Path Directed Symbolic Execution in the K Framework

Irina Măriuca Asăvoae, Mihail Asăvoae, Dorel Lucanu
Faculty of Computer Science, Alexandru Ioan Cuza University, Iaşi, Romania
{mariuca.asavoae, mihail.asavoae, dlucanu}@info.uaic.ro

Abstract—**The K framework is a rewrite-based executable semantic framework built with the purpose to define programming languages and formal analysis methods. This paper introduces K definition of the path-directed symbolic execution, which is that part of Counterexample Guided Abstraction Refinement (CEGAR) where the counterexample is checked for spuriousness. To express this technique in K, we use strongest postcondition computation on straight line code. The programming language at hand is imperative, with simple arithmetic, but the approach can be applied to more complicated languages. This work aims to further advance the integration of CEGAR technique in rewriting logic semantics project in general, and in K in particular. By doing this we obtain an uniform description of the definition of the programming language, the abstract model checking, and the counterexample guided refinement. This uniformity enables formal reasoning about CEGAR's implementation correctness which could be further standardized and eventually automatized.**

I. Introduction

Model checking [3] approach to software verification is effective when it relies on abstraction to control the state space explosion problem. Finding the right abstraction typically means to choose a convenient, rich enough, abstract domain. This is a difficult task considering the complexity of software systems. In many cases, the initial abstract domain needs additional information to tune the abstraction to verify the property of interest. Therefore, the right abstraction is discovered after an iterative process of trying various abstract domains. Such a successful technique is counterexample guided abstraction refinement (CEGAR) [2], [6]. When a particular abstract domain fails to prove the property, the reason for failure, i.e. the counterexample, is analyzed and the outcome is included in the refined abstract domain. This procedure iterates until, hopefully, the abstract domain has enough information to prove or disprove the property.

CEGAR commonly uses a technique called predicate abstraction [7], that spawns from the general framework of abstract interpretation [5]. In predicate abstraction, the abstract model of the program consists of sets of predicates over program variables. The property of interest that is successfully checked in the abstract domain, holds also in the concrete program. In case the property does not hold in the abstract domain, the method generates an abstract counterexample. Usually, the counterexample is symbolically executed to reveal the reason why the property fails to hold.

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

Symbolic execution, introduced in [11], is a static analysis technique that manipulates symbolic input values instead of concrete inputs. It applies on one or more program paths and propagates, forward or backward, these symbolic values. The state of the system under symbolic execution consists of the program counter and the expression of program variables in terms of symbolic values. A path-directed symbolic execution is also called symbolic simulation [2].

The K framework [16], [18] emerged from the rewriting logic semantics project [14], [12], and aims to support the design and analysis of programming languages. K consists of a highly concurrent rewrite abstract machine, a definitional technique, and a specific notation. A definition in K of a programming language or a programming reasoning method consists of a multiset of cells called configuration together with equations and rewrite rules over the contents of these cells. The equations define the abstraction degree and the rewrite rules define the observability degree of the specification. The K definition is modular, semantics-based and executable.

This paper introduces, in a K definitional style, a path-directed forward symbolic execution. It is a continuation of our previous work [1] on using K to define model checking with predicate abstraction. The symbolic execution comes at hand when checking an abstract counterexample for spuriousness. The symbolic execution is path-directed because we consider one counterexample at a time Our symbolic execution relies on strongest postcondition computation on straight line code to propagate the symbolic input values. The resulting postcondition formulas are handled by an external SMT solver. We prove the termination of our symbolic simulation of an abstract counterexample, as well as a correspondence result between the symbolic execution and the concrete execution. Our current work is meant to be part of a larger project which aims to use the K framework to define program reasoning techniques, and, even further, to set the premises for proving the correctness of such program reasoning techniques. This line of research, which targets to verify the verifiers, is currently forwarded by the project "Kernel of Truth" [20].

Related work. There is a large body of work on using abstraction to make model checking scalable for software verification [2], [6], [13]. Finding the right abstraction is difficult and abstraction-based model checking techniques come often with an automated procedure, called refinement. This helps to enrich the abstraction domain, and to increase the chances of proving the property. Usually, during the refinement step, the abstract counterexample is checked for spuriousness using a form of symbolic execution. To tackle this problem, CEGAR uses a symbolic algorithm that simulates the counterexample on the actual model. Lazy abstraction [9] is CEGAR based, localized and on-demand abstraction refinement procedure that uses backward symbolic simulation to detect if an abstract counterexample is spurious. A further improved technique [8], that relies on Craig interpolation to refine the abstract domain, uses forward symbolic execution to check if counterexample is feasible or not. A symbolic execution for systematic design of test cases can be found in [10].

Rewriting logic uses Maude system [4] for specification and verification of programs. Model checking of rewriting logic theories can be done in two ways: using the `search` command of Maude or with the LTL Maude model checker. The `search` command finds reachable states that satisfy some conditions, from a given state. The LTL Maude model checker verifies if a system of interest satisfies an LTL property. Both approaches generate counterexamples. A more advanced technique to get counterexamples is to combine the above Maude techniques with the induction-guided falsification [15] on the structure of the state space.

The K framework [16], [18] specializes rewriting logic for both design of programming languages and program reasoning tools. The K definitions for program reasoning include type systems [18], explicit state model checker [16], and Hoare style program verifier [17]. This verifier is based on matching logic, an extension of Hoare logic, with a rewriting based forward symbolic execution. The current prototype implementation for the K framework is K-Maude, introduced in [19].

Next, we elaborate on how our current work relates with respect to previously mentioned contributions. Among K-based specifications of program reasoning tools, this paper introduces a path-directed symbolic execution as a mechanism to check counterexample spuriousness. It uses strongest postcondition on straight line code to propagate symbolic values. By comparison, the symbolic execution from matching logic [17] explores all program paths (loops are handled with invariants) and uses matching conditions to propagate the information. With respect to rewriting logic, the mentioned model checking approaches provide concrete counterexamples (actual bugs), so there is no need for a mechanism to check their spuriousness. In [21] we find a term-rewriting based program analysis which uses symbolic execution with a representation of states which addresses the path explosion problem. By comparison, our work is tailored to straight line code symbolic execution with the purpose of checking counterexample spuriousness, and does not explore all program paths. As a side note, the matching logic symbolic execution may also be adapted to work in the counterexample abstraction refinement framework. Concluding, our approach brings CEGAR style symbolic simulation in the rewrite logic, by means of the K framework.

Outline of the paper. The structure of the paper is as follows: Section II introduces *the K framework* by defining *SIMP*, a simple imperative programming language. Section III defines a *path-directed forward symbolic execution* for *SIMP*, which is used to check abstract counterexample for spuriousness. Section IV states the termination and correctness results of the path-directed symbolic execution and establishes a correspondence between this and the concrete execution. Finally, in Section V we draw conclusions and present directions for further work.

II. 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 [18].

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*. The strictness attribute that corresponds to the inequality $Aexp {<=} Aexp$ is translated into the set of heating/cooling rule pairs: $a_1 \leq a_2 \rightleftharpoons a_1 \curvearrowright \square \leq a_2$ and $i_1 \leq a_2 \rightleftharpoons a_2 \curvearrowright i_1 \leq \square$. These structural rules state how an inequality 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 compared using the internal operation of integer inequality $\leq_{Int}$ as represented by the rule $i_1 \leq i_2 \to i_1 \leq_{Int} i_2$ in the *SIMP* semantics (right column). This particular choice of evaluation for boolean expressions plays an important role in enabling efficient partial evaluation (reflected in the rules for `and`). The "if" statement has the attribute $seqstrict(1)$ which means that the strictness attribute *seqstrict* is applied only to the first argument of the `if` statement, namely the condition. When the attribute *strict* annotates an n-ari syntactic construct (for example the assignment), it means that we allow arbitrary application of the heating/cooling rule pairs to the assignment arguments.

The K modeling of a program configuration, named K-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, formal analysis results, etc. The *SIMP* program configuration is:

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

where the top cell $\langle \ldots \rangle_\top$ contains two other cells: the computation $\langle K \rangle_{\mathsf{k}}$ and the store $\langle \mathsf{Map}[\,Var \mapsto Int\,] \rangle_{\mathsf{state}}$. 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

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

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

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 $\mathsf{Map}[Var \mapsto Int]$, 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 $left \rightarrow right$ is used.

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 with $\sigma[x]$ 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 "$\_\rangle$" in cell k represents the remaining computation which is not being used in this rule (analogously for the notation "$\langle\_$" in other rules).

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 "$\rightarrow$" for computational rules, and "$\rightharpoonup$" for structural rules. For example, 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 on the initial configuration $Init \equiv \langle \texttt{vars}\ xs; s\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{state}}$ (last rule in the right column in Fig. 1) leaves the computation cell containing the entire set of statements, 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 k cell is empty.

```
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.

A *SIMP* program *pgmX* is given in Fig. 2 as $\texttt{vars x, y, err};\ sX$, 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.

III. Symbolic Execution of a Given Path

Symbolic execution is a method that combines explicit and symbolic techniques for program reasoning. It is used to find bugs or to construct program proofs. In symbolic execution, symbolic values are assigned to program inputs and propagated along the program paths to check if error states are reachable. The propagation can be forward or backward, on all program paths. A path-directed symbolic execution propagates the symbolic values along one path, as when checking an abstract counterexample for spuriousness. By property preserving symbolic execution we mean that the property of interest is an invariant for the symbolic execution.

In this section we elaborate on how to define, in the K framework, a path directed symbolic execution with path-invariant. Essentially, we are interested if the particular path is feasible under a given invariant. This means that the symbolic execution starts with a precondition which implies the invariant, and the invariant holds true in any consequent valid postcondition along the given path. Note that the K-definition of the path directed symbolic execution presented in the followings gives also accurate details of its implementation in K-Maude. As such, we exploit and emphasize the fact that K is an executable semantic framework.

The configuration for the considered symbolic execution $\mathit{Configuration}^\flat$ is defined in Fig. 3. The computation is maintained in the cell $\mathsf{k}^\flat$ as an isomorphic abstraction of the initial program. The cell $\mathsf{state}^\flat$ maintains precondition/postcondition formulas, while cells path and $\mathsf{path}^\flat$ maintain the lists of the program points to be executed and program points already executed, respectively. Finally, cell ci contains the current incarnation of the program variables while cell inv contains the invariant formula.

In the followings we provide detailed description on the structure of each cell of $\mathit{Configuration}^\flat$.

The $\mathsf{k}^\flat$ 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 $K^\flat$ as follows:

$$\langle K^\flat \rangle_{\mathsf{k}^\flat} \begin{cases} PC ::= \text{positive integers as program counters} \\ \mathit{Var} ::= \text{symbols denoting program variables} \\ \mathit{Asg} ::= \mathit{asg}(\mathit{Var}, \mathit{AExp}) \\ \mathit{Cnd} ::= \mathit{cnd}(\mathit{BExp}) \\ \mathit{TransAsg} ::= PC : \mathit{Asg} \\ \mathit{TransCnd} ::= PC : \mathit{Cnd} \\ \mathit{TranSkip} ::= PC : \mathit{skip} \\ \mathit{Trans} ::= \mathit{TransAsg} \mid \mathit{TransCnd} \mid \mathit{TranSkip} \\ \mathit{Kcfg} ::= \mathit{Trans} \mid \mathsf{List}_{\curvearrowright}[\mathit{Kcfg}] \\ \qquad \mid \mathit{if}(\mathit{Kcfg}, \mathit{Kcfg}) \mid \mathit{while}(\mathit{TransCnd}, \mathit{Kcfg}) \\ K^\flat ::= \mathit{Kcfg} \curvearrowright \lfloor PC \rfloor \end{cases}$$

The idea behind the above cell definition is that each node in the control flow graph (CFG) receives a unique label from the PC set, meaning that each basic statement is indexed with a positive integer. The control structures appear in Kcfg as $\mathit{if}(\mathit{Kcfg}, \mathit{Kcfg})$ for branching and $\mathit{while}(\mathit{TransCnd}, \mathit{Kcfg})$ for loops (branching with return). The transformation of the program into the CFG is formally defined as $K^\flat ::= \mathit{Kcfg} \curvearrowright \lfloor PC \rfloor$ and is handled by $k^\flat(\_)$ for which we do not give definition here (for more details on this definition see $k^\sharp(\_)$ in [1]). The definition of $K^\flat$ structure emphasizes that at the end of the control flow graph, here defined by Kcfg, we place an additional program point $\lfloor \_ \rfloor$ marking the exit point of the program. Obviously, when the program has more than one final program points (not the case with *SIMP*), all these are to be collapsed in this newly added one.

The CFG of the program in Fig. 2, provided by $k^\flat(\_)$, is the following one:

$$\begin{array}{l} 1{:}\mathit{asg}(\mathtt{x}, 0) \curvearrowright 2{:}\mathit{asg}(\mathtt{err}, \mathtt{x}) \\ \curvearrowright \mathit{while}(3{:}\mathit{cnd}(\mathtt{y}{\leq}0), 4{:}\mathit{asg}(\mathtt{x}, \mathtt{x}{+}1) \curvearrowright 5{:}\mathit{asg}(\mathtt{y}, \mathtt{y}{+}\mathtt{x}) \\ \curvearrowright 6{:}\mathit{asg}(\mathtt{x}, -1{+}\mathtt{x}) \\ \curvearrowright 7{:}\mathit{cnd}(\neg(\mathtt{x}{=}0)) \curvearrowright \mathit{if}(8{:}\mathit{asg}(\mathtt{err}, 1), 9{:}\mathit{skip})) \curvearrowright \lfloor 10 \rfloor \end{array}$$

A $\mathsf{state}^\flat$ cell contains an abstract state which actually stands for a subset of concrete states. In fact, this subset of concrete states is given by a predicate which comes from the post-condition formula obtained after the symbolically executing a basic statement. Note that, according to Hoare logic, the postcondition may contain disjuncts as a result of branching or looping. However, we consider here a straight line symbolic execution, so the disjunction is eliminated by the choice of a particular branch at a branching point. This choice is provided by the path we consider to direct the symbolic execution. As a consequence, the postcondition formula is obtained as a conjunction of predicates, hence it is a reasonable decision to maintain this formula as a list of predicates. This choice is motivated by the existence in K of the List builtin. We give the formal definition of the contents of cell $\mathsf{state}^\flat$, and provide more explanations afterwards.

$$\langle \mathit{State}^\flat \rangle_{\mathsf{state}^\flat} \begin{cases} \mathit{State}^\flat ::= \mathit{Valid} \mid \mathit{False} \\ \mathit{Valid} ::= \mathsf{List}[\mathit{BExp}^\flat] \\ \mathit{BExp}^\flat ::= \mathit{AExp}^\flat {\leq} \mathit{AExp}^\flat \mid \mathit{AExp}^\flat {=} \mathit{AExp}^\flat \\ \qquad \mid \neg \mathit{BExp}^\flat \mid \mathit{BExp}^\flat \wedge \mathit{BExp}^\flat \\ \mathit{AExp}^\flat ::= \mathit{Var}^\flat \mid \mathit{Int} \mid \mathit{AExp}^\flat {+} \mathit{AExp}^\flat \mid \ldots \\ \mathit{Var}^\flat ::= \mathsf{Pair}[\mathit{Var}, \mathit{Int}] \end{cases}$$

From the beginning we make the distinction between a False state, representing the empty set of concrete states, and the rest of Valid states. As previously mentioned, a *valid* state is given by a list of predicates, the logical meaning behind this list being the satisfiable formula obtained by the conjunction of the predicates in the list. (We use this logical meaning of a valid $\mathsf{state}^\flat$ in the postcondition calculation from Fig. 4.) Moreover, the predicates defined in $\mathit{BExp}^\flat$ follow exactly the pattern of BExp with a significant difference: the program variables appear annotated by an integer standing for the current incarnation of that particular variable. We give more details on the incarnation notion in the description of cell ci.

A cell of type path contains a finite list of integers and represents a possible trace of a program execution prefix i.e., a path in the CFG (where only the CFG nodes are listed in their order of appearance). The cell $\mathsf{path}^\flat$ is defined

$$Configuration^\flat \equiv \langle\langle K^\flat\rangle_{\mathsf{k}^\flat}\,\langle State^\flat\rangle_{\mathsf{state}^\flat}\,\langle \mathsf{List}[Label]\,\rangle_{\mathsf{path}}\,\langle\mathsf{List}[Label]\,\rangle_{\mathsf{path}^\flat}\,\langle\mathsf{Map}[Var\mapsto Int]\rangle_{\mathsf{ci}}\,\langle\Phi^\flat\rangle_{\mathsf{inv}}\rangle_{\top^\flat}$$

$$Initialization^\flat\{\texttt{vars}\,xs;s\}^{\gamma,P_0,\phi} \equiv \langle\langle k^\flat(s)\rangle_{\mathsf{k}^\flat}\,\langle\gamma[xs_0/xs]\rangle_{\mathsf{state}^\flat}\,\langle P_0\rangle_{\mathsf{path}}\,\langle\cdot\rangle_{\mathsf{path}^\flat}\,\langle xs\mapsto 0\rangle_{\mathsf{ci}}\,\langle\phi[xs_0/xs]\rangle_{\mathsf{inv}}\rangle_{\top^\flat}$$

Fig. 3. K-configuration and initialization of the path directed symbolic execution.

similarly. However, from a semantic point of view, the cell $\mathsf{path}^\flat$ contains the current already symbolically executed trace, while the cell path contains the trace remaining to be checked for feasibility by our symbolic execution.

The presence of cell ci is justified by the fact that in a symbolic execution we need to come with fresh versions of the program variables called *incarnations*. To be more specific, the postcondition formula in a symbolic execution follows the principle of single static assignment, which means that each time an assignment takes place the assigned variable is considered a fresh incarnation of the program variable. We count these incarnations, and maintain in the cell ci the mapping of the program variables to an integer representing the current incarnation of each program variable. For example, if we have $\langle \_\,\mathsf{a}\mapsto 3\_\rangle_{\mathsf{ci}}$ then it means that there were three assignments $asg(\mathsf{a}, aexp)$ along the path currently symbolically executed. Say, the last such assignment was $asg(\mathsf{a}, \mathsf{a}+1)$. Then, according to the single static assignment principle, we consider the fresh variable a_3 as the current incarnation of a. Meanwhile, the assignment is interpreted as $asg(\mathsf{a}_3, \mathsf{a}_2+1)$.

An inv cell maintains the formula to be preserved by the symbolic execution along the given path, with the current incarnations for the program variables. In this work we focus on path-invariants $\phi \in \Phi ::= BExp$ which means that "formula ϕ is satisfied in any state, on the given path". However, the cell inv contains formulas φ (with $\varphi \in \Phi^\flat ::= BExp^\flat$) which are obtained by applying the substitution provided by the current incarnation $\langle\varsigma\rangle_{\mathsf{ci}}$ to the path-invariant ϕ (namely, $\varphi := \phi[\varsigma]$). The substitution with current incarnation is formalized as:

$$\_[\varsigma] : AExp \cup BExp \to AExp^\flat \cup BExp^\flat$$
$$E[\varsigma] := E[X_{\varsigma(X)}/X], \text{ for all } X \in keys(\varsigma).$$

Fig. 3 presents also the K structural rule for initialization of the path directed and property preserving symbolic execution, where $\texttt{vars}\,xs;s$ is the program, ϕ is the path-invariant property, P_0 is the path to guide the symbolic execution, and $\gamma \in \Phi$ is the precondition available at the beginning of the path P_0. As described in Fig. 3, the computation cell $\mathsf{k}^\flat$ is initialized with the control flow graph representation of the program statements (provided by $k^\flat(s)$). The ci cell contains the initial incarnation, which associates to all variables in the program the index 0. The initial content of the cell inv is $\phi[xs_0/xs]$ meaning that any occurrence of a program variable x in ϕ is replaced by its initial incarnation x_0. Analogously, the initial set of states is given by the path-precondition $\gamma[xs_0/xs]$, while P_0 is the content of the path cell. Note that the path cell contains the trace to be symbolically executed, while $\mathsf{path}^\flat$ contains the prefix of P_0 that has been traversed. Initially $\mathsf{path}^\flat$ cell is empty. Also, we assume that ϕ holds in γ (i.e. $\gamma \Rightarrow \phi$).

The K rules for path directed symbolic execution with path-invariant are described in Fig. 4. Note that in these rules the cells are considered to appear in the innermost environment wrapping them, according with the *locality principle* [18]. We consider $\Gamma \in Valid$ a valid abstract state and we make the implicit assumption that $false \in False$ is always differentiated from a valid abstract state Γ. In the following, we explain in details each of these rules.

The rules (R1,3) cover the base case when the currently obtained postcondition, found in the $\mathsf{state}^\flat$ cell, is $false$. This means that we hit a program point which cannot be reached from the previous $\mathsf{state}^\flat$ occurring at the previous program point (the last element in the cell $\mathsf{path}^\flat$). As such, we cannot continue the symbolic simulation of the current trace, hence we empty the content of the path cell because the symbolic execution cannot proceed in the current context.

The rules (R2,4-7) cover the cases when the cell $\mathsf{state}^\flat$ contains a valid symbolic state Γ, and the top of the cell path matches the current computation. In the structural rule (R2) the final program point is reached, hence the symbolic execution ends. In rule (R4) we have *skip* at the top of computation, in rule (R5) an assignment, while in rules (R6,7) the top of the computation is a branching condition. In all these computational rules, Γ stands for precondition formula, and we update the $\mathsf{state}^\flat$ cell with the formula given by the strongest postcondition $post\flat^{\varsigma}_{\varphi}$ (except for the case of *skip* where a $\mathsf{state}^\flat$ update is not required). The evaluation of the conditions in the definition of $post\flat^{\varsigma}_{\varphi}$ is directed to a builtin theorem prover, or to an SMT solver, whichever handles the logical backbone of the entire symbolic execution. More exactly, the formula $\alpha \Rightarrow \beta$ makes a call to the theorem prover or the SMT solver. The provided answer is YES/NO which we interpret as TRUE/FALSE in the condition of the rewrite rule. This means that we use $\alpha \Rightarrow \beta$ as a shorthand for $Answer := prove(valid(\alpha \to \beta))$ and $Answer = YES$. Note that we restrict our $prove$ calls to decidable theories (currently, the Presburger arithmetic). We introduce these conditions in order to ensure the fact that ϕ is a path-invariant (holds true in any reached symbolic state). Finally, note that rule (R5) updates in cell ci the current incarnation of the assigned variable, and makes invariant substitution accordingly (such that the invariant formula is always expressed in terms of the last incarnation of the program variables).

Rule (R8) is a structural rule and performs the standard unfolding of the $while$-loop in the control flow graph.

Example 1: For the program in Fig. 2, the path $\langle 1,2,3,4,5,6,7,8,3\rangle_{\mathsf{path}}$ and the path-invariant ϕ as formula $(\texttt{err} = 0)$ we describe in Fig. 5 the path directed symbolic execution starting with the precondition $(\texttt{err} \le 0)$. In the context of CEGAR, if we consider the path cell as the counterexample to be analyzed, the content of $\mathsf{state}^\flat$ cell at the end of the symbolic execution is $false$. In this case, this says that

(R1)
$$\langle \lfloor \ell \rfloor \rangle_{\mathsf{k}^\flat}\ \langle \mathit{false}\rangle_{\mathsf{state}^\flat}\ \langle \underset{\cdot}{\underline{\ell}} \rangle_{\mathsf{path}}$$

(R2)
$$\langle \underset{\cdot}{\underline{\lfloor \ell \rfloor}} \rangle_{\mathsf{k}^\flat}\ \langle \Gamma \rangle_{\mathsf{state}^\flat}\ \langle \underset{\cdot}{\underline{\ell}} \rangle_{\mathsf{path}}\ \langle \_\ \underset{\ell}{\underline{\cdot}} \rangle_{\mathsf{path}^\flat}$$

(R3)
$$\langle \ell : \_\ \_\rangle_{\mathsf{k}^\flat}\ \langle \mathit{false}\rangle_{\mathsf{state}^\flat}\ \langle \underset{\cdot}{\underline{\ell, P}} \rangle_{\mathsf{path}}$$

(R4)
$$\langle \ell : \mathit{skip}\ \_\rangle_{\mathsf{k}^\flat}\ \langle \Gamma \rangle_{\mathsf{state}^\flat}\ \langle \underset{\cdot}{\underline{\ell}}\ \_\rangle_{\mathsf{path}}\ \langle \_\ \underset{\ell}{\underline{\cdot}} \rangle_{\mathsf{path}^\flat}$$

(R5)
$$\langle \underset{\cdot}{\underline{\ell : asg(x,a)}}\ \_\rangle_{\mathsf{k}^\flat} \langle \underset{post\flat^{\varsigma}_{\varphi[x_{\varsigma(x)+_{Int}1}/x_{\varsigma[x]}]}(\Gamma, asg(x,a))}{\underline{\qquad\qquad\Gamma\qquad\qquad}} \rangle_{\mathsf{state}^\flat} \langle \underset{\cdot}{\underline{\ell}}\ \_\rangle_{\mathsf{path}}\ \langle \_\ \underset{\ell}{\underline{\cdot}} \rangle_{\mathsf{path}^\flat} \langle \underset{\varsigma[x \mapsto \varsigma(x) +_{Int} 1]}{\underline{\qquad\varsigma\qquad}} \rangle_{\mathsf{ci}} \langle \underset{\varphi[x_{\varsigma(x)+_{Int}1}/x_{\varsigma[x]}]}{\underline{\qquad\varphi\qquad}} \rangle_{\mathsf{inv}}$$

(R6)
$$\langle \underset{\ell' : t \curvearrowright K}{\underline{\ell : cnd(b) \curvearrowright if(\ell' : t \curvearrowright K,\ \_)}}\ \_\rangle_{\mathsf{k}^\flat}\ \langle \underset{post\flat^{\varsigma}_{\varphi}(\Gamma, cnd(b))}{\underline{\qquad\Gamma\qquad}} \rangle_{\mathsf{state}^\flat}\ \langle \underset{\cdot}{\underline{\ell}}, \ell'\ \_\rangle_{\mathsf{path}}\ \langle \_\ \underset{\ell}{\underline{\cdot}} \rangle_{\mathsf{path}^\flat}\ \langle \varsigma \rangle_{\mathsf{ci}}\ \langle \varphi \rangle_{\mathsf{inv}}$$

(R7)
$$\langle \underset{\ell' : t \curvearrowright K}{\underline{\ell : cnd(b) \curvearrowright if(\_,\ \ell' : t \curvearrowright K)}}\ \_\rangle_{\mathsf{k}^\flat}\ \langle \underset{post\flat^{\varsigma}_{\varphi}(\Gamma, cnd(\neg b))}{\underline{\qquad\Gamma\qquad}} \rangle_{\mathsf{state}^\flat}\ \langle \underset{\cdot}{\underline{\ell}}, \ell'\ \_\rangle_{\mathsf{path}}\ \langle \_\ \underset{\ell}{\underline{\cdot}} \rangle_{\mathsf{trace}\flat}\ \langle \varsigma \rangle_{\mathsf{ci}}\ \langle \varphi \rangle_{\mathsf{inv}}$$

(R8)
$$\langle \underset{\ell : cnd(b) \curvearrowright if(K \curvearrowright while(\ell : cnd(b), K),\ \cdot)}{\underline{\qquad while(\ell : cnd(b),\ K)\qquad}}\ \_\rangle_{\mathsf{k}^\flat}$$

where the postcondition formula $post\flat^{\varsigma}_{\varphi}$ is defined as follows:

$$post\flat^{\varsigma}_{\varphi} : \mathit{Valid} \times (\mathit{Asg} \cup \mathit{Cnd}) \rightarrow \mathit{State}^{\flat}$$

$$post\flat^{\varsigma}_{\varphi}(\Gamma, cnd(b)) := \begin{cases} \mathit{false} & , \text{ if } \Gamma \wedge b[\varsigma] \Rightarrow \mathit{false} \\ b[\varsigma] \mathbin{,,} \Gamma & , \text{ otherwise} \end{cases}$$

$$post\flat^{\varsigma}_{\phi}(\Gamma, asg(x,a)) := \begin{cases} (x_{\varsigma(x)+_{Int}1} = a[\varsigma]) \mathbin{,,} \Gamma & , \text{ if } \Gamma \wedge (x_{\varsigma(x)+_{Int}1} = a[\varsigma]) \Rightarrow \varphi \\ \mathit{false} & , \text{ otherwise} \end{cases}$$

Fig. 4. K rules for path directed symbolic execution with path-invariant.

the given path is a spurious counterexample. The content of the cell $\mathsf{path}^\flat$ at the end of the symbolic execution represents the feasible prefix of the counterexample. (By feasibility of the $\mathsf{path}^\flat$ we mean that there is a prefix of a concrete execution corresponding to the trace denoted by the program points $1, 2, 3, 4, 5, 6, 7$. Indeed, if we start in concrete the execution with the state $\langle \mathtt{err} \mapsto 0, \mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}}$, which satisfies the initial precondition ($\mathtt{err} \leq 0$), then the first steps of this concrete execution consumes from the computation cell k the statements from the program points $1, 2, 3, 4, 5, 6, 7$, in this order.) Meanwhile, if we consider the lazy abstraction as the refinement strategy in CEGAR, the content of the cell $k^\flat$ maintains the abstract computation which is supposed to be model checked with a refined abstraction in the next model checking phase.

We emphasize again that the description of the entire K-definition of path directed symbolic execution is also the description of its K-Maude implementation. This is due to the fact that K is a highly executable framework, and that K-Maude tool follows closely the K syntax, such that we get for free an executable prototype of the path directed symbolic execution from its K-definition.

IV. Correctness of the Path Directed Symbolic Execution

In this part we focus on proving the correctness of the K definition of *SIMP* path directed symbolic simulation with path-invariant. To do this, we formalize the notion of K transition system (KTS) and work out the correctness of our K symbolic execution by analysis of its associated KTS. We exploit here the fact that K is a unifying framework which places under the same umbrella both the programs' concrete executions and programs' verification methods.

Definition 1: A KTS is defined as the triplet $(\mathit{Configuration}, \mathit{Initialization}, \rightarrow \cup \rightharpoonup)$ where $\mathit{Configuration}$ contains all possible instances of a K-configuration, $\mathit{Initialization}$ is the subset of initial instances of the K-configuration, $\rightarrow$ is the transition relation given by the computational rules, while $\rightharpoonup$ is the transition relation given by the structural rules. The other notions for KTSs (as *successor*, *path*, *path fragment*, etc.) have standard definition. We call $\mathcal{K}_{SIMP}$ the KTS associated with the K definition in section II, and $\mathcal{K}^{\flat}_{SIMP}$ the KTS associated with the K definition described in section III.

Termination: We prove that the path directed symbolic execution, as defined in section III, terminates for any given finite path. Moreover, we provide a characterization of the terminating configurations.

Theorem 1: Any path in $\mathcal{K}^{\flat}_{SIMP}$ is finite.
This theorem essentially ensures the termination of the symbolic execution method described in the previous section.

Lemma 1: For any $\langle \_ \langle P \rangle_{\mathsf{path}}\ \_\rangle_{\top^\flat} \rightarrow \langle \_ \langle P' \rangle_{\mathsf{path}}\ \_\rangle_{\top^\flat}$ a computational transition, we have $|P| > |P'|$.
This lemma says that the computational rules decrease the content of the path cell, and gives the variant expression for the termination property in *Theorem* 1.

$$\langle\,\langle 1 : \mathit{asg}(\mathtt{x},0)\_\rangle_{\mathsf{k}^\flat}\,\langle(\mathtt{err}_0\le 0)\rangle_{\mathsf{state}^\flat}\,\langle 1,2,3,4,5,6,7,8,3\rangle_{\mathsf{path}}\,\langle\cdot\rangle_{\mathsf{path}^\flat}\,\langle \mathtt{err}\mapsto 0,\mathtt{x}\mapsto 0,\mathtt{y}\mapsto 0\rangle_{\mathsf{ci}}\,\langle\mathtt{err}_0=0\rangle_{\mathsf{inv}}\,\rangle_{\top^\flat}$$

$$\xrightarrow{R7}\langle\langle 2 : \mathit{asg}(\mathtt{err},\mathtt{x})\_\rangle_{\mathsf{k}^\flat}\,\langle(\mathtt{x}_1=0),,(\mathtt{err}_0\le 0)\rangle_{\mathsf{state}^\flat}$$
$$\langle 2,3,4,5,6,7,8,3\rangle_{\mathsf{path}}\,\langle 1\rangle_{\mathsf{path}^\flat}\,\langle \mathtt{err}\mapsto 0,\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 0\rangle_{\mathsf{ci}}\,\langle\mathtt{err}_0=0\rangle_{\mathsf{inv}}\,\rangle_{\top^\flat}$$

$$\xrightarrow{R5}\langle\langle\,\mathit{while}(3 : \mathit{cnd}(\mathtt{y}\le 0),K)\_\rangle_{\mathsf{k}^\flat}\,\langle(\mathtt{err}_1=\mathtt{x}_1),,(\mathtt{x}_1=0),,(\mathtt{err}_0\le 0)\rangle_{\mathsf{state}^\flat}$$
$$\langle 3,4,5,6,7,8,3\rangle_{\mathsf{path}}\,\langle 1,2\rangle_{\mathsf{path}^\flat}\,\langle \mathtt{err}\mapsto 1,\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 0\rangle_{\mathsf{ci}}\,\langle\mathtt{err}_1=0\rangle_{\mathsf{inv}}\,\rangle_{\top^\flat}$$

where K is $4 : \mathit{asg}(\mathtt{x},\mathtt{x}+1)\curvearrowright 5 : \mathit{asg}(\mathtt{y},\mathtt{y}+\mathtt{x})\curvearrowright 6 : \mathit{asg}(\mathtt{x},-1+\mathtt{x})\curvearrowright 7 : \mathit{cnd}(\neg(\mathtt{x}=0))\curvearrowright \mathit{if}(8 : \mathit{asg}(\mathtt{err},1),9 : \mathit{skip})$

$$\overset{R8}{\rightharpoonup}\ \langle\langle 3 : \mathit{cnd}(\mathtt{y}\le 0)\curvearrowright \mathit{if}(K\curvearrowright \mathit{while}(3 : \mathit{cnd}(\mathtt{y}\le 0),K),\cdot)\_\rangle_{\mathsf{k}^\flat}\,\langle(\mathtt{err}_1=\mathtt{x}_1),,(\mathtt{x}_1=0),,(\mathtt{err}_0\le 0)\rangle_{\mathsf{state}^\flat}$$
$$\langle 3,4,5,6,7,8,3\rangle_{\mathsf{path}}\,\langle 1,2\rangle_{\mathsf{path}^\flat}\,\langle \mathtt{err}\mapsto 1,\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 0\rangle_{\mathsf{ci}}\,\langle\mathtt{err}_1=0\rangle_{\mathsf{inv}}\,\rangle_{\top^\flat}$$

$$\xrightarrow{R6}\langle\langle K\curvearrowright \mathit{while}(3 : \mathit{cnd}(\mathtt{y}\le 0),K)\_\rangle_{\mathsf{k}^\flat}\,\langle(\mathtt{y}_0\le 0),,(\mathtt{err}_1=\mathtt{x}_1),,(\mathtt{x}_1=0),,(\mathtt{err}_0\le 0)\rangle_{\mathsf{state}^\flat}$$
$$\langle 4,5,6,7,8,3\rangle_{\mathsf{path}}\,\langle 1,2,3\rangle_{\mathsf{path}^\flat}\,\langle \mathtt{err}\mapsto 1,\mathtt{x}\mapsto 1,\mathtt{y}\mapsto 0\rangle_{\mathsf{ci}}\,\langle\mathtt{err}_1=0\rangle_{\mathsf{inv}}\,\rangle_{\top^\flat}$$

$$\xrightarrow{R5}\langle\langle 5 : \mathit{asg}(\mathtt{y},\mathtt{y}+\mathtt{x})\_\rangle_{\mathsf{k}^\flat}\,\langle(\mathtt{x}_2=\mathtt{x}_1+1),,(\mathtt{y}_0\le 0),,(\mathtt{err}_1=\mathtt{x}_1),,(\mathtt{x}_1=0),,(\mathtt{err}_0\le 0)\rangle_{\mathsf{state}^\flat}$$
$$\langle 5,6,7,8,3\rangle_{\mathsf{path}}\,\langle 1,2,3,4\rangle_{\mathsf{path}^\flat}\,\langle \mathtt{err}\mapsto 1,\mathtt{x}\mapsto 2,\mathtt{y}\mapsto 0\rangle_{\mathsf{ci}}\,\langle\mathtt{err}_1=0\rangle_{\mathsf{inv}}\,\rangle_{\top^\flat}$$

$$\xrightarrow{R5}\langle\langle 6 : \mathit{asg}(\mathtt{x},-1+\mathtt{x})\_\rangle_{\mathsf{k}^\flat}\,\langle(\mathtt{y}_1=\mathtt{y}_0+\mathtt{x}_2),,(\mathtt{x}_2=\mathtt{x}_1+1),,(\mathtt{y}_0\le 0),,(\mathtt{err}_1=\mathtt{x}_1),,(\mathtt{x}_1=0),,(\mathtt{err}_0\le 0)\rangle_{\mathsf{state}^\flat}$$
$$\langle 6,7,8,3\rangle_{\mathsf{path}}\,\langle 1,2,3,4,5\rangle_{\mathsf{path}^\flat}\,\langle \mathtt{err}\mapsto 1,\mathtt{x}\mapsto 2,\mathtt{y}\mapsto 1\rangle_{\mathsf{ci}}\,\langle\mathtt{err}_1=0\rangle_{\mathsf{inv}}\,\rangle_{\top^\flat}$$

$$\xrightarrow{R5}\langle\langle 7 : \mathit{cnd}(\neg(\mathtt{x}=0))\curvearrowright \mathit{if}(8 : \mathit{asg}(\mathtt{err},1),9 : \mathit{skip})\_\rangle_{\mathsf{k}^\flat}$$
$$\langle(\mathtt{x}_3=-1+\mathtt{x}_2),,(\mathtt{y}_1=\mathtt{y}_0+\mathtt{x}_2),,(\mathtt{x}_2=\mathtt{x}_1+1),,(\mathtt{y}_0\le 0),,(\mathtt{err}_1=\mathtt{x}_1),,(\mathtt{x}_1=0),,(\mathtt{err}_0\le 0)\rangle_{\mathsf{state}^\flat}$$
$$\langle 7,8,3\rangle_{\mathsf{path}}\,\langle 1,2,3,4,5,6\rangle_{\mathsf{path}^\flat}\,\langle \mathtt{err}\mapsto 1,\mathtt{x}\mapsto 3,\mathtt{y}\mapsto 1\rangle_{\mathsf{ci}}\,\langle\mathtt{err}_1=0\rangle_{\mathsf{inv}}\,\rangle_{\top^\flat}$$

$$\xrightarrow{R6}\langle\langle 8 : \mathit{asg}(\mathtt{err},1)\_\rangle_{\mathsf{k}^\flat}\,\langle \mathit{false}\rangle_{\mathsf{state}^\flat}\,\langle 8,3\rangle_{\mathsf{path}}\,\langle 1,2,3,4,5,6,7\rangle_{\mathsf{path}^\flat}\,\langle \mathtt{err}\mapsto 1,\mathtt{x}\mapsto 3,\mathtt{y}\mapsto 1\rangle_{\mathsf{ci}}\,\langle\mathtt{err}_1=0\rangle_{\mathsf{inv}}\,\rangle_{\top^\flat}$$

$$\xrightarrow{R3}\langle\langle 8 : \mathit{asg}(\mathtt{err},1)\_\rangle_{\mathsf{k}^\flat}\,\langle \mathit{false}\rangle_{\mathsf{state}^\flat}\,\langle\cdot\rangle_{\mathsf{path}}\,\langle 1,2,3,4,5,6,7\rangle_{\mathsf{path}^\flat}\,\langle \mathtt{err}\mapsto 1,\mathtt{x}\mapsto 3,\mathtt{y}\mapsto 1\rangle_{\mathsf{ci}}\,\langle\mathtt{err}_1=0\rangle_{\mathsf{inv}}\,\rangle_{\top^\flat}$$

Fig. 5. Example of a K symbolic execution along a particular path.

Lemma 2: There is no path in $\mathcal{K}^\flat_{SIMP}$ containing an infinite subsequence of structural transitions.

This lemma is proved using the termination of any set of structural rules (e.g., the rules for $k^\flat(pgm)$ terminate because the considered programs have a finite length, and the rules for $E[\varsigma]$ terminate because the expressions are always finite).

The proof for *Theorem* 1 is derived from *Lemmas* 1 and 2 and the fact that the content of the path cell is finite.

There are two types of terminating configurations (a terminating configuration appears at the end of a path in $\mathcal{K}^\flat_{SIMP}$ KTS). We enumerate and discuss them next:

(i) $\langle\,\langle K\rangle_{\mathsf{k}^\flat}\,\langle \mathit{false}\rangle_{\mathsf{state}^\flat}\,\langle\cdot\rangle_{\mathsf{path}}\,\langle P\rangle_{\mathsf{path}^\flat}\,\_\rangle_{\top^\flat}$
(ii) $\langle\,\langle K\rangle_{\mathsf{k}^\flat}\,\langle \Gamma\rangle_{\mathsf{state}^\flat}\,\langle\cdot\rangle_{\mathsf{path}}\,\langle P\rangle_{\mathsf{path}^\flat}\,\_\rangle_{\top^\flat}$

Type (i), characterized by the *false* state$^\flat$, appears either after the application of the rule (R5) for assignment, or rules (R6,7) for conditionals. In the first case path$^\flat$ cell contains a concrete counterexample (a bug) for CEGAR while, in the second case, the counterexample is spurious in CEGAR. Meanwhile, termination of type (ii), characterized by a valid state$^\flat$, always makes path$^\flat$ a spurious counterexample in the context of CEGAR. The distinction between concrete and spurious counterexample is made via an additional cell which we chose not to emphasize in the description of the path directed symbolic execution. Note that, if we consider the lazy abstraction as the refinement strategy in CEGAR, the content of the cell $k^\flat$ maintains the abstract computation which is supposed to be model checked with a refined abstraction in the next model checking phase, in the cases when the counterexample is proved spurious.

Relation between concrete and symbolic executions: The silent transitions in a KTS are given by the structural rules which perform configuration changes that preserve the computational semantics. Hence, we define the observable transitions in a KTS as follows:

Definition 2: Given a KTS $\mathcal{K} := (C, I, \rightarrow\cup\rightharpoonup)$, we define the observable transitions relation $\rightsquigarrow\ \subseteq C\times C$ as the path segment containing only one computational transition. This means that $c_1 \rightsquigarrow c_2$ iff there is a computational transition $c'_1\rightarrow c'_2$ such that $c_1 \overset{*}{\rightharpoonup} c'_1 \rightarrow c'_2 \overset{*}{\rightharpoonup} c_2$.

A particular simulation preorder for KTSs is defined in terms of observable transitions as follows:

Definition 3: Given two KTSs $\mathcal{K} := (C, I, \rightarrow\cup\rightharpoonup)$ and $\mathcal{K}' := (C', I', \rightarrow\cup\rightharpoonup)$, and a relation $R\subseteq C\times C'$, we say that R is an *observable stuttering simulation relation* for $(\mathcal{K},\mathcal{K}')$ iff

1. For any $c\in I$ there is $c'\in I'$ such that $c\,R\,c'$;
2. For any pair $(c_1, c'_1)\in R$ and any $c_2\in C$, if $c_1\rightsquigarrow c_2$ is an observable transition in $\mathcal{K}$ then there is a sequence of observable transitions $\wp := c'_1 \overset{+}{\rightsquigarrow} c'_2$ in K'^{t}_{s} such that
   i. $c_2\,R\,c'_2$, and
   ii. $c_1\,R\,c'$, for any $c'\in C'$ with $\wp = c'_1 \overset{*}{\rightsquigarrow} c' \overset{+}{\rightsquigarrow} c'_2$.

We say that $\mathcal{K}'$ *os-simulates* $\mathcal{K}$ if exists an observable stuttering simulation relation for $(\mathcal{K},\mathcal{K}')$.

Theorem 2: There exists an observable stuttering simulation relation $\ll^\flat$ for $(\mathcal{K}^\flat_{SIMP},\mathcal{K}_{SIMP})$ such that for any concrete state $\langle\_\,\langle\sigma\rangle_{\mathsf{state}}\,\_\rangle_{\top}$ in $\mathcal{K}_{SIMP}$ which os-simulates a state $\langle\_\,\langle\Gamma\rangle_{\mathsf{state}^\flat}\,\langle\varsigma\rangle_{\mathsf{ci}}\,\_\rangle_{\top^\flat}$ in $\mathcal{K}^\flat_{SIMP}$ (i.e., $\langle\_\,\langle\Gamma\rangle_{\mathsf{state}^\flat}\,\langle\varsigma\rangle_{\mathsf{ci}}\,\_\rangle_{\top^\flat} \ll^\flat \langle\_\,\langle\sigma\rangle_{\mathsf{state}}\,\_\rangle_{\top}$) we have $\sigma[\varsigma]\models\Gamma$.

Note that the path$^\flat$ cell characterizes a sequence of observable transitions in $\mathcal{K}^\flat_{SIMP}$. A corollary of *Theorem* 2 states that for any path directed symbolic execution there is a concrete execution os-simulating the content of the path$^\flat$ cell.

The proof of *Theorem* 2 uses induction on the length of the observable path in the $\mathcal{K}^\flat_{SIMP}$ KTS. At the inductive step we analyze the computational rules in $\mathcal{K}^\flat_{SIMP}$.

Path-invariant preservation: The path-invariant property ϕ holds true in any concrete execution os-simulating the symbolic execution given by path$^\flat$. This is formalized as:

Theorem 3: For any path π in $\mathcal{K}^\flat_{SIMP}$, where

$$\pi := \langle \_ \langle \cdot \rangle_{\mathsf{path}^\flat} \langle \phi[xs_0/xs] \rangle_{\mathsf{inv}} \_ \rangle_{\top^\flat} \overset{*}{\rightsquigarrow} \langle \_ \langle \Gamma \rangle_{\mathsf{state}^\flat} \langle \varsigma \rangle_{\mathsf{ci}} \langle \phi[\varsigma] \rangle_{\mathsf{inv}} \_ \rangle_{\top^\flat}$$

and any path ϖ in $\mathcal{K}_{SIMP}$, where

$$\varpi := \langle \_ \langle \sigma_0 \rangle_{\mathsf{state}} \_ \rangle_{\top} \overset{*}{\rightsquigarrow} \langle \_ \langle \sigma \rangle_{\mathsf{state}} \_ \rangle_{\top}$$

such that $\pi \ll^\flat \varpi$, we have that $\sigma \vDash \phi$ and $\Gamma \Rightarrow \phi[\varsigma]$.

The proof of this property is made by induction on the length of the content of path$^\flat$ cell using the result from *Theorem* 2, and the observation that if the current state$^\flat$ is valid, then so has to be its predecessor state$^\flat$. (The length of the path$^\flat$ cell content is actually a measure for the structure of the observable path in $\mathcal{K}^\flat_{SIMP}$.) We just emphasize a less obvious part of the proof, namely the case when the symbolic execution applies one of the rules (R6,7) for conditionals $\ell : cnd(b)$. Then Γ must be $post\flat^{\varsigma}_{\phi[\varsigma]}(\Gamma', cnd(b)) := b[\varsigma],, \Gamma'$ which is equivalent with $\Gamma' \wedge b[\varsigma]$. However, $\Gamma' \wedge b[\varsigma] \Rightarrow \phi[\varsigma]$ (because, from the inductive hypothesis, we have $\Gamma' \Rightarrow \phi[\varsigma]$). Since $\pi \ll^\flat \varpi$, we deduce that $\sigma[\varsigma] \vDash \Gamma' \wedge b[\varsigma]$, hence $\sigma[\varsigma] \vDash \phi[\varsigma]$, so $\sigma \vDash \phi$. In the case when rule (R5) for assignment is applied, the postcondition formula $post\flat^{\varsigma}_{\phi[\varsigma]}$ ensures that the next state$^\flat$ is valid only if the path-invariant holds true for it (from the condition $\Gamma \wedge (x_{\varsigma(x)} = a[\varsigma[x \mapsto \varsigma(x) -_{Int} 1]]) \Rightarrow \phi[\varsigma]$).

In the current section we reveal the astute nature of the K framework, which allows us to immediately define abstract machines for reasoning about the computations performed by K. This can be seen as an initial step towards (partially) automatizing the correctness proofs for verification tools. The idea of verifying the verifiers is currently championed by the "Kernel of Truth" project [20].

V. Conclusions and Future Work

In this paper we introduce, in the K framework, a path directed forward symbolic execution. This is used, in the context of counterexample guided abstraction refinement model checking, to prove or disprove an abstract counterexample. Our symbolic execution relies on strongest postcondition computation to propagate symbolic input values, while the resulting formulas are sent to an SMT solver. We also state the correctness of the path directed symbolic execution, setting premisses for a method to formally and, hopefully, automatically prove correctness of verification methods. We emphasize that along our entire work we exploit different characteristics of the K framework (e.g., unifying, executable, aso.). Our current work is meant to be part of a larger project which investigates K-based methods for program analysis and verification.

In near future we plan to embed predicate abstraction CEGAR into the K framework. Another line to pursue is approaching a more efficient symbolic simulation using interpolants, as described in [8]. On a longer term, 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.

References

[1] Asăvoae, I.M., Asăvoae, M.: Collecting semantics under predicate abstraction in the k framework. In: Proceedings of the 8th International Workshop on Rewriting Logic and its Applications (WRLA'10). Lecture Notes in Computer Science, vol. 6381. Springer-Verlag (2010), to appear

[2] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM 50(5), 752–794 (2003)

[3] Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)

[4] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007)

[5] 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)

[6] 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)

[7] 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)

[8] Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 232–244. ACM (2004)

[9] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. SIGPLAN Notices 37(1), 58–70 (2002)

[10] Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: TACAS'03: Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems. pp. 553–568. Springer-Verlag (2003)

[11] King, J.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)

[12] Martí-Oliet, N., Meseguer, J.: Rewriting logic as a logical and semantic framework. Electronic Notes in Theoretical Computer Science 4 (1996)

[13] McMillan, K.L.: Lazy abstraction with interpolants. In: Computer-Aided Verification. pp. 123–136 (2006)

[14] Meseguer, J., Roşu, G.: The rewriting logic semantics project. Electronic Notes in Theoretical Computer Science 156(1), 27–56 (2006)

[15] Ogata, K., Nakano, M., Kong, W., Futatsugi, K.: Induction-guided falsification. In: ICFEM. pp. 114–131 (2006)

[16] 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)

[17] Roşu, G., Ellison, C., Schulte, W.: From rewriting logic executable semantics to matching logic program verification. Tech. Rep. http://hdl.handle.net/2142/13159, University of Illinois (July 2009), http://hdl.handle.net/2142/13159

[18] Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)

[19] 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, vol. 6381. Springer-Verlag (2010), to appear

[20] Shankar, N.: Trust and automation in verification tools. In: ATVA. pp. 4–17 (2008)

[21] Sinha, N.: Symbolic program analysis using term rewriting and generalization. In: International Conference on Formal Methods in Computer-Aided Design. pp. 1–9. IEEE Press (2008)