Bounded Model Checking of Recursive Programs with Pointers in K

Irina Asăvoae, Frank Boer, Marcello Bonsangue, Dorel Lucanu, Jurriaan Rot

Bounded Model Checking of Recursive Programs with Pointers in $\mathbb{K}$

Irina Măriuca Asăvoae[1,⋆], Frank de Boer[2,3], Marcello M. Bonsangue[3,2], Dorel Lucanu[1], Jurriaan Rot[3,2,⋆⋆]

[1] Faculty of Computer Science - Alexandru Ioan Cuza University, Romania
{mariuca.asavoae, dlucanu}@info.uaic.ro
[2] Centrum voor Wiskunde en Informatica, The Netherlands
frb@cwi.nl
[3] LIACS — Leiden University, The Netherlands
{marcello, jrot}@liacs.nl

Abstract. We present an adaptation of model-based verification, via model checking pushdown systems, to semantics-based verification. First we introduce the algebraic notion of pushdown system specifications (PSS) and adapt a model checking algorithm for this new notion. We instantiate pushdown system specifications in the $\mathbb{K}$ framework by means of Shylock, a relevant PSS example. We show why $\mathbb{K}$ is a suitable environment for the pushdown system specifications and we give a methodology for defining the PSS in $\mathbb{K}$. Finally, we give a parametric $\mathbb{K}$ specification for model checking pushdown system specifications based on the adapted model checking algorithm for PSS.

Keywords: pushdown systems, model checking, the $\mathbb{K}$ framework

1 Introduction

The study of computation from a program verification perspective is an effervescent research area with many ramifications. We take into consideration two important branches of program verification which are differentiated based on their perspective over programs, namely model-based versus semantics-based program verification.

Model-based program verification relies on modeling the program as some type of transition system which is then analyzed with specific algorithms. Pushdown systems are known as a standard model for sequential programs with recursive procedures. Intuitively, pushdown systems are transition systems with a stack of unbounded size, which makes them strictly more expressive than finite

⋆ The research of this author has been partially supported by Project POSDRU/88/1.5/S/47646 and by Contract ANCS POS-CCE, O2.1.2, ID nr 602/12516, ctr.nr 161/15.06.2010 (DAK).

⋆⋆ The research of this author has been funded by the Netherlands Organisation for Scientific Research (NWO), CoRE project, dossier number: 612.063.920.

state systems. More importantly, there exist fundamental decidability results for pushdown systems [1] which enable program verification via model checking [17].

Semantics-based program verification relies on specification of programming language semantics and derives the program model from the semantics specification. For example, the rewriting logic semantics project [12] studies the unification of algebraic denotational semantics with operational semantics of programming languages. The main incentive of this semantics unification is the fact that the algebraic denotational semantics is executable via tools like the Maude system [10], or the $\mathbb{K}$ framework [14]. As such, a programming language (operational) semantics specification implemented with these tools becomes an interpreter for programs via execution of the semantics. The tools come with model checking options, so the semantics specification of a programming language have for-free program verification capabilities.

The current work solves the following problem in the rewriting logic semantics project: though the semantics expressivity covers a quite vast and interesting spectrum of programming languages, the offered verification capabilities via model checking are restricted to finite state systems. Meanwhile, the fundamental results from pushdown systems provide a strong incentive for approaching the verification of this class of infinite transition systems from a semantics-based perspective. As such, we introduce the notion of *pushdown system specifications* (PSS), which embodies the algebraic specification of pushdown systems. Furthermore, we adapt a state-of-the-art model checking algorithm for pushdown systems [17] to work for PSS and present an algebraic specification of this algorithm implemented in the $\mathbb{K}$ tool [15]. Our motivating example is Shylock, a programming language with recursive procedures and pointers, introduced by the authors in [16].

Related work. $\mathbb{K}$ is a rewriting logic based framework for the design, analysis, and verification of programming languages, originating in the rewriting logic semantics project. $\mathbb{K}$ specifies transition systems and is built upon a continuation-based technique and a series of notational conventions to allow for more compact and modular executable programming language definitions. Because of the continuation-based technique, $\mathbb{K}$ specifications resemble PSS where the stack is the continuation. The most complex and thorough $\mathbb{K}$ specification developed so far is the C semantics [5].

The standard approach to model checking programs, used for $\mathbb{K}$ specifications, involves the Maude LTL model checker [4] which is inherited from the Maude back-end of the $\mathbb{K}$ tool. The Maude LTL checker, by comparison with other model checkers, presents a great versatility in defining the state properties to be verified (these being given as a rewrite theory). Moreover, the actual model checking is performed on-the-fly, so that the Maude LTL checker can verify systems with states that involve data in types of infinite cardinality under the assumption of a *finite reachable state space*. However, this assumption is infringed by PSS because of the stack which is allowed to grow unboundedly, hence the Maude LTL checker cannot be used for PSS verification.

The Moped tool for model checking pushdown systems was successfully used for a subset of C programs [17] and was adapted for Java with full recursion, but with a fixed-size number of objects, in jMoped [6]. The WPDS++ tool [8] uses a weighted pushdown system model to verify x86 executable code. However, we cannot employ any of these dedicated tools for model checking pushdown systems because we work at a higher level, namely with *specifications* of pushdown system where we do not have the actual pushdown system.

Structure of the paper. In Section 2 we introduce pushdown system specifications and an associated invariant model checking algorithm. In Section 3 we introduce the $\mathbb{K}$ framework by showing how Shylock's PSS is defined in $\mathbb{K}$. In Section 4 we present the $\mathbb{K}$ specification of the invariant model checking for PSS and show how a certain type of bounded model checking can be directly achieved.

2 Model Checking Specifications of Pushdown Systems

In this section we discuss an approach to model checking pushdown system specifications by adapting an existing model checking algorithm for ordinary pushdown systems. Recall that a *pushdown system* is an input-less pushdown automaton without acceptance conditions. Basically, a pushdown system is a transition system equipped with a finite set of control locations and a stack. The stack consists of a non-a priori bounded string over some finite stack alphabet [1,17]. The difference between a pushdown system specification and an ordinary pushdown system is that the former uses production rules with open terms for the stack and control locations. This allows for a more compact representation of infinite systems and paves the way for applications of model checking to recursive programs defined by means of structural operational semantics.

We assume a countably infinite set of variables $\mathit{Var} = \{v_1, v_2, \ldots\}$. A signature Σ consists of a finite set of function symbols $g_1, g_2, \ldots$, each with a fixed arity $ar(g_1), ar(g_2), \ldots$. Function symbols with arity 0 are called constants. The set of terms, denoted by $T_\Sigma(\mathit{Var})$ and typically ranged over by s and t, is inductively defined from the set of variables Var and the signature Σ. A substitution σ replaces variables in a term with other terms. A term s can match term t if there exists a substitution σ such that $\sigma(t) = s$. A term t is said to be closed if no variables appear in t, and we use the convention that these terms are denoted as "hatted" terms, i.e., $\hat{t}$.

A *pushdown system specification* (PSS) is a tuple $(\Sigma, \Xi, \mathit{Var}, \Delta)$ where Σ and Ξ are two signatures, Var is a set of variables, and Δ is a finite set of production rules (defined below). Terms in $T_\Sigma(\mathit{Var})$ define *control locations* of a pushdown system, whereas terms in $T_\Xi(\mathit{Var})$ define the *stack alphabet*. A production rule in Δ is defined as a formula of the form $(s, \gamma) \Rightarrow (s', \Gamma)$, where s and s' are terms in $T_\Sigma(\mathit{Var})$, γ is a term in $T_\Xi(\mathit{Var})$, and Γ is a finite (possibly empty) sequence of terms in $T_\Xi(\mathit{Var})$. The pair (s, γ) is the *source* of the rule, and (s', Γ) is the *target*. We require for each rule that all variables appearing in the target are included in those of the source. A rule with no variables in the source is called

an *axiom*. The notions of substitution and matching are lifted to sequences of terms and to formulae as expected.

Example 1. Let $Var = \{s, t, \gamma\}$, let $\Sigma = \{0, a, +\}$ with $ar(0) = ar(a) = 0$ and $ar(+) = 2$, and let $\Xi = \{L, R\}$ with $ar(L) = ar(R) = 0$. Moreover consider the following three production rules, denoted as a set by Δ:

$$(a, \gamma) \Rightarrow (0, \varepsilon) \quad (s + t, L) \Rightarrow (s, R) \quad (s + t, R) \Rightarrow (t, LR)\,.$$

Then $(\Sigma, \Xi, Var, \Delta)$ is a pushdown system specification.

Given a pushdown system specification $\bar{\mathcal{P}} = (\Sigma, \Xi, Var, \Delta)$, a concrete configuration is a pair $\langle \hat{s}, \hat{\Gamma} \rangle$ where $\hat{s}$ is a closed term in $T_\Sigma(Var)$ denoting the *current control state*, and $\hat{\Gamma}$ is a finite sequence of closed terms in $T_\Xi(Var)$ representing the content of the *current stack*. A transition $\langle \hat{s}, \hat{\gamma} \cdot \hat{\Gamma} \rangle \longrightarrow \langle \hat{s}', \hat{\Gamma}' \cdot \hat{\Gamma} \rangle$ between concrete configurations is derivable from the pushdown system specification $\bar{\mathcal{P}}$ if and only if there is a rule $r = (s_r, \gamma_r) \Rightarrow (s'_r, \Gamma_r)$ in Δ and a substitution σ such that $\sigma(s_r) = \hat{s}$, $\sigma(\gamma_r) = \hat{\gamma}$, $\sigma(s'_r) = \hat{s}'$ and $\sigma(\Gamma_r) = \hat{\Gamma}'$. The above notion of pushdown system specification can be extended in the obvious way by allowing also conditional production rules and equations on terms.

Continuing on Example 1, we can derive the following sequence of transitions:

$$\langle a + (a + a), R \rangle \longrightarrow \langle a + a, LR \rangle \longrightarrow \langle a, RR \rangle \longrightarrow \langle 0, R \rangle\,.$$

Note that no transition is derivable from the last configuration $\langle 0, R \rangle$.

A pushdown system specification $\bar{\mathcal{P}}$ is said to be *locally finite* w.r.t. a concrete configuration $\langle \hat{s}, \hat{\Gamma} \rangle$, if the set of all closed terms appearing in the configurations reachable from $\langle \hat{s}, \hat{\Gamma} \rangle$ by transitions derivable from the rules of $\bar{\mathcal{P}}$ is finite. Note that this does not imply that the set of concrete configurations reachable from a configuration $\langle \hat{s}, \hat{\Gamma} \rangle$ is finite, as the stack is not bounded. However all reachable configurations are constructed from a *finite* set of control locations and a *finite* stack alphabet. An ordinary *finite pushdown system* is thus a pushdown system specification which is locally finite w.r.t. a concrete *initial* configuration $\hat{c}_0$, and such that all rules are axioms, i.e., all terms appearing in the source and target of the rules are closed.

For example, if we add $(s, L) \Rightarrow (s+a, L)$ to the rules of the pushdown system specification $\bar{\mathcal{P}}$ defined in Example 1, then it is not hard to see that there are infinitely many different location reachable from $\langle a, L \rangle$, meaning that $\bar{\mathcal{P}}$ is not locally finite w.r.t. the initial configuration $\langle a, L \rangle$. However, if instead we add the rule $(s, L) \Rightarrow (s, LL)$ then all reachable configurations from $\langle a, L \rangle$ will only use a or 0 as control locations and L as the only element of the stack alphabet. In this case $\bar{\mathcal{P}}$ is locally finite w.r.t. the initial configuration $\langle a, L \rangle$.

2.1 A Model Checking Algorithm for PSS

Next we describe a model checking algorithm for (locally finite) pushdown system specifications. We adapt the algorithm for checking LTL formulae against pushdown systems, as presented in [17], which, in turn, exploits the result from [1],

where it is proved that for any finite pushdown system the set $R(\hat{c}_0)$ of all configurations reachable from the initial configuration $\hat{c}_0$ is regular. The LTL model checking algorithm in [17] starts by constructing a finite automaton which recognizes this set $R(\hat{c}_0)$. This automaton has the property that $\langle \hat{s}, \hat{\Gamma} \rangle \in R(\hat{c}_0)$ if the string $\hat{\Gamma}$ is accepted in the automaton, starting from $\hat{s}$.

According to [17], the automaton associated to $R(\hat{c}_0)$, denoted by A_{post^*}, can be constructed in a forward manner starting with $\hat{c}_0$, as described in Fig. 1. We use the notation $\hat{x} \in T_\Sigma(Var)$ for closed terms representing control states in $\bar{\mathcal{P}}$, $\hat{\gamma}, \hat{\gamma}_1, \hat{\gamma}_2 \in T_\Xi(Var)$ for closed terms representing stack letters, $\hat{y}_{\hat{x},\hat{\gamma}}$ for the new states of the A_{post^*} automaton, f for the final states in A_{post^*}, while $\hat{y}, \hat{z}, \hat{u}$ stand for any state in A_{post^*}. The transitions in A_{post^*} are denoted by $\hat{y} \overset{\hat{\gamma}}{\rightsquigarrow} \hat{z}$ or $\hat{y} \overset{\varepsilon}{\rightsquigarrow} \hat{z}$. The notation $\hat{y} \overset{\hat{\Gamma}}{\rightsquigarrow} \hat{z}$, where $\hat{\Gamma} = \hat{\gamma}_1..\hat{\gamma}_n$, stands for $\hat{y} \overset{\hat{\gamma}_1}{\rightsquigarrow} .. \overset{\hat{\gamma}_n}{\rightsquigarrow} \hat{z}$.

In Fig. 1 we present how the reachability algorithm in [17] for generating A_{post^*} can be adjusted to invariant model checking pushdown system specifications. We emphasize that the transformation is minimal and consists in:

(a) The modification in the lines containing the code:
"**for all** $\hat{z}$ such that $\langle \hat{x}, \hat{\gamma} \rangle \hookrightarrow \langle \hat{z}, \hat{} \rangle$ is a rule in the pushdown system **do**"
i.e., lines 9, 12, 15 in Fig. 1, where instead of *rules in the pushdown system* we use *transitions derivable from the pushdown system specification* as follows:
"**for all** $\hat{z}$ such that $\langle \hat{x}, \hat{\gamma} \rangle \longrightarrow \langle \hat{z}, \hat{} \rangle$ is derivable from $\bar{\mathcal{P}}$ **do**"
(b) The addition of lines 1, 10, 13, 16 where the state invariant ϕ is checked to hold in the newly discovered control state y.

This approach for producing the A_{post^*} in a "breadth-first" manner is particularly suitable for specifications of pushdown systems as we can use the newly discovered configurations to produce transitions based on Δ, the production rules in $\bar{\mathcal{P}}$. Note that we assume, without loss of generality, that the initial stack has one symbol on it.

Note that in the algorithm Apost* of [17], the set of states of the automaton is determined statically at the beginning. This is clearly not possible starting with a PSS, because this set is not known in advance, and could be infinite if the algorithm does not terminate. Hence, the states that are generated when needed, that is, in line 9, 12 and 15, where the derivable transitions are considered.

We give next some keynotes on the algorithm in Fig. 1. The "trans" variable is a set containing the transitions to be processed. Along the execution of the algorithm **Apost***$(\phi, \bar{\mathcal{P}})$, the transitions of the A_{post^*} automaton are incrementally deposited in the "rel" variable which is a set where we collect transitions in the A_{post^*} automaton. The outermost **while** is executed until the end, i.e., until "trans" is empty, only if all states satisfy the control state formula ϕ. Hence, the algorithm in Fig. 1 verifies the invariant $\Box\phi$. In case ϕ is a state invariant for the pushdown system specification, the algorithm collects in "rel" the entire automaton A_{post^*}. Otherwise, the algorithm stops at the first encountered state x which does not satisfy the invariant ϕ.

Note that the algorithm in Fig. 1 assumes that the pushdown system specification has only rules which push on the stack at most two stack letters. This

Algorithm Apost* $(\phi, \bar{\mathcal{P}})$
Input: a initial concrete configuration $\langle \hat{x}_0, \hat{\gamma}_0 \rangle$.

if $\hat{x}_0 \not\models \phi$ **then return** false;
trans := $\{\hat{x}_0 \overset{\hat{\gamma}_0}{\rightsquigarrow} f\}$;
rel := $\emptyset$;
while trans = $\{\hat{x} \overset{\hat{\gamma}}{\rightsquigarrow} \hat{y}\} \cup$ trans' **do**
  trans := trans';
  if $\hat{x} \overset{\hat{\gamma}}{\rightsquigarrow} \hat{y} \notin$ rel **then**
    rel := rel $\cup \{\hat{x} \overset{\hat{\gamma}}{\rightsquigarrow} \hat{y}\}$;
    if $\hat{\gamma} \neq \varepsilon$ **then**
      for all $\hat{z}$ such that $\langle \hat{x}, \hat{\gamma} \rangle \longrightarrow \langle \hat{z}, \varepsilon \rangle$ is derivable from $\bar{\mathcal{P}}$ **do**
        if $\hat{z} \not\models \phi$ **then return** false;
        trans := trans $\cup \{\hat{z} \overset{\varepsilon}{\rightsquigarrow} \hat{y}\}$;
      for all $\hat{z}$ such that $\langle \hat{x}, \hat{\gamma} \rangle \longrightarrow \langle \hat{z}, \hat{\gamma}_1 \rangle$ is derivable from $\bar{\mathcal{P}}$ **do**
        if $\hat{z} \not\models \phi$ **then return** false;
        trans := trans $\cup \{\hat{z} \overset{\hat{\gamma}_1}{\rightsquigarrow} \hat{y}\}$;
      for all $\hat{z}$ such that $\langle \hat{x}, \hat{\gamma} \rangle \longrightarrow \langle \hat{z}, \hat{\gamma}_1 \hat{\gamma}_2 \rangle$ is derivable from $\bar{\mathcal{P}}$ **do**
        if $\hat{z} \not\models \phi$ **then return** false;
        trans := trans $\cup \{\hat{z} \overset{\hat{\gamma}_1}{\rightsquigarrow} \hat{y}_{\hat{z},\hat{\gamma}_1}\}$;
        rel := rel $\cup \{\hat{y}_{\hat{z},\hat{\gamma}_1} \overset{\hat{\gamma}_2}{\rightsquigarrow} \hat{y}\}$;
        for all $\hat{u} \overset{\varepsilon}{\rightsquigarrow} \hat{y}_{\hat{z},\hat{\gamma}_1} \in$ rel **do**
          trans := trans $\cup \{\hat{u} \overset{\hat{\gamma}_2}{\rightsquigarrow} \hat{y}\}$;
    else
      for all $\hat{y} \overset{\hat{\gamma}_1}{\rightsquigarrow} \hat{z} \in$ rel **do**
        trans := trans $\cup \{\hat{x} \overset{\hat{\gamma}_1}{\rightsquigarrow} \hat{z}\}$;
od;
return true

Fig. 1. The algorithm for obtaining A_{post*} adapted for pushdown system specifications.

assumption is inherited from the algorithm for A_{post^*} in [17] where the requirement is imposed without loss of generality. The approach in [17] is to adopt a standard construction for pushdown systems which consists in transforming the rules that push on the stack more than two stack letters into multiple rules that push at most two letters. Namely, any rule $\hat{r}$ in the pushdown system, of the form $\langle \hat{x}, \hat{\gamma} \rangle \hookrightarrow \langle \hat{x}', \hat{\gamma}_1..\hat{\gamma}_n \rangle$ with $n \geq 3$, is transformed into the following rules:

$$\langle \hat{x}, \hat{\gamma} \rangle \hookrightarrow \langle \hat{x}', \hat{\nu}_{\hat{r},n-2}\hat{\gamma}_n \rangle, \langle \hat{x}', \hat{\nu}_{\hat{r},i} \rangle \hookrightarrow \langle \hat{x}', \hat{\nu}_{\hat{r},i-1}\hat{\gamma}_{i+1} \rangle, \langle \hat{x}', \hat{\nu}_{\hat{r},1} \rangle \hookrightarrow \langle \hat{x}', \hat{\gamma}_1\hat{\gamma}_2 \rangle$$

where $2 \leq i \leq n-2$ and $\hat{\nu}_{\hat{r},1}, .., \hat{\nu}_{\hat{r},n-2}$ are new stack letters. This transformation produces a new pushdown system which simulates the initial one, hence the assumption in the A_{post^*} generation algorithm does not restrict the generality.

However, the aforementioned assumption makes impossible the application of the algorithm **Apost*** to pushdown system specifications $\bar{\mathcal{P}}$ for which the stack can be increased with any number of stack symbols. The reason is that

15 **for all** $\hat{z}$ such that $\langle \hat{x}, \hat{\gamma} \rangle \longrightarrow \langle \hat{z}, \hat{\gamma}_1..\hat{\gamma}_n \rangle$ is derivable from $\bar{\mathcal{P}}$ with $n \geq 2$ **do**
16 **if** $\hat{z} \not\models \phi$ **then return** false;
17 trans := trans $\cup \{\hat{z} \overset{\hat{\gamma}_1}{\rightsquigarrow} \hat{y}_{\hat{z},\hat{\gamma}_1}\}$;
18 rel := rel $\cup \{\hat{y}_{\hat{z},\nu(\hat{r},i)} \overset{\hat{\gamma}_{i+2}}{\rightsquigarrow} \hat{y}_{\hat{z},\nu(\hat{r},i+1)} \mid 0 \leq i \leq n-2\}$;
where $\hat{r}$ denotes $\langle \hat{x}, \hat{\gamma} \rangle \longrightarrow \langle \hat{z}, \hat{\gamma}_1..\hat{\gamma}_n \rangle$
and $\nu(\hat{r}, i), 1 \leq i \leq n-2$ are new symbols
(i.e., ν is a new function symbol s.t. $ar(\nu) = 2$)
and $\hat{y}_{\hat{z},\nu(\hat{r},0)} = \hat{y}_{\hat{z},\hat{\gamma}_1}$ and $\hat{y}_{\hat{z},\nu(\hat{r},n-1)} = \hat{y}$
19 **for all** $\hat{u} \overset{\varepsilon}{\rightsquigarrow} \hat{y}_{\hat{z},\nu(\hat{r},i)} \in$ rel, $0 \leq i \leq n-2$ **do**
20 trans := trans $\cup \{\hat{u} \overset{\hat{\gamma}_{i+2}}{\rightsquigarrow} \hat{y}_{\hat{z},\nu(\hat{r},i+1)} \mid 0 \leq i \leq n-2\}$;

Fig. 2. The modification required by the generalization of the algorithm **Apost***.

$\bar{\mathcal{P}}$ defines rule schemas and we cannot identify beforehand which rule schema applies for which concrete configuration, i.e., we cannot identify the $\hat{r}$ in $\nu_{\hat{r},i}$. Our solution is to obtain a similar transformation on-the-fly, as we apply the **Apost*** algorithm and discover instances of rule schemas which increase the stack, i.e., we discover $\hat{r}$. This solution induces a localized modification of the lines 15 through 20 of the **Apost*** algorithm, as described in Fig. 2. We denote by **Apost*gen** the **Apost*** algorithm in Fig. 1 with the lines 15 through 20 replaced by the lines in Fig. 2. The correctness of the new algorithm is a rather simple generalization of the one presented in [17].

3 Specification of Pushdown Systems in $\mathbb{K}$

In this section we introduce $\mathbb{K}$ by means of an example of a PSS defined using $\mathbb{K}$, and we justify why $\mathbb{K}$ is an appropriate environment for PSS.

A $\mathbb{K}$ specification evolves around its *configuration*, a nested bag of labeled cells denoted as $\langle content \rangle_{\mathsf{label}}$, which defines the state of the specified transition system. The movement in the transition system is triggered by the $\mathbb{K}$ rules which define transformations made to the configuration. A key component in this mechanism is introduced by a special cell, labeled k, which contains a list of *computational tasks* that are used to trigger computation steps. As such, the $\mathbb{K}$ rules that specify transitions discriminate the modifications made upon the configuration based on the *current computation task*, i.e., the first element in the k-cell. This instills the stack aspect to the k-cell and induces the resemblance with a PSS. Namely, in a $\mathbb{K}$ configuration we make the conceptual separation between the k-cell, seen as the stack, and the rest of the cells which form the control location. Consequently, we promote $\mathbb{K}$ as a suitable environment for PSS.

In the remainder of this section we describe the $\mathbb{K}$ definition of Shylock by means of a PSS that is based on the operational semantics of Shylock introduced in [16]. In Section 3.1 we present the configuration of Shylock's $\mathbb{K}$ implementation with emphasis on the separation between control locations and stack elements. In Section 3.2 we introduce the $\mathbb{K}$ rules for Shylock, while in Section 3.3 we

point out a methodology of defining in $\mathbb{K}$ production rules for PSS. We use this definition to present $\mathbb{K}$ notations and to further emphasize and standardize a $\mathbb{K}$ style for defining PSS.

3.1 Shylock's $\mathbb{K}$ Configuration

The PSS corresponding to Shylock's semantics is given in terms of a programming language specification. First, we give a short overview of the syntax of Shylock as in [16], then describe how this syntax is used in Shylock's $\mathbb{K}$-configuration.

A Shylock program is finite set of *procedure declarations* of the form $p_i :: B_i$, where B_i is the *body* of procedure p_i and denotes a statement defined by the grammar:

$$B ::= a.f := b \mid a := b.f \mid a := \text{new} \mid [a = b]B \mid [a \neq b]B \mid B + B \mid B;\ B \mid p$$

We use a and b for program variables ranging over $G \cup L$, where G and L are two disjoint finite sets of global and local variables, respectively. Moreover we assume a finite set F of field names, ranged over by f. G, L, F are assumed to be defined for each program, as sets of *Id*s, and we assume a distinguished *initial* program procedure `main`.

Hence, the body of a procedure is a sequence of statements that can be: assignments or object creation denoted by the function "$\_:=\_$" where $ar(:=) = 2$ (we distinguish the object creation by the "new" constant appearing as the second argument of ":="); conditional statements denoted by "$[\_]\_$"; nondeterministic choice given by "$\_+\_$"; and function calls. Note that $\mathbb{K}$ proposes the BNF notation for defining the language syntax as well, with the only difference that the variables are replaced by their respective sorts.

A $\mathbb{K}$ configuration is a nested bag of labeled cells where the cell content can be one of the predefined types of $\mathbb{K}$, namely $K, Map, Set, Bag, List$. The $\mathbb{K}$ configuration used for the specification of Shylock is the following:

$$\langle K \rangle_{\mathsf{k}}\ \langle \langle Map \rangle_{\mathsf{var}}\ \langle \langle Map \rangle_{\mathsf{fld}^\star} \rangle_{\mathsf{h}} \rangle_{\mathsf{heap}}\ \langle \langle Set \rangle_{\mathsf{G}}\ \langle Set \rangle_{\mathsf{L}}\ \langle Set \rangle_{\mathsf{F}}\ \langle Map \rangle_{\mathsf{P}} \rangle_{\mathsf{pgm}}\ \langle K \rangle_{\mathsf{kAbs}}$$

The pgm-cell is designated as a program container where the cells $\mathsf{G}, \mathsf{L}, \mathsf{F}$ maintain the above described finite sets of variables and fields associated to a program, while the cell P maintains the set of procedures stored as a map, i.e., a set of map items $p \mapsto B$.

The heap-cell contains the current heap H which is formed by the variable assignment cell var and the field assignment cell h. The var cell contains the mapping from local and global variables to their associated identities ranging over $\mathbb{N}_\perp = \mathbb{N} \cup \{\perp\}$, where $\perp$ stands for "not-created". The h cell contains a set of fld cells, each cell associated to a field variable from F. The mapping associated to each field contains items of type $n \mapsto m$, where n, m range over the object identities space $\mathbb{N}_\perp$. Note that any fld-cell always contains the item $\perp \mapsto \perp$ and $\perp$ is never mapped to another object identity.

Intuitively, the contents of the heap-cell form a directed graph with nodes labeled by object identities (i.e., values from $\mathbb{N}_\perp$) and arcs labeled by field names.

Moreover, the contents of the var-cell (i.e., the variable assignment) define *entry nodes* in the graph. We use the notion of *visible heap*, denoted as $\mathcal{R}(H)$, for the set of nodes reachable in the heap H from the entry nodes.

The k-cell maintains the current continuation of the program, i.e., a list of syntax elements that are to be executed by the program. Note that the sort K is tantamount with an associative list of items separated by the set-aside symbol "$\curvearrowright$". The kAbs-cell is introduced for handling the heap modifications required by the semantics of certain syntactic operators. In this way, we maintain in the cell k only the "pure" syntactic elements of the language, and move into kAbs any additional computational effort used by the abstract semantics for object creation, as well as for procedure call and return.

In conclusion, the k-cell stands for the stack in a PSS $\bar{\mathcal{P}}$, while all the other cells, including kAbs, form together the control location. Hence the language syntax in $\mathbb{K}$ practically gives a sub-signature of the stack signature in $\bar{\mathcal{P}}$, while the rest of the cells give a sub-signature, the control location signature in $\bar{\mathcal{P}}$.

3.2 Shylock's $\mathbb{K}$ Rules

We present here the $\mathbb{K}$ rules which implement the abstract semantics of Shylock, according to [16]. Besides introducing the $\mathbb{K}$ notation for rules, we also emphasize on the separation of concerns induced by viewing the $\mathbb{K}$ definitions as PSS.

In $\mathbb{K}$ we distinguish between computational rules that describe state transitions, and structural rules that only prepare the current state for the next transition. Rules in $\mathbb{K}$ have a bi-dimensional localized notation that stands for "what is above a line is rewritten into what is bellow that line in a particular context given by the matching with the elements surrounding the lines". Note that the solid lines encode a *computational rule* in $\mathbb{K}$ which is associated with a rewrite rule, while the dashed lines denote a *structural rule* in $\mathbb{K}$, which is compiled by the $\mathbb{K}$-tool into a Maude equation.

The production rules in PSS are encoded in $\mathbb{K}$ by computational rules which basically express changes to the configuration triggered by an atomic piece of syntax matched at the top of the stack, i.e., the k-cell. An example of such encoding is the following rule:

$$\text{RULE } \langle \underline{a.f := b} \ \cdots \rangle_{\mathsf{k}} \ \langle \cdots \ v(a) \mapsto \underline{\_} \ \cdots \rangle_{\mathsf{fld}(f)} \ \langle v \rangle_{\mathsf{var}} \qquad \text{when } v(a) \neq_{Bool} \bot$$
$$\cdot \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad v(b)$$

which reads as: if the first element in the cell k is the assignment $a.f := b$ then this is consumed from the stack and the map associated to the field f, i.e., the content of the cell $\mathsf{fld}(f)$, is modified by replacing whatever object identity was pointed by $v(a)$ with $v(b)$, i.e., the object identity associated to the variable b by the current variable assignment v, only when a is already created, i.e., $v(a)$ is not $\bot$. Note that this rule is conditional, the condition being introduced by the keyword "when".

We emphasize the following notational elements in $\mathbb{K}$ that appear in the above rule: "_" which stands for "anything" and the ellipses "⋯". The meaning

of the ellipses is basically the same as "_" the difference being that the ellipses appear always near the cell walls and are interpreted according to the contents of the respective cell. For example, given that the content of the k-cell is a list of computational tasks separated by "$\curvearrowright$", the ellipses in the k-cell from the above rule signify that the assignment $a.f := b$ is at the top of the stack of the PSS. On the other hand, because the content of a fld cell is of sort *Map* which is a commutative sequence of map items, the ellipses appearing by both walls of the cell fld denote that the item $v(a) \mapsto \_$ may appear "anywhere" in the fld-cell. Meanwhile, the notation for the var cell signifies that v is the entire content of this cell, i.e., the map containing the variable assignment. Finally, "$\cdot$" stands for the null element in any $\mathbb{K}$ sort, hence "$\cdot$" replacing $a.f := b$ at the top of the k-cell stands for ε from the production rules in $\bar{\mathcal{P}}$.

All the other rules for assignment, conditions, and sequence are each implemented by means of a single computational rule which considers the associated piece of syntax at the top of the k-cell. The nondeterministic choice is implemented by means of two computational rules which replace $B_1 + B_2$ at the top of a k-cell by either B_1 or B_2.

Next we present the implementation of one of the most interesting rules in Shylock namely object creation. The common semantics for an object creation is the following: if the current computation (the first element in the cell k) is "a:=new", then whatever object was pointed by a in the var-cell is replaced with the "never used before" object "*oNew*" obtained from the cell $\langle \_ \rangle_{\mathsf{kAbs}}$. Also, the fields part of the heap, i.e., the content of h-cell, is updated by the addition of a new map item "$oNew \mapsto \bot$". However, in the semantics proposed by Shylock, the value of *oNew* is *the minimal address not used in the current visible heap* which is calculated by the function $\texttt{min}(\mathcal{R}(H)^c)$ that ends in the normal form $\texttt{oNew}(n)$. This represents the memory reuse mechanism which is handled in our implementation by the kAbs-cell. Hence, the object creation rules are:

$$\text{RULE } \langle a := \texttt{new} \ \cdots \rangle_{\mathsf{k}} \ \langle H \rangle_{\mathsf{heap}} \ \langle \ \underbrace{\cdot}_{\texttt{min}(\mathcal{R}(H)^c)} \ \rangle_{\mathsf{kAbs}}$$

$$\text{RULE } \langle a := \texttt{new} \ \cdots \rangle_{\mathsf{k}} \ \langle H_h \rangle_{\mathsf{h}} \ \langle \texttt{oNew}(n) \curvearrowright \ \underbrace{\cdot}_{\texttt{update } H_h \texttt{ with } n \mapsto \bot} \ \rangle_{\mathsf{kAbs}}$$

$$\text{RULE } \langle \underbrace{a := \texttt{new}}_{\cdot} \ \cdots \rangle_{\mathsf{k}} \ \langle \cdots \ x \mapsto \underbrace{\_}_{n} \ \cdots \rangle_{\mathsf{var}} \ \langle \ \underbrace{H_h}_{H'_h} \ \rangle_{\mathsf{h}} \ \langle \underbrace{\texttt{oNew}(n) \curvearrowright \texttt{updated}(H'_h)}_{\cdot} \rangle_{\mathsf{kAbs}}$$

where "$\texttt{min}(\mathcal{R}(H)^c)$" finds n, the first integer not in $\mathcal{R}(H)$, and ends in $\texttt{oNew}(n)$, then "update *Bag* with *MapItem*" adds $n \mapsto \bot$ to the map in each cell fld contained in the h-cell and ends in the normal form $\texttt{updated}(Bag)$. Note that all the operators used in the kAbs-cell are implemented equationally, by means of structural $\mathbb{K}$-rules. In this manner, we ensure that the computational rule which consumes $a :=$ new from the top of the k-cell is accurately updating the control location with the required modification.

The rules for procedure call/return are presented in Fig. 3. They follow the same pattern as the one proposed in the rules for object creation. The renaming

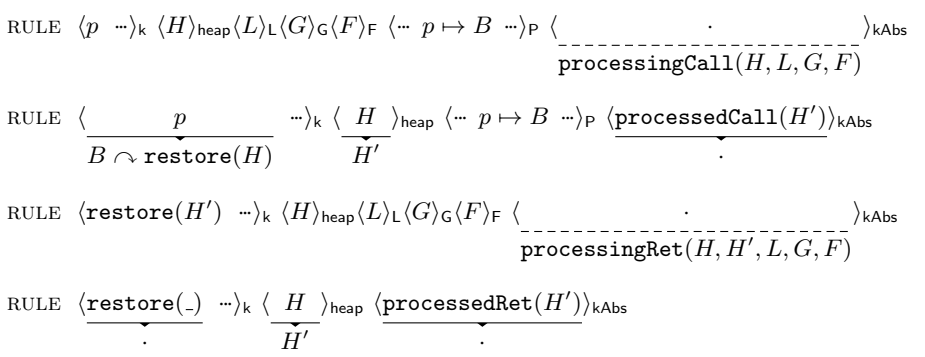

Fig. 3. $\mathbb{K}$-rules for the procedure's call and return in Shylock

scheme defined for resolving name clashes induced by the memory reuse for object creation is based in Shylock on the concept of *cut points* as introduced in [13]. Cut points are objects in the heap that are referred to from both local and global variables, and as such, are subject to modifications during a procedure call. Recording cut points in extra logical variables allows for a sound return in the calling procedure, enabling a precise abstract execution w.r.t. object identities. For more details on the semantics of Shylock we refer to [16].

3.3 Shylock as PSS

The benefit of a Shylock's $\mathbb{K}$ specification lies in the rules for object creation, which implement the memory reuse mechanism, and for procedure call/return, which implement the renaming scheme. Each element in the memory reuse mechanism is implemented equationally, i.e., by means of structural $\mathbb{K}$ rules which have equational interpretation when compiled in Maude. Hence, if we interpret Shylock as an abstract model for the standard semantics, i.e., with standard object creation, the $\mathbb{K}$ specification for Shylock's abstract semantics renders an equational abstraction. As such, Shylock is yet another witness to the versatility of the equational abstraction methodology [11].

Under the assumption of a bounded heap, the $\mathbb{K}$ specification for Shylock is a locally finite PSS and compiles in Maude into a rewriting system. Obviously, in the presence of recursive procedures, the stack grows unboundedly and, even if Shylock produces a finite pushdown system, the equivalent transition system is infinite and so is the associated rewriting system. We give next a relevant example for this idea.

Example 2. The following Shylock program, denoted as `pgm0`, is the basic example we use for Shylock. It involves a recursive procedure `p0` which creates an object `g`.

```
gvars: g        main :: p0        p0 :: g:=new; p0
```

In a standard semantics, because the recursion is infinite, so is the set of object identities used for g. However, Shylock's memory reuse guarantees to produce a finite set of object identities, namely $\bot, 0, 1$. Hence, the pushdown system associated to pgm0 Shylock program is finite and has the following (ground) rules:

$(\mathtt{g}{:}\bot, \mathtt{main}) \hookrightarrow (\mathtt{g}{:}\bot, \mathtt{p0}; \mathtt{restore}(\mathtt{g}{:}\bot))$
$(\mathtt{g}{:}\bot, \mathtt{p0}) \hookrightarrow (\mathtt{g}{:}\bot, \mathtt{g} := \mathtt{new}; \mathtt{p0}; \mathtt{restore}(\mathtt{g}{:}\bot))$ $(\mathtt{g}{:}\bot, \mathtt{g} := \mathtt{new}) \hookrightarrow (\mathtt{g}{:}0, \epsilon)$
$(\mathtt{g}{:}0, \mathtt{p0}) \hookrightarrow (\mathtt{g}{:}0, \mathtt{g} := \mathtt{new}; \mathtt{p0}; \mathtt{restore}(\mathtt{g}{:}0))$ $(\mathtt{g}{:}0, \mathtt{g} := \mathtt{new}) \hookrightarrow (\mathtt{g}{:}1, \epsilon)$
$(\mathtt{g}{:}1, \mathtt{p0}) \hookrightarrow (\mathtt{g}{:}1, \mathtt{g} := \mathtt{new}; \mathtt{p0}; \mathtt{restore}(\mathtt{g}{:}1))$ $(\mathtt{g}{:}1, \mathtt{g} := \mathtt{new}) \hookrightarrow (\mathtt{g}{:}0, \epsilon)$

Note that we cannot obtain the pushdown system by the exhaustive execution of Shylock[pgm0] because the exhaustive execution is infinite due to recursive procedure p0. For the same reason, Shylock[pgm0] specification does not comply with Maude's LTL model checker prerequisites. Moreover, we cannot use directly the dedicated pushdown systems model checkers as these work with the pushdown system automaton, while Shylock[pgm0] is a pushdown system specification. This example creates the premises for the discussion in the next section where we present a $\mathbb{K}$-specification of a model checking procedure amenable for pushdown systems specifications.

4 Model Checking $\mathbb{K}$ Definitions

We recall that the PSS perspective over the $\mathbb{K}$ definitions enables the verification by model checking of a richer class of programs which allow (infinite) recursion. In this section we focus on describing $kA_{post^*}(\phi, \bar{\mathcal{P}})$, the $\mathbb{K}$ specification of the algorithm **Apost*gen**. Note that $kA_{post^*}(\phi, \bar{\mathcal{P}})$ is parametric, where the two parameters are $\bar{\mathcal{P}}$, the $\mathbb{K}$ specification of a pushdown system, and ϕ a control state invariant. We describe $kA_{post^*}(\phi, \bar{\mathcal{P}})$ along justifying the behavioral equivalence with the algorithm **Apost*gen**.

The **while** loop in **Apost*gen**, in Fig. 1, is maintained in kA_{post^*} by the application of rewriting, until the term reaches the normal form, i.e. no other rule can be applied. This is ensured by the fact that from the initial configuration:

$$Init \equiv \langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces'}}\langle\langle x_0 \overset{\gamma_0}{\rightsquigarrow} f\rangle_{\mathsf{trans}}\langle\cdot\rangle_{\mathsf{rel}}\langle\cdot\rangle_{\mathsf{memento}}\langle\phi\rangle_{\mathsf{formula}}\langle true\rangle_{\mathsf{return}}\rangle_{\mathsf{collect}}$$

the rules keep applying, as long as trans-cell is nonempty.

We assume that the rewrite rules are applied at-random, so we need to direct/pipeline the flow of their application via matching and conditions. The notation RULEi [*label*] in the beginning of each rule hints, via [*label*], towards which part of the **Apost*gen** algorithm that rule is handling. In the followings we discuss each rule and justify its connection with code fragments in **Apost*gen**.

The last rule, RULE$\mathcal{P}$, performs the exhaustive unfolding for a particular configuration in cell trace. We use this rule in order to have a parametric definition of the kA_{post^*} specification, where one of the parameters is $\bar{\mathcal{P}}$, i.e., the $\mathbb{K}$ specification of the pushdown system. Recall that the other parameter is the specification of the language defining the control state invariant properties

$Init \equiv \langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces'}}\langle\langle x_0 \overset{\gamma_0}{\rightsquigarrow} f\rangle_{\mathsf{trans}}\langle\cdot\rangle_{\mathsf{rel}}\langle\cdot\rangle_{\mathsf{memento}}\langle\phi\rangle_{\mathsf{formula}}\langle true\rangle_{\mathsf{return}}\rangle_{\mathsf{collect}}$

RULE1 [**if** $\hat{x} \overset{\hat{\gamma}}{\rightsquigarrow} \hat{y} \notin$ rel **else**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces'}}\langle\cdots\ \langle\cdots\ \underbrace{x \overset{\gamma}{\rightsquigarrow} y}_{\cdot}\ \cdots\rangle_{\mathsf{trans}}\langle\cdots\ x \overset{\gamma}{\rightsquigarrow} y\ \cdots\rangle_{\mathsf{rel}}\langle\cdot\rangle_{\mathsf{memento}}\ \cdots\rangle_{\mathsf{collect}}$

RULE2 [**if** $(\hat{x} \overset{\hat{\gamma}}{\rightsquigarrow} \hat{y} \notin$ rel **then**...**if** $\hat{\gamma} \neq \epsilon$ **else**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces'}}\langle\cdots\ \langle\cdots\ \underbrace{x \overset{\epsilon}{\rightsquigarrow} y}_{(x \rightsquigarrow Rel[y \bar{\rightsquigarrow}])}\ \cdots\rangle_{\mathsf{trans}}\langle\underbrace{\cdot}_{x \overset{\epsilon}{\rightsquigarrow} y} Rel\rangle_{\mathsf{rel}}\langle\cdot\rangle_{\mathsf{memento}}\ \cdots\rangle_{\mathsf{collect}}$

when $x \overset{\epsilon}{\rightsquigarrow} y \notin Rel$

RULE3 [**if** $\hat{x} \overset{\hat{\gamma}}{\rightsquigarrow} \hat{y} \notin$ rel **then**...**if** $\hat{\gamma} \neq \epsilon$ **then**] :

$\langle\underbrace{\cdot}_{\langle\langle x\rangle_{\mathsf{ctrl}}\langle\gamma\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces'}}\langle\cdots\ \langle\cdots\ \underbrace{x \overset{\gamma}{\rightsquigarrow} y}_{\cdot}\ \cdots\rangle_{\mathsf{trans}}\langle\underbrace{\cdot}_{x \overset{\gamma}{\rightsquigarrow} y} Rel\rangle_{\mathsf{rel}}\langle\underbrace{\cdot}_{x \overset{\gamma}{\rightsquigarrow} y}\rangle_{\mathsf{memento}}\ \cdots\rangle_{\mathsf{collect}}$

when $x \overset{\gamma}{\rightsquigarrow} y \notin Rel\ and_{Bool}\ \gamma \neq \varepsilon$

RULE4 [**for all** $\hat{z}$ s.t. $\langle\hat{x},\hat{\gamma}\rangle \longrightarrow \langle\hat{z},\varepsilon|\hat{\gamma}|\hat{\gamma}_1..\hat{\gamma}_n\rangle$ is derivable from $\bar{\mathcal{P}}$ **do if** $\hat{z} \not\models \phi$ **then**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\underbrace{\langle\langle z\rangle_{\mathsf{ctrl}}\ \cdots\rangle_{\mathsf{trace}}}_{\cdot}\ \_\rangle_{\mathsf{traces'}}\langle\langle\underbrace{\_}_{\cdot}\rangle_{\mathsf{trans}}\langle\underbrace{\_}_{\cdot}\rangle_{\mathsf{rel}}\langle\underbrace{\_}_{\cdot}\rangle_{\mathsf{memento}}\langle\phi\rangle_{\mathsf{formula}}\langle\underbrace{true}_{false}\rangle_{\mathsf{return}}\rangle_{\mathsf{collect}}$

when $z \not\models \phi$

RULE5 [**for all** $\hat{z}$ s.t. $\langle\hat{x},\hat{\gamma}\rangle \longrightarrow \langle\hat{z},\varepsilon|\hat{\gamma}_1\rangle$ is derivable from $\bar{\mathcal{P}}$ **do**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\cdots\ \underbrace{\langle\langle z\rangle_{\mathsf{ctrl}}\langle\Gamma'\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}_{\cdot}\ \cdots\rangle_{\mathsf{traces'}}\langle\cdots\ \langle\cdots\ \underbrace{\cdot}_{z \overset{\Gamma'}{\rightsquigarrow} y}\ \cdots\rangle_{\mathsf{trans}}\langle x \overset{\gamma}{\rightsquigarrow} y\rangle_{\mathsf{memento}}\langle\phi\rangle_{\mathsf{formula}}\ \cdots\rangle_{\mathsf{collect}}$

when $|\Gamma'| \leq 1\ and_{Bool}\ z \models \phi$

RULE6 [**for all** $\hat{z}$ s.t. $\langle\hat{x},\hat{\gamma}\rangle \longrightarrow \langle\hat{z},\hat{\gamma}_1..\hat{\gamma}_n\rangle$ is derivable from $\bar{\mathcal{P}}$ **do**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\cdots\ \underbrace{\langle\langle z\rangle_{\mathsf{ctrl}}\langle\gamma' \curvearrowright \Gamma'\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}_{\cdot}\ \cdots\rangle_{\mathsf{traces'}}$

$\langle\cdots\ \underbrace{\cdot}_{z \overset{\gamma'}{\rightsquigarrow} new(z,\gamma')\,(Rel[\overset{\epsilon}{\rightsquigarrow} new(z,\gamma'), news(x,\gamma,z,\gamma',\Gamma')] \overset{\Gamma'}{\rightsquigarrow} news(x,\gamma,z,\gamma',\Gamma'),\, y)}\ \cdots\rangle_{\mathsf{trans}}$

$\langle Rel\ \underbrace{\cdot}_{new(z,\gamma'), news(x,\gamma,z,\gamma',\Gamma') \overset{\Gamma'}{\rightsquigarrow} news(x,\gamma,z,\gamma',\Gamma'), y}\rangle_{\mathsf{rel}}\langle x \overset{\gamma}{\rightsquigarrow} y\rangle_{\mathsf{memento}}\langle\phi\rangle_{\mathsf{formula}}$

when $|\Gamma'| \geq 1\ and_{Bool}\ z \models \phi$

RULE7 [**for all** $\hat{z}$ s.t. $\langle\hat{x},\hat{\gamma}\rangle \longrightarrow \langle\hat{z},\varepsilon|\hat{\gamma}|\hat{\gamma}_1..\hat{\gamma}_n\rangle$ is derivable from $\bar{\mathcal{P}}$ **do**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces'}}\langle\cdots\ \underbrace{\langle x \overset{\gamma}{\rightsquigarrow} y\rangle_{\mathsf{memento}}}_{\cdot}\ \cdots\rangle_{\mathsf{collect}}$

RULE$\mathcal{P}$ [**all** $\hat{z}$ s.t. $\langle\hat{x},\hat{\gamma}\rangle \longrightarrow \langle\hat{z},\hat{\Gamma}\rangle$ is derivable from $\bar{\mathcal{P}}$] :

$\langle\cdots\ \underbrace{\langle\langle x\rangle_{\mathsf{ctrl}}\langle\gamma \curvearrowright \Gamma\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}_{\cdot}\ \cdots\rangle_{\mathsf{traces}}\langle\cdots\ \underbrace{\cdot}_{\langle\langle z_0\rangle_{\mathsf{ctrl}}\langle\Gamma_0 \curvearrowright \Gamma\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}..\langle\langle z_n\rangle_{\mathsf{ctrl}}\langle\Gamma_n \curvearrowright \Gamma\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}\ \cdots\rangle_{\mathsf{traces'}}$

Fig. 4. $kA_{post^*}(\phi, \bar{\mathcal{P}})$

ϕ which are to be verified on the produced pushdown system. RULE$\mathcal{P}$ takes $\langle x\rangle_{\mathsf{ctrl}}\langle\gamma \curvearrowright \Gamma\rangle_{\mathsf{k}}$ a configuration in $\bar{\mathcal{P}}$ and gives, based on the rules in $\bar{\mathcal{P}}$, all the configurations $\langle z_i\rangle_{\mathsf{ctrl}}\langle\Gamma_i \curvearrowright \Gamma\rangle_{\mathsf{k}}, 0 \leq i \leq n$ obtained from $\langle x\rangle_{\mathsf{ctrl}}\langle\gamma \curvearrowright \Gamma\rangle_{\mathsf{k}}$ after exactly one rewrite.

The pipeline stages are the following sequence of rules' application:

$$\text{RULE3RULE}\mathcal{P}(\text{RULE4} + \text{RULE5} + \text{RULE6})^*\text{RULE7}$$

The cell memento is filled in the beginning of the pipeline, RULE3, and is emptied at the end of the pipeline, RULE7. We use the matching on a nonempty memento for localizing the computation in **Apost*gen** at the lines $7 - 20$. We explain next the pipeline stages.

Firstly, note that when no transition derived from $\bar{\mathcal{P}}$ is processed by kA_{post^*} we enforce cells $\mathsf{traces}, \mathsf{traces}'$ to be empty (with the matching $\langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}$). This happens in RULEs 1 and 2 because the respective portions in **Apost*gen** do not need new transitions derived from $\bar{\mathcal{P}}$ to update "trans" and "rel".

The other cases, namely when the transitions derived from $\bar{\mathcal{P}}$ are used for updating "trans" and "rel", are triggered in RULE3 by placing the desired configuration in the cell traces, while the cell traces' is empty. At this point, since all the other rules match on either traces empty, or traces' nonempty, only RULE$\mathcal{P}$ can be applied. This rule populates traces' with all the next configurations obtained by executing $\bar{\mathcal{P}}$.

After the application of RULE$\mathcal{P}$, only one of the RULEs $4, 5, 6$ can apply because these are the only rules in kA_{post^*} matching an empty traces and a nonempty traces'.

Among the rules 4,5,6 the differentiation is made via conditions as follows:

RULE4 handles all the cases when the new configuration has a control location z which does not verify the state invariant ϕ (i.e., lines $10, 13, 16$ in **Apost*gen**). In this case we close the pipeline and the algorithm by emptying all the cells $\mathsf{traces}, \mathsf{traces}, \mathsf{trans}$. Note that all the rules handling the **while** loop match on at least a nonempty cell $\mathsf{traces}, \mathsf{traces}$, or trans, with a pivot in a nonempty trans.

RULEs 5 and 6 are applied disjunctively of RULE4 because both have the condition $z \models \phi$. Next we describe these two rules. RULE5 handles the case when the semantic rule in $\bar{\mathcal{P}}$ which matches the current $< \hat{x}, \hat{\gamma} >$ does not increase the size of the stack. This case is associated with the lines 9 and 11, 12 and 14 in **Apost*gen**. RULE6 handles the case when the semantic rule in $\bar{\mathcal{P}}$ which matches the current $< \hat{x}, \hat{\gamma} >$ increases the stack size and is associated with lines 15 and $17 - 20$ in **Apost*gen**.

Both rules 5 and 6 use the memento cell which is filled upon pipeline initialization, in RULE3. The most complicated rule is RULE6, because it handles a **for all** piece of code, i.e., lines $17-20$ in Fig. 2. This part is reproduced by matching the entire content of cell rel with Rel, and using the projection operator:

$$Rel[\overset{\gamma}{\rightsquigarrow} z_1, .., z_n] := \{u \mid (u, \gamma, z_1) \in Rel\}, .., \{u \mid (u, \gamma, z_n) \in Rel\}$$

where $z_1, .., z_n$ in the left hand-side is a list of z-symbols, while in the right hand-side we have a list of sets. Hence, the notation:

$$(Rel[\overset{\epsilon}{\rightsquigarrow} new(z,\gamma'), news(x,\gamma,z,\gamma',\Gamma')] \overset{\Gamma'}{\rightsquigarrow} news(x,\gamma,z,\gamma',\Gamma'), y)$$

in RULE6 cell trans stands for the lines 17 and 19−20 in Fig. 2. (Note that instead of notation $\hat{r}$ for rule $< \hat{x}, \hat{\gamma} > \longrightarrow < \hat{z}, \hat{\gamma}' \hat{\Gamma}' >$ we use the equivalent unique representation $(\hat{x}, \hat{\gamma}, \hat{z}, \hat{\gamma}', \hat{\Gamma}')$ and that instead of $\hat{y}_{\hat{z},\nu(\hat{r},0)}$ we use directly $\hat{y}_{\hat{z},\hat{\gamma}'}$, i.e., $new(z,\gamma')$, while instead of $\hat{y}_{\hat{z},\nu(\hat{r},n-1)}$ in Fig. 2 we use directly $\hat{y}$.) Also, the notation in cell rel: "$new(z,\gamma'), news(x,\gamma,z,\gamma',\Gamma') \overset{\Gamma'}{\rightsquigarrow} news(x,\gamma,z,\gamma',\Gamma'), y$" stands for line 18 in Fig. 2.

RULEs $4, 5, 6$ match on a nonempty traces′-cell and an empty traces, and no other rule matches alike. RULE7 closes the pipeline when the traces′ cell becomes empty by making the memento cell empty. Note that traces′ empties because rules 4, 5, 6 keep consuming it.

Example 3. We recall that the Shylock program `pgm0` from Example 2 was not amenable by semantic exhaustive execution or Maude's LTL model checker, due to the recursive procedure `p0`. Likewise, model checkers for pushdown systems which can handle the recursive procedure `p0` cannot be used because Shylock[`pgm0`], the pushdown system obtained from Shylock's PSS, is not available. However, we can employ kA_{post^*} for Shylock's $\mathbb{K}$-specification in order to discover the reachable state space, the A_{post^*} automata, as well as the pushdown system itself. In the Fig. 5 we describe the first steps in the execution of $kA_{post^*}(true, \text{Shylock}[\texttt{pgm0}])$ and the reachability automaton generated automatically by $kA_{post^*}(true, \text{Shylock}[\texttt{pgm0}])$.

4.1 Bounded Model Checking for Shylock

One of the major problems in model checking programs which manipulate dynamic structures, such as linked lists, is that it is not possible to bound a priori the state space of the possible computations. This is due to the fact that programs may manipulate the heap by dynamically allocating an unbounded number of new objects and by updating reference fields. This implies that the reachable state space is potentially infinite for Shylock programs with recursive procedures. Consequently for model checking purposes we need to impose some suitable bounds on the model of the program.

A natural bound for model checking Shylock programs, without necessarily restricting their capability of allocating an unbounded number of objects, is to impose constraints on the size of the *visible* heap [2]. Such a bound still allows for storage of an unbounded number of objects onto the call-stack, using local variables. Thus termination is guaranteed with heap-bounded model checking of the form $\models_k \Box\phi$ meaning $\models \Box\phi \wedge le(k)$, where $le(k)$ verifies if the size of the visible heap is smaller than k.

$$\langle\cdot\rangle_{\mathsf{traces}}\ \langle\cdot\rangle_{\mathsf{traces}'}\ \langle\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{main}}{\rightsquigarrow} \mathtt{fin}\rangle_{\mathsf{trans}}\langle\cdot\rangle_{\mathsf{rel}}\langle\cdot\rangle_{\mathsf{memento}}\langle \mathit{true}\rangle_{\mathsf{formula}}\langle \mathit{true}\rangle_{\mathsf{return}}\ \rangle_{\mathsf{collect}}$$

$\overset{\text{RULE3}}{\Rrightarrow}\ \langle\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}\rangle_{\mathsf{ctrl}}\langle\mathtt{main}\rangle_{\mathsf{k}}\rangle_{\mathsf{traces}}$

$\quad\langle\cdot\rangle_{\mathsf{trans}}\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{main}}{\rightsquigarrow} \mathtt{fin}\rangle_{\mathsf{rel}}\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{main}}{\rightsquigarrow} \mathtt{fin}\rangle_{\mathsf{memento}}$

$\overset{\text{RULE}\mathcal{P}}{\Rrightarrow}\ \langle\cdot\rangle_{\mathsf{traces}}\ \langle\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}\rangle_{\mathsf{ctrl}}\langle \mathtt{p0}\curvearrowright\mathtt{restore}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}})\rangle_{\mathsf{k}}\rangle_{\mathsf{traces}'}$

$\overset{\text{RULE6}}{\Rrightarrow}\ \langle\cdot\rangle_{\mathsf{traces}}\ \langle\cdot\rangle_{\mathsf{traces}'}\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{main}}{\rightsquigarrow} \mathtt{fin}\rangle_{\mathsf{memento}}$

$\quad\langle\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{p0}}{\rightsquigarrow} \mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{p0})\ \rangle_{\mathsf{trans}}$

$\quad\langle\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{main}}{\rightsquigarrow} \mathtt{fin}\quad \mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{p0}) \overset{\mathtt{restore}(\ldots)}{\rightsquigarrow} \mathtt{fin}\ \rangle_{\mathsf{rel}}$

$\overset{\text{RULE7}}{\Rrightarrow}\ \langle\cdot\rangle_{\mathsf{traces}}\ \langle\cdot\rangle_{\mathsf{traces}'}\ \langle\cdot\rangle_{\mathsf{memento}}$

$\overset{\text{RULE3}}{\Rrightarrow}\ \langle\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}\rangle_{\mathsf{ctrl}}\langle\mathtt{p0}\rangle_{\mathsf{k}}\rangle_{\mathsf{traces}}$

$\quad\langle\cdot\rangle_{\mathsf{trans}}\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{p0}}{\rightsquigarrow} \mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{p0})\rangle_{\mathsf{memento}}$

$\quad\langle\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{main}}{\rightsquigarrow} \mathtt{fin}\quad \mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{p0}) \overset{\mathtt{restore}(\ldots)}{\rightsquigarrow} \mathtt{fin}$

$\quad\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{p0}}{\rightsquigarrow} \mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{p0})\ \rangle_{\mathsf{rel}}$

$\overset{\text{RULE}\mathcal{P}}{\Rrightarrow}\ \langle\cdot\rangle_{\mathsf{traces}}\langle\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}\rangle_{\mathsf{ctrl}}\langle \mathtt{g}:=\mathtt{new}\curvearrowright\mathtt{p0}\curvearrowright\mathtt{restore}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}})\rangle_{\mathsf{k}}\rangle_{\mathsf{traces}'}$

$\overset{\text{RULE6}}{\Rrightarrow}\ \langle\cdot\rangle_{\mathsf{traces}}\ \langle\cdot\rangle_{\mathsf{traces}'}\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{p0}}{\rightsquigarrow} \mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{p0})\rangle_{\mathsf{memento}}$

$\quad\langle\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{g:=new}}{\rightsquigarrow} \mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{g}:=\mathtt{new})\ \rangle_{\mathsf{trans}}$

$\quad\langle\ \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{main}}{\rightsquigarrow} \mathtt{fin}\quad \mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{p0}) \overset{\mathtt{restore}(\ldots)}{\rightsquigarrow} \mathtt{fin}$

$\quad\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}} \overset{\mathtt{p0}}{\rightsquigarrow} \mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{p0})$

$\quad\mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{g}:=\mathtt{new})$

$\quad\overset{\mathtt{p0}}{\rightsquigarrow}\mathtt{news}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{p0}, \langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{g}{:=}\mathtt{new}, \mathtt{p0}\curvearrowright\mathtt{restore}(\ldots), 1)$

$\quad\overset{\mathtt{restore}(\ldots)}{\rightsquigarrow}\ \mathtt{new}(\langle\langle\langle \mathtt{g}\mapsto\bot\rangle_{\mathsf{var}}\langle\cdot\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}, \mathtt{p0})\ \rangle_{\mathsf{rel}}$

$\overset{\text{RULE7}}{\Rrightarrow}\ \langle\cdot\rangle_{\mathsf{traces}}\ \langle\cdot\rangle_{\mathsf{traces}'}\ \langle\cdot\rangle_{\mathsf{memento}}\ \overset{\text{RULE3}}{\Rrightarrow}\ \ldots$

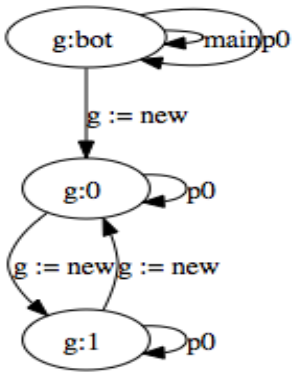

Fig. 5. The first pipeline iteration for $kA_{post^*}(\mathit{true}, \text{Shylock}[\mathtt{pgm0}])$ and the automatically produced reachability automaton at the end of $kA_{post^*}(\mathit{true}, \text{Shylock}[\mathtt{pgm0}])$. Note that for legibility reasons we omit certain cells appearing in the control state, like $\langle\langle\mathtt{g}\rangle_{\mathsf{G}}\langle\cdot\rangle_{\mathsf{L}}\langle\cdot\rangle_{\mathsf{F}}\langle\mathtt{main}\mapsto\mathtt{p0}\ \mathtt{p0}\mapsto\mathtt{g}:=\mathtt{new};\mathtt{p0}\rangle_{\mathsf{P}}\rangle_{\mathsf{pgm}}$, which do not change along the execution. Hence, for example, the ctrl-cell is filled in RULE3 with both cells heap and pgm.

To this end, we define the set of atomic propositions ($\phi \in$) *Rite* as the smallest set defined by the following grammar:

$$r ::= \varepsilon \mid x \mid \neg x \mid f \mid r.r \mid r + r \mid r^*$$

where x ranges over variable names (to be used as tests) and f over field names (to be used as actions). The atomic proposition in *Rite* are basically expressions from the Kleene algebra with tests [9], where the global and local variables are used as nominals while the fields constitute the set of basic actions. The $\mathbb{K}$ specification of *Rite* is based on the circularity principle [7,3] to handle the possible cycles in the heap. We employ *Rite* with $kA_{post^*}(\phi, \bar{\mathcal{P}})$, i.e., $\phi \in$ *Rite*, for verifying heap-shape properties for Shylock programs. For the precise definition of the interpretation of these expressions in a heap we refer to the companion paper [16]. We conclude with an example showing a simple invariant property of a Shylock program.

Example 4. The following Shylock program `pgmList` creates a potentially infinite linked list which starts in object `first` and ends with object `last`.

```
gvars: first, last     lvars: tmp     flds:  next
main ::  last:=new;  last.next:=last;  first:=last;  p0
p0 ::  tmp:=new;  tmp.next:=first;  first:=tmp;   (p0 + skip)
```

This is an example of a program which induces, on some computation path, an unbounded heap. When we apply the heap-bounded model checking specification, by instantiating ϕ with the property $le(10)$, we collect all lists with a length smaller or equal than 10. We can also check the heap-shape property "($\neg$`first`+`first.next`*`.last`)". This property says that either the `first` object is not defined or the `last` object is reached from `first` via the `next` field.

5 Conclusions

In this paper we introduced pushdown system specifications (PSS) with an associated invariant model checking algorithm **Apost*gen**. We showed why the $\mathbb{K}$ framework is a suitable environment for pushdown systems specifications, but not for their verification via the for-free model checking capabilities available in $\mathbb{K}$. We gave a $\mathbb{K}$ specification of invariant model checking for pushdown system specifications, kA_{post^*}, which is behaviorally equivalent with **Apost*gen**. To the best of our knowledge, no other model checking tool has the flexibility of having structured atomic propositions and working with the generation of the state space on-the-fly.

Future work includes the study of the correctness of our translation of Shylock into the $\mathbb{K}$ framework as well as of the translation of the proposed model checking algorithm and its generalization to any LTL formula. From a more practical point of view, future applications of pushdown system specifications could be found in semantics-based transformation of real programming languages like C or Java or in benchmark-based comparisons with existing model-based approaches for program verification.

Acknowledgments. We would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. Bouajjani, A., Esparza, J., Maler, O.: Reachability Analysis of Pushdown Automata: Application to Model Checking. In: Mazurkiewicz, A.W., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150, Springer (1997)
2. Bouajjani, A., Fratani, S., Qadeer, S.: Context-Bounded Analysis of Multithreaded Programs with Dynamic Linked Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 207–220, Springer (2007)
3. Bonsangue, M., Caltais, G., Goriac, E., Lucanu, D., Rutten, J., Silva, A.: A Decision Procedure for Bisimilarity of Generalized Regular Expressions. In: Davies, J., Silva, L., da Silva Simão, A. (eds.) SBM 2010. LNCS, vol. 6527, pp. 226–241, Springer (2010)
4. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL Model Checker. Electr. Notes Theor. Comput. Sci. 71, 162–187 (2002)
5. Ellison, C., Roşu, G.: An Executable Formal Semantics of C with Applications. In: Field, J., Hicks, M. (eds.) POPL 2012, pp. 533–544, ACM (2012)
6. Esparza, J., Schwoon, S.: A BDD-Based Model Checker for Recursive Programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336, Springer (2001)
7. Goguen, J., Lin, K., Roşu, G.: Circular Coinductive Rewriting. In: ASE 2000, pp. 123–132. IEEE (2000)
8. Kidd, N., Reps, T., Melski, D., Lal, A.: WPDS++: A C++ Library for Weighted Pushdown Systems, http://www.cs.wisc.edu/wpis/wpds++ (2005)
9. Kozen, D.: Kleene Algebra with Tests. ACM Trans. Program. Lang. Syst. 19, 427–443 (1997)
10. Maude, http://maude.cs.uiuc.edu/
11. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational Abstractions. Theor. Comput. Sci. 403(2-3), 239–264 (2008)
12. Meseguer, J., Roşu, G.: The Rewriting Logics Semantics Project. Theor. Comput. Sci. 373(3), 213–237 (2007)
13. Rinetzky, N., Bauer, J., Reps, T.W, Sagiv, S., Wilhelm, R.: A Semantics for Procedure Local Heaps and its Abstractions. In: Palsberg, J., Abadi, M. (eds.) POPL 2005, pp. 296–309, ACM (2005)
14. Roşu, G., Şerbănuţă, T.F.: An Overview of the K Semantic Framework. J. Log. Algebr. Program. 79(6), 397–434 (2010)
15. Roşu, G., Şerbănuţă, T.F.: K-Maude: A Rewriting Based Tool for Semantics of Programming Languages. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp.104–122, Springer (2010)
16. Rot, J., Asavoae, I.M., de Boer, F., Bonsangue, M., Lucanu, D.: Interacting via the Heap in the Presence of Recursion. In: Carbone, M., Lanese, I., Silva, A., Sokolova, A. (eds.) ICE 2012. EPTCS, vol. 104, pp. 99-113 (2012)
17. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)