Internal Report 2012-2013-15 August 2013

Universiteit Leiden
Opleiding Informatica

Implementing a Model Checker for
Simple Recursive Languages

Sander van Rijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

1 Introduction

As computers become faster and more memory becomes available, programs continue to grow larger and more complex. To ensure they function as intended, a program is tested with as many input combinations as possible in a certain timeframe. Because of the growing complexity of these programs, efficient testing becomes harder. This is why model checking, which can confirm the *absence* of errors rather than only their *presence*, is gaining popularity. To contribute to the techniques required for model checking programs in popular languages, we present an implementation of the $post^*$-algorithm to generate a reachability analysis of simple recursive programs.

In Section 2 we give some preliminary definitions of the automata, pushdown systems, pushdown system specifications and context free grammars that will be used throughout this paper. The main focus will be on pushdown systems and pushdown system specifications as our intended model of programs.

Section 3 will discuss the $post^*$-algorithm. After a general description of its use in reachability analysis, we first apply this algorithm to a pushdown system and give an example of how it works. Then we proceed to adapt this algorithm to work with the more abstract pushdown system specifications, and illustrate how this allows us to perform the intended reachability analysis on programs in a simple recursive language.

The programming language we used for our implementation, *Maude*, will be discussed in Section 4. As it is much less common than for example C++ and uses a different programming paradigm, we give a short introduction to the language. Following our motivation for using this language, we discuss some design choices that were made and highlight some aspects of the implementation we found interesting.

2 Preliminaries

In this section we will present formal definitions of the automata and grammars we will be using in Section 3. The definitions have been kept as similar as possible to those in [1, 3].

2.1 Finite Automata

Finite automata are basic models of computation, used as language acceptors.

Definition 1 (*Finite Automaton*)**.**
A finite automaton $\mathcal{A}$ is a quintuple $(Q, \Gamma, \rightarrow, P, F)$, where Q is a finite set of states, Γ is the alphabet, $\rightarrow \subseteq (Q \times \Gamma \times Q)$ is a finite set of transitions, $P \subseteq Q$ is a finite set of initial states and $F \subseteq Q$ is a finite set of accepting states.

We write $q \xrightarrow{\gamma} q'$ if $(q, \gamma, q') \in \rightarrow$. If $p \xrightarrow{\omega}{}^{*} q \in \mathcal{A}$, we say that the automaton $\mathcal{A}$ *accepts* the string $\omega \in \Gamma^*$.

2.2 Pushdown Systems

To support recursion, we have to add a stack to our finite automaton, giving us a pushdown automaton. A pushdown system however, does have the stack of a pushdown automaton, but does not have the input alphabet or the final states. A transition of a pushdown system can only view and use the element at the top of the stack.

Definition 2 (*Pushdown System*)**.**
A pushdown system $\mathcal{P}$ is a quadruple (P, Γ, Δ, c_0), where P is a finite set of states, Γ is the stack alphabet, $\Delta \subseteq (P \times \Gamma \times P \times \Gamma^*)$ is a finite set of rules, and c_0 is the initial configuration. We write $\langle q, \gamma \rangle \hookrightarrow \langle p', w \rangle$ if $(p, \gamma, p', w) \in \Delta$.

We limit the pushdown systems we look at to those which satisfy $|w| \leq 2$ for every rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \omega \rangle$. Specifically this means the pushdown systems we examine will have only three kinds of rules:

- $\langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle$ popping an element of the stack,
- $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ replacing the top element of the stack.
- $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ (replacing and) pushing a new element on the stack,

For any pushdown system that does not satisfy our limitation, an equivalent pushdown system can be constructed that does satisfy the restriction, as stated in [1].

2.3 Context Free Grammars

The simple (programming) languages we want to use in section 3 are *recursive* languages. These are strictly more powerful than non-recursive languages, so we need a more powerful method of defining and modelling them than with finite automata. First we will define the syntax of our programming language using a *context free grammar* (CFG).

Definition 3 (*Context Free Grammar*)**.**
A context free grammar $\mathcal{R}$ is a quadruple (V, Σ, R, B), where V is a finite set of non-terminals or variables, Σ is a finite set of terminals that make up the alphabet of our language, $R \subseteq V \times (V \cup \Sigma)^*$ is a finite set of productions and B is the start variable.

For use throughout this paper, we now define our simple recursive language using a CFG $\mathcal{R}$. The production rules of $\mathcal{R}$ are:

$B \to A \mid (B;B) \mid (B+B) \mid X$
Where A is an action ranging over $a, b, \ldots, z$ and X is a procedure call ranging over $x_0, x_1, \ldots, x_n$.
Where B symbolizes the body of a procedure, $\alpha \in A$ are executable actions and $x_i \in X$ are procedure calls. The semantics of the ; and + operators we introduce here will be explained in section 2.4.

Starting with B, this grammar now can now define the body of a single procedure x_i. However, a program in our language will usually consist of multiple procedures, each having their own body. Generating the bodies of these different procedures through repeated application of the rules when starting just with B would require introducing a cycle in the form of $x_i \to B$, leading to infinite sizes for recursive procedures. A program will therefore be defined as a finite set of procedures $\{x_0 \mapsto B_0, x_1 \mapsto B_1, \ldots, x_n \mapsto B_n\}$ where B_i represents the body of procedure x_i.

2.4 Pushdown System Specifications

Now we have defined the syntax of our recursive language in section 2.3, we want to define the semantics of this language. To do this we aim to construct a PDS $\bar{\mathcal{P}}$ with stack alphabet Γ that will be able to symbolically execute any program that can be generated by $\mathcal{R}$. A program will be given to $\bar{\mathcal{P}}$ as the stack in the initial configuration with a trivial state p. The rules of the PDS can then evaluate the program.

Note that in constructing the PDS from the CFG, we have no use for defining different states. Because of this, we only use a single, trivial state, which we will omit in further references.
The first rules we define for $\bar{\mathcal{P}}$ are those that simulate the execution of basic actions. Because an action is no longer relevant to the program once it has been passed, we can simply remove it from the stack using the finite set of rules: $\{\langle\alpha\rangle \hookrightarrow \langle\epsilon\rangle \mid \alpha \in A\}$. These rules are *language-specific* as the finite set of executable actions A will be defined for the entire language, not just a single program.

Next we look at the procedure calls. Just like non-recursive programs, when we encounter a procedure call x_i, we want to replace it with it's body. As mentioned in Section 2.3, the substitutions of procedure calls by their bodies will be what defines a program in our language. The finite set of rules: $\{\langle x_i\rangle \hookrightarrow \langle B_i\rangle\}$ will therefore be *program-specific*. Note that since we can only replace one stack element by one other element, the entire body B_i will be seen as a single element on the stack.

Finally we define the semantics for the operators ; and + that were mentioned in Section 2.3. The ; operator will be used for *sequential composition* and the + operator will indicate *choice*. Again, because of the kinds of rules we restrict ourselves to, $(a;\beta)$ and $(a+\beta)$ will be seen as a single stack element. That is why we need to add rules that get rid of the operators used by the language, and end up with a stack consisting only of elements in $(A \cup X)$. This gives us the following rules for our PDS:

$$\langle\alpha;\beta\rangle \hookrightarrow \langle\alpha\beta\rangle$$
$$\langle\alpha+\beta\rangle \hookrightarrow \langle\alpha\rangle$$

$$\langle \alpha + \beta \rangle \hookrightarrow \langle \beta \rangle$$
$$\forall \alpha, \beta \in \Gamma.$$

With these semantics defined, we encounter a problem. Since α and β represent any elements in the stack alphabet Γ and $(\alpha; \beta)$ & $(\alpha + \beta)$ are also elements of Γ, the pattern is such that a rule for $\alpha; \beta$ and $\alpha + \beta$ must be given for every pair of programs α, β. This would result in an infinite set of rules and is not necessary as the pattern itself is clear from just those general rules. To solve this problem we introduce the *pushdown system specification* [3].

Definition 4 (*Signature*)**.**
A signature Σ consists of a finite set of symbols $\phi_1, \phi_2, \ldots$, each with a fixed arity $ar(\phi_1)$, $ar(\phi_2), \ldots$. Symbols with arity 0 are called constants. The set of terms, denoted by $T_\Sigma(Var)$, is inductively defined from the set of variables Var and the signature Σ. A term s can match term t if there exists a substitution σ such that $\sigma(t) = s$.

Definition 5 (*Pushdown System Specification*)**.**
A pushdown system specification $\bar{\mathcal{P}}$ is a quadruple $(\Xi, \Sigma, Var, \bar{\Delta})$, where Ξ is a finite set of states, Σ is a signature, Var is a finite set of variables, and $\bar{\Delta} \subseteq (\Xi \times T_\Sigma(Var) \times \Xi \times T_\Sigma(Var)^*)$ is a finite set of production rules. Terms in $T_\Sigma(Var)$ define the stack alphabet. We write $\langle p, \phi \rangle \hookrightarrow \langle p', \Phi \rangle$ for a rule in $\bar{\Delta}$, where ϕ is a term in $T_\Sigma(Var)$ and Φ is a finite, possibly empty, sequence of terms in $T_\Sigma(Var)$. When using a trivial state p,

Given the language defined by $\bar{\mathcal{R}}$, $\Sigma = \{;, +, a, b, \ldots, x_0, x_1, \ldots\}$ with $ar(;) = ar(+) = 2$ and $ar(\alpha \in A) = ar(x_i) = 0$. With this we can use variables $s, t \in Var$ to represent any terms in the stack alphabet as can be constructed from Σ.

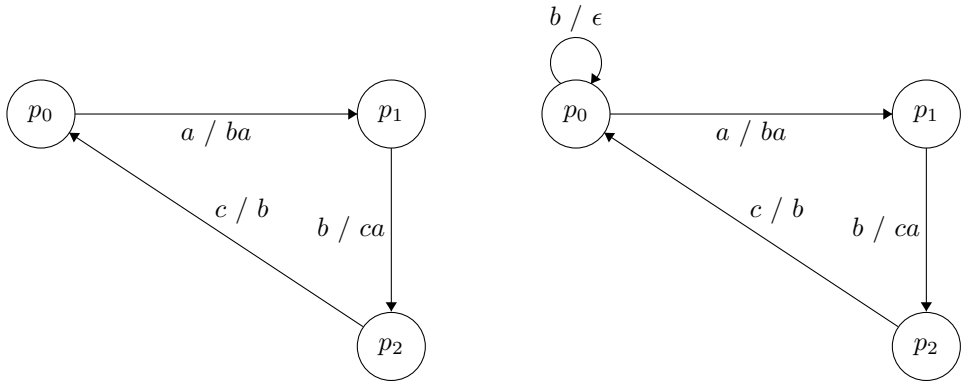
By using variables in production rules, we have avoided the problem of a PDS with an infinite stack alphabet. This leads to the following *finite* set of rules in $\bar{\Delta}$:
$\langle \alpha \rangle \hookrightarrow \langle p, \epsilon \rangle$
$\langle s; t \rangle \hookrightarrow \langle st \rangle$
$\langle s + t \rangle \hookrightarrow \langle s \rangle$
$\langle s + t \rangle \hookrightarrow \langle t \rangle$
$\langle x_i \rangle \hookrightarrow \langle B_i \rangle$
Where $\alpha \in A$, and $s, t \in Var$.

The definition of a PSS can be extended to also use a signature for the set of states (as has been done in [3]), but given our single trivial state, it has been omitted in this paper.

3 Constructing Reachability Automata

In this section we will discuss the reachability algorithm $post^*$. Reachability by itself is not complete model checking, but it is an important first step which can be expanded. We will be specifically looking at reachability of pushdown systems and pushdown system specifications as discussed in section 2. First we will discuss $post^*$ in general, followed by the algorithm for generating a finite automaton accepting $post^*_{\mathcal{P}}(C)$, given a PDS $\mathcal{P}$ with its set of initial configurations C as input [1]. Then we expand the $post^*$ algorithm to accept a PSS rather than a PDS [3].

As explained in Section 2.2, a pushdown system effectively consists of a number of states, a stack with stack alphabet, rules between these states that can modify the top element of the stack and an initial configuration. From a model checking point of view, the obvious question then is: *"which configurations can be reached?"*. We could look at configurations which are *predecessors* of our initial configuration c_0, also denoted as $pre^*(c_0)$. In this paper



(a) $\mathcal{P}$ (b) $\mathcal{P}'$

Figure 1: Pushdown systems

however we will focus on which configurations are *successors* of our initial configuration, denoted as $post^*(c_0)$.

3.1 Post*

To make the concept of successor configurations more precise, we give a formal definition of $post^*$.

Definition 6 ($post^*$)**.**
Given a PDS $\mathcal{P}$ with its set of initial configurations C, $post^*(C) = \{c \mid \exists c' \in C : c' \hookrightarrow^* c\}$ is the set containing all successors of C.

As these concepts are best explained with an example, we will determine $post^*(C_\mathcal{P})$ of the PDS $\mathcal{P}$ in Figure 1 [1]. For the set of initial configurations C, we use $\{\langle p_0, aa\rangle\}$. In this simple example, we can apply each rule of $\mathcal{P}$ exactly once, adding three configurations to our initial set C, leading to $post^*(C_\mathcal{P} = \{\langle p_0, aa\rangle\}) = \{\langle p_0, aa\rangle, \langle p_1, baa\rangle, \langle p_2, caaa\rangle, \langle p_0, baaa\rangle\}$.

3.2 Generating $\mathcal{A}$-$post^*$ for Pushdown Systems

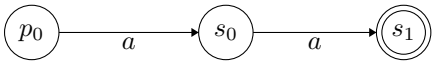
The easy example in Figure 1a leads to a finite set of configurations, but by adding a rule $\langle p_0, b\rangle \hookrightarrow \langle p_0, \epsilon\rangle$ to $\mathcal{P}$ to form a new PDS $\mathcal{P}'$ (Figure 1b) we would get an infinite set of configurations:
$post^*(C_{\mathcal{P}'} = \{\langle p_0, aa\rangle\}) = \{\langle p_0, aa\rangle, \langle p_1, baa\rangle, \langle p_2, caaa\rangle, \langle p_0, baaa\rangle, \langle p_0, aaa\rangle, \langle p_1, baaa\rangle, \ldots\}$

When looking at the set of configurations reachable from $\langle p_0, aa\rangle$ in $\mathcal{P}'$, a clear pattern can be seen, which suggests that $post^*(C_{\mathcal{P}'})$ is regular and can therefore be accepted by a *finite automaton*. To obtain the finite automaton $\mathcal{A}$ accepting $post^*(C_{\mathcal{P}'})$, we use the process as described in [1].

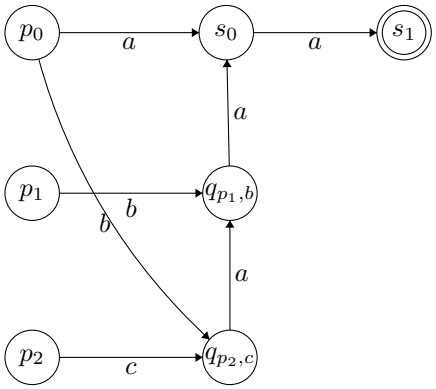
The input for this process is a pushdown system $\mathcal{P}$ and an automaton $\mathcal{A}_0$ accepting $C_\mathcal{P}$. Without loss of generality, we assume that $\mathcal{A}_0$ has no transition leading to an initial state. We compute $post^*(C)$ as the language accepted by an automaton $\mathcal{A}_{post^*}$ with ϵ-moves. $\mathcal{A}_{post^*}$ is obtained from $\mathcal{A}_0$ in two stages:

- For each pair (p', γ') such that $\mathcal{P}$ contains at least one rule of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$, add a new state $q_{p',\gamma'}$
- Add new transitions to $\mathcal{A}$ according to the following saturation rules:

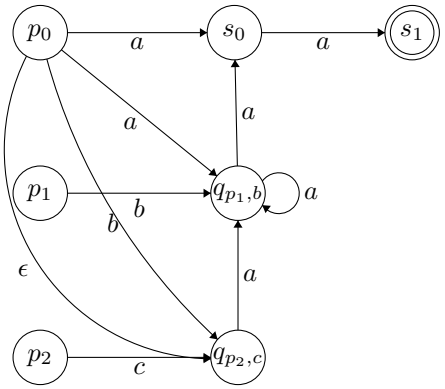
  (i) If $\langle p, \gamma\rangle \hookrightarrow \langle p', \epsilon\rangle \in \Delta_\mathcal{P}$ and $p \xrightarrow{\gamma}{}^* q$ in the current automaton, add a transition (p', ϵ, q)



(a) $\mathcal{A}_0$ accepting $C = \{\langle p_0, aa\rangle\}$ of PDS $\mathcal{P}$ (Figure 1)



(b) $\mathcal{A}$-$post^*(C)$ of PDS $\mathcal{P}$ (Figure 1a)



(c) $\mathcal{A}'$-$post^*(C)$ of PDS $\mathcal{P}'$ (Figure 1b)

Figure 2: Corresponding reachability automata for Figure 1

(ii) If $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \in \Delta_{\mathcal{P}}$ and $p \xrightarrow{\gamma}{}^{*} q$ in the current automaton, add a transition (p', γ', q)

(iii) If $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle \in \Delta_{\mathcal{P}}$ and $p \xrightarrow{\gamma}{}^{*} q$ in the current automaton, add the transitions $(p', \gamma', q_{p',\gamma'})$ and $(q_{p',\gamma'}, \gamma'', q)$

The set $post^*(C_{\mathcal{P}})$ is regular if it is recognised by our finite automaton $\mathcal{A}$ generated using these saturation rules. Because the correctness of these rules has been proven in [1], $post^*(C_{\mathcal{P}})$ is recognised by $\mathcal{A}_{\mathcal{P}}$ and is therefore regular.

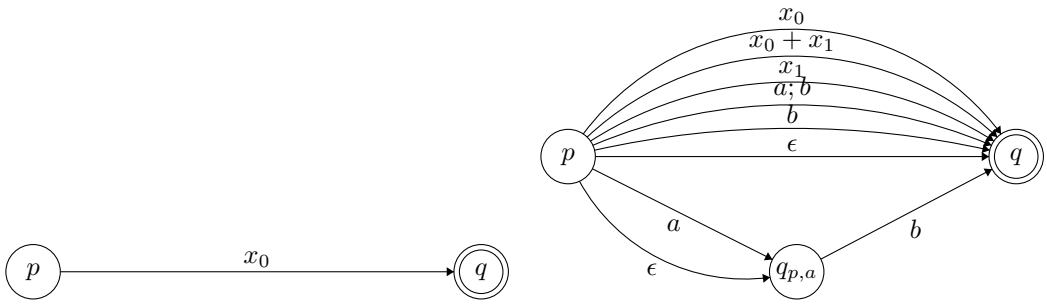
Manually applying these saturation rules requires constantly looking at both the PDS $\mathcal{P}$ and the intended final automaton $\mathcal{A}$, making it a difficult and tedious process. The general idea of these rules is as follows: If from a state $p \in \mathcal{A}$ we can move to another state $q \in \mathcal{A}$ by accepting stack element γ, then anything that $\mathcal{P}$ can replace γ with, must also be accepted by $\mathcal{A}$, ending up in the same state q.

As an example of the application of these rules, Figure 2a shows the initial automaton $\mathcal{A}_0$ that is used as input for generating $\mathcal{A}$ and $\mathcal{A}'$. Note that this automaton has additional states s_0, s_1 that do not occur in $\mathcal{P}$. To this automaton, states p_1, p_2 are added as additional initial states since there are rules in $\mathcal{P}$ leading to them.

For the first stage of the process, states $q_{p_1,b}$ and $q_{p_2,c}$ are added because of the rules $\langle p_0, a\rangle \hookrightarrow \langle p_1, ba\rangle$ and $\langle p_1, b\rangle \hookrightarrow \langle p_2, ca\rangle$ respectively. These states will serve as *"stepping stones"* for dealing with rules that push additional elements on the stack.

As our next steps towards completing $\mathcal{A}$, we add the transitions from p_1, via $q_{p_1,b}$ to s_0 because of $\langle p_0, a\rangle \hookrightarrow \langle p_1, ba\rangle$. Then we add the transitions from p_2, via $q_{p_2,c}$ to $q_{p_1,b}$ because of $\langle p_1, b\rangle \hookrightarrow \langle p_1, ca\rangle$. Finally we add the transition $(p_0, b, q_{p_2,c})$ for the rule $\langle p_2, c\rangle \hookrightarrow \langle p_0, b\rangle$. This concluudes the construction of $\mathcal{A}$, accepting precisely the four configurations listed in Section 3.1.

The addition of the single rule $\langle p_0, b\rangle \hookrightarrow \langle p_0, \epsilon\rangle$ to $\mathcal{P}'$ adds not one, but three new transitions to $\mathcal{A}'$. First, $(p_0, \epsilon, q_{p_2,c})$ is added, simply replacing b with ϵ. However, since ϵa can be accepted in a single transition by a finite automaton, we also add $(p_0, a, q_{p_1,b})$, allowing us to skip the ϵ-transition. Because we now have a new transition accepting a from p_0, we apply



(a) Automaton $\bar{\mathcal{A}}_0$ accepting $C = \langle p, x_0 \rangle$ of $\bar{\mathcal{P}}$

(b) $\bar{\mathcal{A}}$-$post^*(C)$ of $\bar{\mathcal{P}}$

Figure 3: $\bar{\mathcal{A}}_0$ and $\bar{\mathcal{A}}$ for a PSS $\bar{\mathcal{P}}$ with input $\{(x_0 \hookrightarrow (x_0 + x_1)), (x_1 \hookrightarrow (a; b))\}$

our saturation rule again. The transitions that should be added due to $\langle p_0, a \rangle \hookrightarrow \langle p_1, ba \rangle$ are $(p_1, b, q_{p_1,b})$ and $(q_{p_1,b}, a, q_{p_1,b})$. As the first of these already exists, only the a-loop at $q_{p_1,b}$ is visibly added. This completes $\mathcal{A}'$.

3.3 Generating $\mathcal{A}$-$post^*$ for Pushdown System Specifications

With respect to the simple recursive language we defined in Section 2.3, the method for generating reachability automata is lacking the capability to handle the potentially infinite stack alphabet. This problem is similar to the one we encountered in Section 2.4, so we will now adapt the method described in Section 3.2 to work with *pushdown system specifications*.

We want to use rules with variables from our PSS $\bar{\mathcal{P}}$. However, the method described above cannot make the exact matches required to obtain $\bar{\mathcal{A}}$, since the variables used in $\bar{\mathcal{P}}$ do not occur in $\bar{\mathcal{A}}$. To solve this issue, we change the following two aspects of the generation algorithm:

1. In all saturation rules, we no longer require that a production rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \Gamma^* \rangle$ (where Γ^* is a string of 0, 1 or 2 stack elements), *exists* in $\bar{\mathcal{P}}$. Instead, we require such a rule to be *derivable* from $\bar{\mathcal{P}}$.

2. For pairs (p', γ') such that a rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \gamma'' \rangle$ exists, the state $q_{p',\gamma'}$ will be added to $\bar{\mathcal{A}}$ while adding the transitions according to saturation rule (iii).

The first of these changes is a direct result of $\bar{\mathcal{P}}$'s production rules containing variables which cannot occur in $\bar{\mathcal{A}}$. The reason we no longer add all these *"stepping stone"*-states before applying the saturation rules, is because we can't. Unless we define a specific PSS with no variables in production rules that push elements on the stack, the set of these rules is unknown at the start of the algorithm.

Recall the example PSS defined in Section 2.4. As a program in our language, we define the following functions, which are added to the PSS as production rules with a trivial state p: $\{(x_0 \hookrightarrow (x_0 + x_1)), (x_1 \hookrightarrow (a; b))\}$. Starting in the initial configuration $\langle p, x_0 \rangle$, this program can choose to loop back to x_0 recursively, or execute actions a and b before terminating. Figure 3a shows the input automaton $\bar{\mathcal{A}}_0$ for the $post^*$-generation algorithm.

To obtain the final reachability automaton $\bar{\mathcal{A}}$ as shown in Figure 3b, we first add a transition accepting the body of procedure x_0: $(p, (x_0 + x_1), q)$. This new transition is split up into two new transitions, (p, x_0, q) and (p, x_1, q), of which the first one was already in our automaton. With the call for procedure x_1 now free as a single stack element, we can replace it with its body: $(p, (a; b), q)$. The ;-bound actions a and b are now still represented as a single stack element.

```
fmod STACK is

  sorts Element Stack .
  subsort Element < Stack .

  op epsilon   :               -> Element .
  op __        : Element Stack -> Stack [assoc id: epsilon] .
  op pop_from_ : Element Stack -> Stack .
  op push_on_  : Element Stack -> Stack .

  vars A B C   : Element .
  var  S       : Stack

  eq pop  A from A S = S .
  eq push A on   S   = A S .

endfm
```

Figure 4: A simple implementation of a stack in *Maude*

Before we apply the appropriate separation rule, note how the production rule $\langle p, (a; b)\rangle \hookrightarrow \langle p, ab\rangle$ can now be derived from $\bar{\mathcal{P}}$. This calls for the introduction of a new state $q_{p,a}$ and the transitions $(p, a, q_{p,a}), (q_{p,a}, b, q)$ to and from it. The action a from the first of these transitions can be executed and replaced by the empty element ϵ: $(p, \epsilon, q_{p,a})$. This new transition can then be combined with the $(q_{p,a}, b, q)$ transition, skipping $q_{p,a}$ to add (p, b, q). Finally, this action b can then also be executed, adding (p, ϵ, q).

The final automaton $\bar{\mathcal{A}}$ is fairly crowded with its 6 transitions between the same two states, but there is useful information to gain from it. Most notably, the final transition (p, ϵ, q) indicates that, starting with a stack containing only x_0, an empty stack can be reached. Since every stack element is analog to an action, a procedure call or a procedure body, this means that an execution exists that will cause the program to terminate.

4 Maude

In this section we introduce the programming language *Maude* [2] which was used for implementing the algorithm described in Section 3.3. After some general information about Maude as a programming language, we highlight what makes it so useful for implementing this algorithm. Finally we discuss the several aspects of the actual implementation.

4.1 What is Maude?

Maude is different from most popular, high-end imperative languages like C++ or Java. Instead, Maude implements a rewrite logic in which the programmer does not specify what *has* to happen and when, but instead defines what *can* happen in an algebraic model using equations and rewrite rules. In this sense it is most comparable with declarative languages such as *Prolog*. These equations and rules make up modules that can be seen as a model on their own, or can be combined to form more extensive definitions. Although this not intended to be an exhaustive tutorial, we will give a few examples of how Maude works.

Everything in Maude is defined using so-called *sorts*. Figure 4 shows how a stack can be defined using the sorts *Element* and *Stack*. By declaring *Element* a *subsort* of *Stack*, we say that everything that can be done with a *Stack*, can also be done to an *Element*.

To define the syntax in which these sorts are used, we declare *operators*, abbreviated to *op*. In these declarations, an underscore "_" shows where we expect the arguments. This gives us the freedom to make our operators use prefix, postfix or mixfix notation. The sorts of these arguments are then defined after the colon, followed by the sort we want it to be after the arrow. The operator *epsilon* is a constant of the sort *Element*. Using flags, we can add attributes such as associativity, commutativity and an identity relation to an operator such as concatenation of an element onto a stack: "_ _". Apart from a few reserved symbols, we can make anything into an operator, allowing for very clear definitions with little commentary needed.

For the semantics behind our operators, we use *equations*, or *eq* for short. In these equations we use the variables declared by the keyword *var* or *vars*. Unlike languages like Java, we do not assign values to these variables, but use them as placeholders for potential input. The two parts of an equation can usually be substituted equally, but Maude always aims to go to a simpler form. Maude does this by assuming the *right-hand side* is always simpler and will always replace the left-hand side of an equation by the right-hand side, never the inverse. From our example in Figure 4, `pop (A) from (A B)` will therefore always be replaced by the stack `B`, instead of the arbitrary but still equal `pop (C) from (pop (A) from (C A B))`. By writing equations with this in mind, we can steer Maude towards terminating rather than constantly performing useless rewrites.

With this module in place, we can include it in anything we would like to use a stack. If we want a stack of some sort *Object*, we only have to make it a subsort of *Element* and everything else is handled by Maude.

4.2 Motivation for using Maude

To make the algorithm as described in Section 3 work in an imperative language, we would have to write a lot of code to be able to deal with the algebraic nature of our problem. The actual algorithm would then require a lot of while-loops, if-statements and additional lists and variables to keep track of where we are in the execution.

By writing the algorithm in a way that uses recursion and pattern matching, we can stay close to the original algebraic definitions. Maude then takes care of the rest, making the coding process easier and the code much more readable. The resulting code will also be much shorter than that of an equivalent implementation written in an imperative language. This makes the code much easier to work with for others. Overall, for these types of problems, Maude is simply much more convenient to work with than most other options.

4.3 Final Implementation

Because modularity and extensibility were a large focus while writing this code, it has been separated into several modules and files. Each of these modules defines a distinct part of the code, and earlier modules included where possible. The final program consists of the following modules:

- *PRELIMINARIES*, containing the basic definitions of a stack, states and "links" as abstract sorts to simplify the syntax of rules and transitions.
- *AUTOMATON*, adding the definition of transitions to complete the requirements for modelling finite automata
- *PDS*, adding the definition of rules to complete the requirements for modelling pushdown systems.

Input procedure definitions	Hardcoded method	Modular method
$x_0 \to a$	57	89
$x_0 \to x_1$	31	31
$x_0 \to (a; b)$	335	447
$x_0 \to (a + b)$	152	404
$(x_0 \to x_1), (x_1 \to (x_0 + a))$	171	356
$(x_0 \to x_1), (x_1 \to (a; x_0))$	176	292
$(x_0 \to (x_1 + x_2)), (x_1 \to (a; b)), (x_2 \to (x_0))$	731	1186
$(x_0 \to (x_1 + x_2)), (x_1 \to (x_0; x_2)), (x_2 \to (a))$	1550	2473
$(x_0 \to (x_1; x_2)), (x_1 \to ((a; b) + c)), (x_2 \to (x_0 + c))$	1835	2785

Figure 5: Experimental comparison between the number of rewrites required to process a given input when using a hardcoded and modular approach of implementing $post^*$-generation for PSS. The initial configuration is always $\langle p, x_0 \rangle$.

- *POSTSTAR*, defining the input syntax and adding the saturation rules from [1].
- *PSS*, defining the language-specific rules of our pushdown system specification.
- *PSS-POSTSTAR*, adding the equations that can process the PSS-specific syntax into general PDS syntax, so the equations in the *POSTSTAR* module can be matched.

With the code separated into different files, each of the files can be treated as a black box for anyone wanting to work with them. Especially the language-specific rules of our pushdown system specifications have been placed in a separate file. To define a new language, the new syntax and semantics have to be defined in this module without any need to change the rest of the program.

The reason for module *AUTOMATON*'s emptiness is that almost all of its definitions are reused by the the *PDS* module. Those sorts and operators are all defined in the *PRELIMINARIES* module because of that.

With the focus of this implemention on generating the reachability automata for recursive languages, we defined a sort *Rule* and a sort *InRule* (input-rule) in the module *PDS* to represent the production rules of a our PSS $\mathcal{P}$ with only one trivial state. The *InRule* is expanded by Maude to a *Rule* with the trivial state p. This means that an input rule such as $(x0 \to x1)$ will be expanded to $((p, x0) \to (p, x1))$ automatically, making the input easier to write and read.

In the *POSTSTAR* module, the three saturation rules described in Section 3.2 have been translated to three equations. The equations don't exactly match the saturation rules. The rules that pop or replace elements from the stack are handled by just one equation, since popping an element is equivalent to replacing with ϵ. However, because the equations can only look at one transition per rewrite, we need an additional equation that actively merges any ϵ-transitions that occur.

The modules *PSS* and *PSS-POSTSTAR* containing the equations expanding *POSTSTAR* require some further explanation. A clear trade-off has to be made between modularity and execution speed. The fastest way is to write equations that incorporate knowledge of the saturation rules and semantics of the PSS. These "hardcoded" equations are not as easy to read or maintain, but directly produce the transitions for the reachability automaton, making no sacrifice in terms of speed.

For the PSS to be declared in a separate module with a syntax more friendly to the user, a generalized substitution function is required. These equations explicitly derive the rule a regular PDS would need, causing in a 2-step process to generate the transitions for the

automaton $\mathcal{A}$. The difference in the number of rewrites required for several inputs can be seen in Figure 5. For documentation purposes, both implementations have been left in the code. If speed is an absolute requirement, the existing hardcoded equations can be used directly for this language or adapted to work for a different syntax.

5 Discussion and Future Work

In this paper, we have seen that when we define simple recursive languages algebraically using CFG's and pushdown system specifications, there exists an algorithm to generate a reachability automaton of it [1, 3]. Furthermore, we have shown that this algorithm can be almost directly translated into a working program using the language Maude [2], because of its algebraic paradigm. Although at first difficult to work with because of this different paradigm, it has ultimately proven very efficient in use. A clear view of Maude's syntax is helpful when programming, so to improve on this efficiency even further while writing the implementation, a syntax highlighting file for the Kate editor [4] has been written.

Our implementation of the $post^*$-algorithm provides an easy method for generating reachability automata of pushdown system specifications. It can easily be adapted to work for other languages using a different syntax because of its modular structure. The separation of these modules into different files ensures that the algorithm itself can be used as a black box, requiring only a valid PSS definition and input.

The generated automata may be limited in their use, but already determine if a program can terminate by the existance of a transition $p \xrightarrow{\epsilon} q$. However, this implementation is no full model checker. Further logic will have to be added to the generated reachability automata before anything else can be said of the programs that were examined. After all, we now only accept all possible executions of a program, but do not know anything about which will actually be executed.

Finally, now an implementation exists to work directly with pushdown system specifications, it will be interesting to conduct some experiments between this algorithm and first generating a PDS from a PSS to use with the original $post^*$-algorithm from [1].

6 Appendix

6.1 Maude Code

File: poststar.maude

```
***(
***  "Implementing a Model Checker for Simple Recursive Languages"
***
***  Contains definitions for Finite Automata, pushdown systems (PDS),
***  and an implementation of the post* algorithm for PDS.
***
***  Written by  : Sander van Rijn (svr003@gmail.com)
***  Supervisors : Marcello Bonsangue & Jurriaan Rot
***)

fmod PRELIMINARIES is

    protecting INT .

    sorts Element Stack .
    subsort Element < Stack .

    sorts State InitState Config .
    subsort InitState < State .

    sorts BaseLink Link Links .
    subsort BaseLink < Link < Links .

    op epsilon      : -> Element [ctor] .
    op __           : Stack Stack -> Stack [ctor assoc id: epsilon] .
    op l_           : Stack -> Int .

    op _&_          : InitState Element -> State .
    op _,_          : State Stack -> Config .

    op nil          : -> BaseLink [ctor] .
    op _/_          : Links Links -> Links [ctor assoc comm id: nil].
    op Exist_In_    : Links Links -> Bool .
    op NotExist_In_ : Links Links -> Bool .

    var g           : Element .
    var S           : Stack .

    vars l0 l1 l    : Link .
    vars L0 L1 L    : Links .

    eq  l(epsilon) = 0 .
    ceq l(g S)     = 1 + l(S) if g =/= epsilon .

    *** Idempotency of Links.
    eq l / l = l .
```

```
47
48      eq  Exist (l)  In (l / L) = true .
49      ceq Exist (l) In (L) = false
50          if L == nil .
51      ceq Exist (l)  In (l) = false
52          if l == nil .
53      ceq Exist (l0)  In (l1 / L) = Exist (l0) In (L)
54          if l0 =/= l1 /\ l1 =/= nil .
55      ceq Exist (l / L0) In (L1) =
56          Exist (l) In (L1) and Exist (L0) In (L1)
57          if
58          l =/= nil /\ L0 =/= nil .
59
60      eq  NotExist (L0) In (L1) = not (Exist (L0) In (L1)) .
61
62  endfm
63
64
65  fmod AUTOMATON is
66
67      protecting PRELIMINARIES .
68
69      *** PDS-automaton A = (Q, Gamma, ->, P, F), where -> : Q x Gamma x Q
70      *** Transitions = ->
71      sorts Transition Transitions .
72      subsort Transition < Transitions .
73      subsort Transition < Link .
74      subsort Transitions < Links .
75
76      op _,_          : Config State -> Transition .
77
78  endfm
79
80
81  fmod PDS is
82
83      protecting PRELIMINARIES .
84
85      *** PDS P = (P, Gamma, Delta, c), where Delta : (P x Gamma) -> (P x Gamma*)
86      *** Rules = Delta
87      sorts InRule Rule Rules .
88      subsort Rule < Link .
89      subsort Rules < Links .
90      subsorts InRule < Rule < Rules .
91
92      op p            : -> InitState [ctor] .
93      op _->_         : Config Config -> Rule [ctor] .
94      op _->_         : Element Stack -> InRule [ctor] .
95
96      var g           : Element .
97      var S           : Stack .
98
```

```
99      *** When no state is given, a trivial state 'p' is presumed and added.
100     eq g -> S  = p,g -> p,S .
101
102 endfm
103
104
105 fmod POSTSTAR is
106
107     protecting AUTOMATON .
108     protecting PDS .
109
110     sort Input .
111
112     op {_|_}        : Rules Transitions -> Input .
113
114     vars G0 G1 G2 G : Element .
115     vars Q0 Q1 Q2 Q : State .
116     vars P0 P1 P2 P : InitState .
117     vars R T        : Links .
118
119     *** Basic transition generation
120     ceq { (P0,G0 -> P1,G1) / R | (P0,G0,Q) / T } =
121         { (P0,G0 -> P1,G1) / R | (P1,G1,Q) / (P0,G0,Q) / T }
122         if
123         l(G1) < 2 /\ NotExist (P1,G1,Q) In ((P0,G0,Q) / T) .
124
125     *** When stack increases in size, make a new place and a transition to and
126     *** away from it.
127     *** Note: Of these 2-part transitions, the 2nd can be different
128     ***       when the 1st already exists!!
129     ceq { (P0,G0 -> P1,G1 G2) / R | (P0,G0,Q) / T } =
130         { (P0,G0 -> P1,G1 G2) / R | (P1,G1,(P1 & G1)) / ((P1 & G1),G2,Q) /
131                                     (P0,G0,Q)         / T }
132         if
133         G1 =/= epsilon /\ G2 =/= epsilon /\
134         NotExist ((P1 & G1),G2,Q) In ((P0,G0,Q) / T) .
135
136     *** Elimination/collapse of epsilon-transitions
137     ceq { R | (Q0,epsilon,Q1) / (Q1,G0,Q2) / T } =
138         { R | (Q0,G0,Q2) / (Q0,epsilon,Q1) / (Q1,G0,Q2) / T }
139         if
140         NotExist (Q0,G0,Q2) In ((Q0,epsilon,Q1) / (Q1,G0,Q2) / T) .
141
142 endfm
```

File: PSS.maude

```
1 ***(
2 ***  "Implementing a Model Checker for Simple Recursive Languages"
3 ***
4 ***  Defines the Pushdown System Specification (PSS) as used by PSS-POSTSTAR
5 ***
```

```
6  ***  Written by  : Sander van Rijn (svr003@gmail.com)
7  ***  Supervisors : Marcello Bonsangue & Jurriaan Rot
8  ***)
9
10 fmod PSS is
11
12     protecting PDS .
13     protecting AUTOMATON .
14
15     sorts Action ProcCall Abstract .
16     subsort Action < Abstract .
17     subsort ProcCall < Element .
18     subsort Abstract < Element .
19
20     op x            : -> ProcCall .
21     op __           : ProcCall Int -> Element [prec 20] .
22
23     *** Choice and Sequential Composition
24     *** a + b ; c = (a) + (b;c)
25     op _+_          : Element Element -> Abstract [comm prec 35] .
26     op _;_          : Element Element -> Abstract [assoc prec 25] .
27
28     op sub(_)       : Transition -> InRule .
29
30     var  a          : Action .
31     vars Q Q'       : State .
32     vars G1 G2      : Element .
33
34     *** Definition of PSS rules
35     eq  sub(Q,a,Q')          = (a -> epsilon) .
36     eq  sub(Q,(G1 + G2),Q') = (((G1 + G2) -> G1) / ((G1 + G2) -> G2)) .
37     eq  sub(Q,(G1 ; G2),Q') = ((G1 ; G2) -> G1 G2) .
38
39     ***(
40     *** sadly, commutativity in G1 + G2 does not result in a double rewrite
41     *** by Maude, rendering the sub2(_) function below useless
42     op sub2(_)      : Abstract -> Stack .
43     var  A          : Abstract .
44
45     eq  sub(Q,A,Q') = (A -> sub2(A)) .
46
47     eq  sub2(a) = epsilon .
48     eq  sub2((G1 + G2)) = G1 .
49     eq  sub2((G1 ; G2)) = G1 G2 .
50     ***)
51
52 endfm
```

File: PSS-poststar.maude

```
1  ***(
2  ***  "Implementing a Model Checker for Simple Recursive Languages"
```

```
3  ***
4  ***  Contains the extension of the post* algorithm that accepts a
5  ***  pushdown system specification (PSS).
6  ***
7  ***  Written by  : Sander van Rijn (svr003@gmail.com)
8  ***  Supervisors : Marcello Bonsangue & Jurriaan Rot
9  ***)
10
11 fmod PSS-POSTSTAR is
12
13     protecting PSS .
14     protecting POSTSTAR .
15
16     var  A       : Abstract .
17     vars R T     : Links .
18     vars Q Q' q  : State .
19     vars a b c   : Action .
20
21     *** Modular implementation, sub(_) is defined in module PSS
22     ceq { R | (Q,A,Q') / T }
23         =
24         { R / sub(Q,A,Q') | (Q,A,Q') / T }
25         if
26         NotExist (sub(Q,A,Q')) In (R) .
27
28     ***(
29     *** Hardcoded PSS implementation
30     vars G0 G1 G2 : Element .
31
32     ceq { R | (Q,(G1 + G2),Q') / T } =
33         { R | (Q,(G1 + G2),Q') / (Q,G1,Q') / T }
34         if
35         NotExist(Q,G1,Q') In ((Q,(G1 + G2),Q') / T) .
36
37     ceq { R | (Q,(G1 ; G2),Q') / T } =
38         { R | (Q,(G1 ; G2),Q') / (Q,G1,(Q & G1)) / ((Q & G1),G2,Q') / T }
39         if
40         NotExist((Q & G1),G2,Q') In ((Q,(G1 ; G2),Q') / T) .
41
42     ceq { R | (Q,a,Q') / T } =
43         { R | (Q,a,Q') / (Q,epsilon,Q') / T }
44         if
45         NotExist(Q,epsilon,Q') In ((Q,a,Q') / T) .
46     ***)
47
48 endfm
```

6.2 Maude Syntax Highlighter for Kate

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE language SYSTEM "language.dtd">
```

```
 3
 4 <language name="Maude" section="Sources"
 5           version="1.01" kateversion="3.10.3"
 6           extensions="*.maude"
 7           author="Sander van Rijn - svr003@gmail.com">
 8 <!--
 9     ************************************************************************
10     *** Recognises Maude's basic keywords for operators, equations and rules
11     *** Highlights operator arrows and attributes
12     ************************************************************************
13 -->
14   <highlighting>
15     <list name="modules">
16       <item>fmod</item>
17       <item>mod</item>
18       <item>omod</item>
19       <item>endfm</item>
20       <item>endm</item>
21       <item>endom</item>
22       <item>protecting</item>
23       <item>including</item>
24       <item>extending</item>
25       <item>pr</item>
26       <item>ex</item>
27       <item>inc</item>
28     </list>
29     <list name="sorts">
30       <item>sort</item>
31       <item>sorts</item>
32       <item>subsort</item>
33       <item>subsorts</item>
34     </list>
35     <list name="variables">
36       <item>var</item>
37       <item>vars</item>
38     </list>
39     <list name="operators">
40       <item>op</item>
41       <item>ops</item>
42       <item>mb</item>
43       <item>cmb</item>
44     </list>
45     <list name="attributes">
46       <item>ctor</item>
47       <item>assoc</item>
48       <item>comm</item>
49       <item>id</item>
50       <item>idem</item>
51       <item>iter</item>
52       <item>prec</item>
53       <item>gather</item>
54       <item>memo</item>
```

```
55      </list>
56      <list name="equations">
57        <item>eq</item>
58        <item>ceq</item>
59      </list>
60      <list name="rules">
61        <item>rl</item>
62        <item>crl</item>
63      </list>
64      <contexts>
65        <context name="Normal" attribute="Normal Text" lineEndContext="#stay">
66          <keyword attribute="Modules" context="Modules" String="modules"/>
67          <keyword attribute="Sorts" context="Sorts" String="sorts"/>
68          <keyword attribute="Variables" context="Variables" String="variables"/>
69          <keyword attribute="Operators" context="Operators" String="operators"/>
70          <keyword attribute="Equations" context="Equations" String="equations"/>
71          <keyword attribute="Rules" context="Rules" String="rules"/>
72          <DetectChar attribute="Normal Text" context="Attributes" char="["/>
73          <StringDetect attribute="MLComment" context="MLComment" String="***("/>
74          <StringDetect attribute="Comment" context="Comment" String="***"/>
75        </context>
76        <context name="Modules" attribute="Normal Text" lineEndContext="#pop">
77          <StringDetect attribute="Modules" context="#pop" String="is"/>
78          <StringDetect attribute="Modules" context="#pop" String="end"/>
79          <StringDetect attribute="MLComment" context="MLComment" String="***("/>
80          <StringDetect attribute="Comment" context="Comment" String="***"/>
81        </context>
82        <context name="Sorts" attribute="Normal Text" lineEndContext="#stay">
83          <DetectChar attribute="Normal Text" context="#pop" char="."/>
84          <StringDetect attribute="MLComment" context="MLComment" String="***("/>
85          <StringDetect attribute="Comment" context="Comment" String="***"/>
86        </context>
87        <context name="Variables" attribute="Normal Text" lineEndContext="#stay">
88          <DetectChar attribute="Normal Text" context="#pop" char="."/>
89          <StringDetect attribute="MLComment" context="MLComment" String="***("/>
90          <StringDetect attribute="Comment" context="Comment" String="***"/>
91        </context>
92        <context name="Operators" attribute="Normal Text" lineEndContext="#stay">
93          <DetectChar attribute="Normal Text" context="Pre-Operator" char=":"/>
94          <DetectChar attribute="Normal Text" context="Attributes" char="["/>
95          <DetectChar attribute="Normal Text" context="#pop" char="."/>
96          <StringDetect attribute="MLComment" context="MLComment" String="***("/>
97          <StringDetect attribute="Comment" context="Comment" String="***"/>
98        </context>
99        <context name="Pre-Operator" attribute="Normal Text" lineEndContext="#stay">
100         <Detect2Chars attribute="Operators" context="#pop" char="-" char1=">"/>
101         <StringDetect attribute="MLComment" context="MLComment" String="***("/>
102         <StringDetect attribute="Comment" context="Comment" String="***"/>
103         </context>
104       <context name="Attributes" attribute="Normal Text" lineEndContext="#pop">
105         <keyword attribute="Attributes" context="#stay" String="attributes"/>
106         <DetectChar attribute="Normal Text" context="#pop" char="]"/>
```

```
107         <StringDetect attribute="MLComment" context="MLComment" String="***("/>
108         <StringDetect attribute="Comment" context="Comment" String="***"/>
109       </context>
110       <context name="Equations" attribute="Normal Text" lineEndContext="#stay">
111         <StringDetect attribute="Equations" context="#stay" String="if"/>
112         <DetectChar attribute="Normal Text" context="#pop" char="."/>
113         <StringDetect attribute="MLComment" context="MLComment" String="***("/>
114         <StringDetect attribute="Comment" context="Comment" String="***"/>
115       </context>
116       <context name="Rules" attribute="Normal Text" lineEndContext="#stay">
117         <StringDetect attribute="Rules" context="#stay" String="if"/>
118         <StringDetect attribute="Rules" context="#stay" String="=>"/>
119         <DetectChar attribute="Normal Text" context="#pop" char="."/>
120         <StringDetect attribute="MLComment" context="MLComment" String="***("/>
121         <StringDetect attribute="Comment" context="Comment" String="***"/>
122       </context>
123       <context name="Comment" attribute="Comment" lineEndContext="#pop">
124         <LineContinue attribute="Comment" context="#stay"/>
125       </context>
126       <context name="MLComment" attribute="MLComment" lineEndContext="#stay">
127         <LineContinue attribute="MLComment" context="#stay"/>
128         <StringDetect attribute="MLComment" context="#pop" String="***)"/>
129       </context>
130     </contexts>
131     <itemDatas>
132       <itemData name="Normal Text" defStyleNum="dsNormal"/>
133       <itemData name="Modules"/>
134       <itemData name="Sorts" defStyleNum="dsKeyword"/>
135       <itemData name="Variables" defStyleNum="dsDataType"/>
136       <itemData name="Operators" defStyleNum="dsFunction"/>
137       <itemData name="Attributes"/>
138       <itemData name="Equations"/>
139       <itemData name="Rules"/>
140       <itemData name="Comment" defStyleNum="dsComment"/>
141       <itemData name="MLComment" defStyleNum="dsComment"/>
142       <itemData name="Other" defStyleNum="dsOthers"/>
143       </itemDatas>
144   </highlighting>
145   <general>
146     <comments>
147       <comment name="singleLine" start="***" position="afterwhitespace"/>
148       <comment name="multiline" start="***(" end="***)"/>
149     </comments>
150   </general>
151 </language>
152
153
154 <!--
155 // kate: space-indent on; indent-width 2; replace-tabs on;
156 -->
```

References

[1] Schwoon, S.: *Model-Checking Pushdown Systems*. PhD thesis, Technische Universitt Munchen (2002)

[2] Maude, *http://maude.cs.uiuc.edu/*

[3] I. M. Asavoae, F. de Boer, M. M. Bonsangue, D. Lucano, and J. Rot: *Bounded model checking of recursive programs with pointers in K*. In: Proceedings of WADT 2012, LNCS 7841, pp. 59–76, Springer.

[4] Kate, *http://kate-editor.org/*