\mathsf{state}^\sharp}\langle\_\,8\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}$$
$$\langle\_(\!|7,(\mathtt{err}=0)\ \neg(\mathtt{x}=0)|\!)\rangle_{\mathsf{store}^\sharp}\ \_\rangle_{\top^\sharp}$$

$$\overset{R9}{\rightharpoonup}\ \langle\langle\langle 3 : cnd(\mathtt{y}\le 0)\curvearrowright if(\dots)\ \_\rangle_{\mathsf{k}^\sharp}\langle false\rangle_{\mathsf{state}^\sharp}\langle 1,2,3,4,5,6,7,8\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}\ \_\rangle_{\top^\sharp}$$

$$\xrightarrow{R4}\ \langle\langle\cdot\rangle_{\mathsf{k}^\sharp}\langle\cdot\rangle_{\mathsf{state}^\sharp}\langle 1,2,3,4,5,6,7,8,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}^\sharp}\ \_\rangle_{\top^\sharp}\ \rightharpoonup\ \langle 1,2,3,4,5,6,7,8,3\rangle_{\mathsf{CE}}$$

Fig. 7. Example of a K abstract execution

of the concrete semantics. In other words, we investigate if our K description of model checking with predicate abstraction can be soundly used to prove certain properties for *SIMP* programs.

We revise first some basics of predicate abstraction, namely the Galois connection. In the context of predicate abstraction, the Galois connection is defined as $\mathcal{P}(\mathcal{S}) \overset{\alpha}{\rightleftarrows}_{\gamma} \mathcal{L}(\Pi)$ where $\mathcal{S} = \{\sigma : V \mapsto \mathbf{Z}\}$ is the set of all states for a *SIMP* program. The abstraction-concretization pair $\langle \alpha, \gamma \rangle$ is defined as follows:

$$\alpha(S) := \sqcap\{\varphi \mid (\forall \sigma \in S)\ \sigma \vDash \varphi\}, \text{ for any subset of states } S \subseteq \mathcal{S}$$
$$\gamma(\varphi) := \{\sigma \in \mathcal{S} \mid \sigma \vDash \varphi\}, \text{ for any formula } \varphi \in \mathcal{L}(\Pi)$$

It is easy to verify that $\langle \alpha, \gamma \rangle$ forms a Galois connection. Also, it is standard that $post^\sharp$ is a sound approximation of the strongest postcondition (i.e. if $\sigma \vDash \varphi$ and $\sigma' \in post(\sigma)$ then $\sigma' \vDash post^\sharp(\varphi)$). More on these can be found in [6].

In what follows we prove a similar property about K executions of programs in concrete semantics and collecting semantics under predicate abstraction, respectively. In other words, we check that any concrete execution of a program can be retrieved from the meta-execution, and we state what conditions need to be satisfied such that we can derive from the meta-execution the validity of the property of interest for the given *SIMP* program. We assume as given the *SIMP* program $pgm = \texttt{vars}\ xs; s$, the finite set of predicates Π, and the invariant $\mathrm{AG}\phi$.

Theorem 1. *Any* K *execution in collecting semantics under predicate abstraction is finite.*

This theorem essentially ensures the termination of the program verification method described in the previous section.

Lemma 1. *For any* $\langle \_ \langle \Sigma_1 \rangle_{\mathsf{store}^\sharp}\ \_ \rangle_{\top^\sharp} \overset{*}{\rightarrow} \langle \_ \langle \Sigma_2 \rangle_{\mathsf{store}^\sharp}\ \_ \rangle_{\top^\sharp}$*, a fragment of execution in collecting semantics, we have* $\Sigma_1 \subseteq \Sigma_2$.

This is easy to see from the fact that any rule (R1-9) produces transitions that preserve the ascending inclusion of the $\mathsf{store}^\sharp$ terms.

Lemma 2. *If the* K *execution in collecting semantics encounters a transition that does not change the* $\mathsf{store}^\sharp$ *cell, as* $\langle \_ \langle \Sigma_1 \rangle_{\mathsf{store}^\sharp}\ \_ \rangle_{\top^\sharp} \overset{Ri}{\rightarrow} \langle \_ \langle \Sigma_1 \rangle_{\mathsf{store}^\sharp}\ \_ \rangle_{\top^\sharp}$ *where* $i = 7, 8, 9$*, then the execution evolves either into a terminal configuration, with the rules* $(R1-4)$*, or into a configuration* $\langle \_ \langle \Sigma_2 \rangle_{\mathsf{store}^\sharp}\ \_ \rangle_{\top^\sharp}$ *where* $\Sigma_1 \subset \Sigma_2$*, with the rules* $(R5-6)$.

This lemma ensures that any structural rule enables a computational rule, and, consequently, there is no execution in collecting semantics with a suffix that does not increment the content of the $\mathsf{store}^\sharp$ cell.

Proof of Theorem 1 (sketch). The proof follows from Lemma 1 and Lemma 2, coupled with the fact that there is an upper bound for any $\mathsf{store}^\sharp$ term (because any *SIMP* program has a finite number of labels, and Π has a finite number of predicates, hence $\mathcal{L}(\Pi)$ has a finite number of elements). □

Theorem 2. *If the concrete execution initialized with a* $\langle \sigma_0 \rangle_{\mathsf{state}}$ *evolves into a concrete configuration with* $\langle \sigma \rangle_{\mathsf{state}}$*, namely* $\langle \langle s \rangle_{\mathsf{k}} \langle \sigma_0 \rangle_{\mathsf{state}} \rangle_{\top} \overset{*}{\rightarrow} \langle \_ \langle \sigma \rangle_{\mathsf{state}} \rangle_{\top}$*, and if* $\sigma_0 \vDash \Gamma_0$ *(i.e.* σ_0 *is contained in the subset of abstract states denoted by* Γ_0*), then the abstract execution starting with the abstract state* $\langle \Gamma_0 \rangle_{\mathsf{state}^\sharp}$ *evolves into an abstract configuration with* $\langle \Gamma \rangle_{\mathsf{state}^\sharp}$*, namely* $\langle \_ \langle k^\sharp(s) \rangle_{\mathsf{k}^\sharp} \langle \Gamma_0 \rangle_{\mathsf{state}^\sharp} \_ \rangle_{\top^\sharp} \overset{*}{\rightarrow} \langle \_ \langle \Gamma \rangle_{\mathsf{state}^\sharp} \_ \rangle_{\top^\sharp}$*, such that* $\langle \_ \langle \sigma \rangle_{\mathsf{state}} \_ \rangle_{\top} \vDash \langle \_ \langle \Gamma \rangle_{\mathsf{state}^\sharp} \_ \rangle_{\top^\sharp}$ *holds true.*

This theorem states that any K execution in the concrete semantics is sinked into a K execution in the collecting semantics under predicate abstraction (in case we take Γ_0 to be *true*, then $\sigma_0 \vDash \Gamma_0$ for any initial concrete state σ_0).

Remark 1. By $\langle \_ \langle \sigma \rangle_{\mathsf{state}} \_ \rangle_{\top} \vDash \langle \_ \langle \Gamma \rangle_{\mathsf{state}^\sharp} \_ \rangle_{\top^\sharp}$ we understand that $\sigma \vDash \Gamma$, and that the abstract computation $\langle k^\sharp \rangle_{\mathsf{k}^\sharp}$ from the $\mathsf{trace}^\sharp$ cell containing $\langle \Gamma \rangle_{\mathsf{state}^\sharp}$ is the abstract computation of the program fragment obtained from the cell $\langle k \rangle_{\mathsf{k}}$ in $\langle \_ \langle \sigma \rangle_{\mathsf{state}} \_ \rangle_{\top}$.

Lemma 3. *For any* $\langle \_ \langle \sigma \rangle_{\mathsf{state}} \_ \rangle_{\top} \rightarrow \langle \_ \langle \sigma' \rangle_{\mathsf{state}} \_ \rangle_{\top}$ *a concrete transition in a concrete execution, if there is an abstract configuration* $\langle \_ \langle \Gamma \rangle_{\mathsf{state}^\sharp} \_ \rangle_{\top^\sharp}$ *such that* $\langle \_ \langle \sigma \rangle_{\mathsf{state}} \_ \rangle_{\top} \vDash \langle \_ \langle \Gamma \rangle_{\mathsf{state}^\sharp} \_ \rangle_{\top^\sharp}$*, then there is* $\langle \_ \langle \Gamma' \rangle_{\mathsf{state}^\sharp} \_ \rangle_{\top^\sharp}$ *an abstract configuration satisfying the following two properties:*
(1) $\langle \_ \langle \Gamma \rangle_{\mathsf{state}^\sharp} \_ \rangle_{\top^\sharp} \overset{*}{\rightarrow} \langle \_ \langle \Gamma' \rangle_{\mathsf{state}^\sharp} \_ \rangle_{\top^\sharp}$ *and*
(2) $\langle \_ \langle \sigma' \rangle_{\mathsf{state}} \_ \rangle_{\top} \vDash \langle \_ \langle \Gamma' \rangle_{\mathsf{state}^\sharp} \_ \rangle_{\top^\sharp}$.

Proof (Lemma 3). The proof goes by case analysis over the rules in the concrete semantics. For example, let us consider that the concrete transition from the hypothesis, $\langle \_ \langle \sigma \rangle_{\mathsf{state}} \_ \rangle_{\top} \rightarrow \langle \_ \langle \sigma' \rangle_{\mathsf{state}} \_ \rangle_{\top}$, is the result of the application of the assignment rule $x := a$. Then, in the abstract semantics, the transition is made via application of the rule (R5), and in the next configuration the $\mathsf{state}^\sharp$ cell contains $\Gamma' = post^\sharp_\phi(\Gamma, asg(x, a))$. However, from the definition of $post^\sharp_\phi$ we see that $\Gamma[x'/x] \wedge (x = a[x'/x]) \Rightarrow \Gamma'$. But, from hypothesis we have $\sigma \vDash \Gamma$, so $\sigma' \vDash \Gamma \wedge (x = a[x'/x])$, hence $\sigma' \vDash \Gamma'$. □

Proof of Theorem 2 (sketch). The proof uses induction on the length of the concrete derivation and Lemma 3. □

Theorem 3. *For any pgm,* Π*, and* $AG\phi$*, if we have the abstract execution*

$$pgm^\phi_\Pi \rightharpoonup \langle \langle \langle k^\sharp(s) \rangle_{\mathsf{k}^\sharp} \ \langle \sqcap \{\varphi \mid \phi \Rightarrow \varphi\} \rangle_{\mathsf{state}^\sharp} \ \langle \cdot \rangle_{\mathsf{path}} \rangle_{\mathsf{trace}^\sharp} \ \langle \cdot \rangle_{\mathsf{store}^\sharp} \ \langle \phi \rangle_{\mathsf{inv}} \rangle_{\top^\sharp} \overset{*}{\rightarrow}$$
$$\langle \langle \_ \rangle_{\mathsf{store}^\sharp} \ \langle \phi \rangle_{\mathsf{inv}} \rangle_{\top^\sharp} \rightharpoonup \langle \cdot \rangle_{\mathsf{CE}}$$

then, for all $\langle \sigma_0 \rangle_{\mathsf{state}}$ *and* $\langle \sigma \rangle_{\mathsf{state}}$ *concrete states from a concrete execution* $pgm \rightharpoonup \langle \langle s \rangle_{\mathsf{k}} \langle \sigma_0 \rangle_{\mathsf{state}} \rangle_{\top} \overset{*}{\rightarrow} \langle \_ \langle \sigma \rangle_{\mathsf{state}} \rangle_{\top}$*, if* $\sigma_0 \vDash \phi$*, then* $\sigma \vDash \phi$ *holds true.*

This theorem says that if the K execution in collecting semantics under predicate abstraction terminates without finding any counterexample, then the property ϕ is an invariant for any concrete execution of the program *pgm*.

Proof of Theorem 3 (sketch). We observe that since all $\mathsf{trace}^\sharp$ terms disappear in the final state of the abstract execution, then it means that rules $(\mathrm{R2,4})_{\boxtimes}$ were never executed. We can apply Theorem 2 (we know that $\sigma_0 \vDash \sqcap\{\varphi \mid \phi \Rightarrow \varphi\}$ because $\sigma_0 \vDash \phi$ and $\phi \Rightarrow \sqcap\{\varphi \mid \phi \Rightarrow \varphi\}$). Hence, there exists a valid abstract state Γ such that $\sigma \vDash \Gamma$. Moreover, because any intermediate $\langle \Gamma \rangle_{\mathsf{state}^\sharp}$ is a $post^\sharp_\phi$ result, it means that $\Gamma \Rightarrow \phi$. From these two, namely $\sigma \vDash \Gamma$ and $\Gamma \Rightarrow \phi$, we conclude that $\sigma \vDash \phi$. □

It is notorious that model checking with abstraction is not a complete procedure. Essentially, this comes from the false negative answers the model checking with abstraction can issue. Nevertheless, the incompleteness of abstract model checking gives rise to a bundle of work known as "abstraction refinement".

5 Conclusions and Future Work

In this paper we study the embedding of predicate abstraction model checking into the K framework. This work makes two contributions: first, it shows that model checking with predicate abstraction can be incorporated as a formal analysis approach following the very same definitional style used for concrete semantics of programming languages in K; second, it shows how to relate the concrete semantics and the predicate abstracted semantics (i.e. collecting semantics), and proves that the latter is correct for the original language.

In near future we plan to give a K definition for symbolic executions of programs, such that we could embed also predicate abstraction CEGAR into the K framework. Another line to pursue is enriching the class of properties we want to verify. However, it is well known that under predicate abstraction we are forced to limit the properties of interest to safety properties. To overcome this, we should investigate the transition predicate abstraction which enables verification of liveness properties. Ultimately, these steps would lead to an automated and founded system in K for defining program reasoning techniques.

Acknowledgments. We would like to give our special thanks to Professor Dorel Lucanu for his insights in organizing and standardizing this work. Our deep gratitude goes to Professor Grigore Roşu for providing us with the latest K documentation and observations to improve our work. Also, we thank to the anonymous reviewers for useful and much appreciated comments and corrections.

References

1. Alba-Castro, M., Alpuente, M., Escobar, S.: Automatic certification of Java source code in rewriting logic. In: Leue, S., Merino, P. (eds.) FMICS. Lecture Notes in Computer Science, vol. 4916, pp. 200–217. Springer (2007)
2. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
3. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principles of Programming Languages. pp. 238–252. ACM Press (1977)

4. Das, S., Dill, D.L.: Counter-example based predicate discovery in predicate abstraction. In: Formal Methods in Computer-Aided Design. LNCS, vol. 2517, pp. 19–32. Springer-Verlag (2002)
5. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Gaducci, F., Montanari, U. (eds.) Workshop on Rewriting Logic and Its Applications. Electronic Notes in Theoretical Computer Science, vol. 71. Elsevier (September 2002)
6. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Proceedings of the 9th Conference on Computer-Aided Verification. pp. 72–83. Springer-Verlag (1997)
7. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. SIGPLAN Notices 37(1), 58–70 (2002)
8. Hills, M., Roşu, G.: KOOL: An application of rewriting logic to language prototyping and analysis. In: Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA'07). LNCS, vol. 4533, pp. 246–256. Springer (2007)
9. Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. Electronic Notes in Theoretical Computer Science 4 (1996)
10. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer-Aided Verification. pp. 123–136 (2006)
11. Meredith, P., Hills, M., Roşu, G.: An executable rewriting logic semantics of K-scheme. In: Dube, D. (ed.) Proceedings of the 2007 Workshop on Scheme and Functional Programming (SCHEME'07), Technical Report DIUL-RT-0701. pp. 91–103. Laval University (2007)
12. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
13. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstraction. In: Automated Deduction CADE-19. Lecture Notes in Computer Science, vol. 2741, pp. 2–16. Springer (2003)
14. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theoretical Computer Science 373(3), 213–237 (2007)
15. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc. (1999)
16. Palomino, M.: A predicate abstraction tool for Maude. Documentation and tool available at `http://maude.sip.ucm.es/~miguelpt/bibliography.html`
17. Roşu, G.: K: A rewriting-based framework for computations – preliminary version. Tech. Rep. Department of Computer Science UIUCDCS-R-2007-2926 and College of Engineering UILU-ENG-2007-1827, University of Illinois at Urbana-Champaign (2007)
18. Roşu, G., Schulte, W.: Matching logic — extended report. Tech. Rep. Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign (January 2009)
19. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming (Submitted)
20. Şerbănuţă, T.F., Roşu, G.: K-Maude: A rewriting based tool for semantics of programming languages. In: Proceedings of the 8th International Workshop on Rewriting Logic and its Applications (WRLA'10). Lecture Notes in Computer Science (2010), this volume
21. Şerbănuţă, T.F., Roşu, G., Meseguer, J.: A rewriting logic approach to operational semantics. Information and Computation 207, 305–340 (2009)