Verifying Multithreaded Recursive Programs with Integer Variables

Narjes Ben Rajeb
LIP2 and Institut National des Sciences Appliquées et de Technologie de Tunis
narjes.benrajeb@planet.tn

Brahim Nasraoui
LIP2 and Faculté des Sciences de Tunis
brahim.nasraoui@gmail.com

Riadh Robbana
LIP2 and Polytechnic School of Tunisia
riadh.robbana@fst.rnu.tn

Tayssir Touili
LIAFA, CNRS and University Paris Diderot, France
touili@liafa.jussieu.fr

Abstract
We consider the verification problem of programs containing the following complex features: (1) dynamic creation of parallel threads, (2) synchronisation between parallel threads via global variables, (3) (possibly recursive) procedure calls, and (4) integer variables. The configurations of such programs are represented by terms, and their transitions by term rewriting systems. The novelty of our modeling w.r.t. other existing works consists in explicitly modeling integer variables in the terms. We propose a semi-decision procedure that, in case of termination, checks whether an infinite set of configurations, represented by a regular tree language, is reachable from an infinite set of initial configurations of the program (usually represented by a set of non-ground terms). As far as we know, this is the first time that reachability between non-ground terms and regular tree languages is considered. We implemented our techniques in a tool, and tested it successfully on several examples.

Keywords: Multithreaded programs with procedure calls, integer variables and synchronisation, program analysis, verification, rewriting systems, tree automata.

1 Introduction
Software analysis is nowadays one of the most challenging problems in computer-aided verification. Indeed, programs present several complex features such as: (i)
the manipulation of variables ranging over unbounded domains such as reals and integers; (ii) the manipulation of complex data structures such as tables, lists, etc; (iii) the use of recursive procedures; (iv) the use of primitives such as spawn that allow the dynamic creation of parallel processes; and (v) the synchronisation between parallel processes. Ramalingam [20] showed that as soon as synchronisation and procedure calls are taken into account, the reachability problem (even of a single control point) is undecidable for programs. Due to this undecidability, to analyse software, we need to find general semi-algorithms that are not guaranteed to terminate in general, but that behave well and terminate in the most practical cases [4,18,9].

One of the techniques that has been recently used for program verification is regular tree model checking [4,1]. In this framework, programs are modeled and analyzed using automata-based symbolic representations: the control structures of the program that record the names of the procedures to call, and the sequential/parallel order in which they must be called are encoded as trees (of arbitrary sizes), (infinite) sets of configurations are represented using tree automata, and the instructions of the program as a term rewriting system $R$. Then, verification problems based on performing reachability analysis are reduced to the computation of closures of regular languages under term rewriting systems. More precisely, the problem is to determine whether a (potentially infinite) set of bad configurations $Bad$ can be reached from an initial (potentially infinite) set of program configurations $Init$, where $Init$ and $Bad$ are given as finite state tree automata. This amounts to computing the set of reachable configurations $R^*(Init)$, where $R^*$ is the reflexive-transitive closure of $R$, and checking whether it has a nonempty intersection with $Bad$. Since computing $R^*(Init)$ is impossible in general, the main issue in regular model checking is to find accurate and powerful fixpoint acceleration techniques that help the convergence of the reachability analysis in the general case. There are several works based on regular tree model checking that have been applied to program verification [12,13,5,21,6,22,2].

However, the regular model checking based works mentioned above present two main limitations: (1) The first limitation is that they consider only boolean programs, i.e., they cannot deal with programs with variables such as integers or reals. (2) The second limitation is that regular tree automata cannot express equality constraints between subterms. For example, the set of terms \{$f(t, t)$ for any term $t$\} is not regular. However, as we will see later, for program verification it is important to be able to represent sets with these constraints.

In this work, we tackle these two limitations. The contributions of this paper can be summarized as follows:

(i) We propose a class of rewriting systems that can be used to model multi-threaded recursive programs with integer variables. The idea is to use terms to represent integers (e.g. the term $s(s(s))$ represents the integer 3) as well as the control structure of the program; and term rewriting systems to represent the dynamics of the program.

(ii) We propose a semi-decision procedure that, in case of termination, decides
whether starting from a set of non ground terms (i.e., terms with variables), it is possible to reach a regular tree language \( L \) by a rewriting system \( R \). Non ground terms allow to express equality constraints between subterms. For example, the set \( \{ f(t,t) \text{ for any term } t \} \) mentioned above can be represented by the term \( f(x,x) \), where \( x \) is a variable. As far as we know, this is the first time that reachability between non ground terms and a regular tree language is tackled. Indeed, standard rewriting strategies consider only reachability between two terms, whereas regular model checking deals with reachability between two tree automata. Mixing tree automata and non ground terms has never been considered. This mixture is important in program verification. Indeed, the initial set of configurations of a given program can naturally be represented by e.g. a term of the form \( e_{\text{main}}(a_1(x),a_2(x)) \) to express that the program starts its execution at the entry point of the main procedure with a configuration where the two integer variables \( a_1 \) and \( a_2 \) have the same value (the variable \( x \) represents this value). Note that this term represents an infinite set of configurations because the variable \( x \) represents an infinite number of possible instances. Regular tree automata cannot represent such sets of configurations with equalities between integer variables.

On the other hand, the bad configurations of a program can naturally be represented by tree automata, and non ground terms are not sufficient to represent them. This is due to the fact that usually a bad configuration is a configuration where some bad control point \( n \) is active. Such configurations can be written of the form \( C[n] \), where \( C \) is any execution context of the program. The sets of configurations of this form can be represented by tree automata but not by non ground terms.

(iii) We implemented our techniques in a tool prototype based on MAUDE [10]. We applied our tool to different case studies. We obtained encouraging results. In particular, we were able to find automatically two bugs in two different versions of a Windows NT Bluetooth driver. These bugs were already found in SPADE [17]. The novelty of our approach w.r.t. SPADE is that it allows to model explicitly the integer variables of the program, whereas in SPADE, a pushdown stack was needed to encode the values of the variables.

**Related Work.** Abstract-interpretation techniques [7] have been used to deal with data ranging over unbounded domains. More recently, automated predicate-abstraction techniques [14] have been proposed to deal with this issue [3,16,8]. In contrast to predicate abstraction where the exact value of the variables are abstracted away, our modeling remains precise for integer variables. We can use predicate abstraction to deal with the other non integer variables of the program.

MAUDE [10] is a system that performs reachability between two terms for term rewriting systems with equational theories. It also supports arithmetic operations on natural numbers by considering rewriting modulo associativity, commutativity, and distributivity. Reachability modulo these operations is undecidable and tedious. In our modeling, we do not need to use such properties to deal with integers. On
the other hand, MAUDE does not perform reachability between terms and tree automata.

2 Preliminaries

2.1 Trees and terms

An alphabet $\Sigma$ is ranked if it is endowed with a mapping rank : $\Sigma \to \mathbb{N}$. For $k \geq 0$, $\Sigma_k$ is the set of elements of rank (or arity) $k$. Let $\mathcal{X}$ be a denumerable set of symbols called variables. The set $T_\Sigma[\mathcal{X}]$ of terms over $\Sigma$ and $\mathcal{X}$ is defined inductively as follows:

- if $f \in \Sigma_0$, then $f \in T_\Sigma[\mathcal{X}]$;
- if $x \in \mathcal{X}$, then $x \in T_\Sigma[\mathcal{X}]$;
- if $k \geq 1$, $f \in \Sigma_k$ and $t_1, \ldots, t_k \in T_\Sigma[\mathcal{X}]$, then $f(t_1, \ldots, t_k)$ is in $T_\Sigma[\mathcal{X}]$.

$T_\Sigma$ will stand for $T_\Sigma[\emptyset]$. Terms in $T_\Sigma$ are called ground terms. A term in $T_\Sigma[\mathcal{X}]$ is linear if each variable occurs at most once. A linear term is called a context. A binary symbol $f \in \Sigma_2$ is associative if for any terms $t, t', t''$; $f((f(t, t'), t''))$ is equivalent to $f(t, f(t', t''))$. It is commutative if for any terms $t, t'$; $f(t, t')$ is equivalent to $f(t', t)$. We denote by $\equiv$ the equivalence relation between terms induced by the associativity/commutativity of such symbols. A set of terms $L$ is $\equiv$-compatible iff

$$\text{if } (t \in L \text{ and } t' \equiv t) \text{ then } t' \in L.$$  

A substitution $\sigma$ is a mapping from $\mathcal{X}$ to $T_\Sigma[\mathcal{X}]$, written as $\sigma = \{x_1 \to t_1, \ldots, x_n \to t_n\}$, where $t_i$, $1 \leq i \leq n$, is a term that substitutes the variable $x_i$. The term obtained by applying the substitution $\sigma$ to a term $t$ is written $t \sigma$, and can be denoted by $t[t_1, \ldots, t_n]$. We call it an instance of $t$. A position in a term $t$ is defined as a sequence of positive integers in the following way: Let $t$ be a term; if $t = f(t_1, \ldots, t_k)$, then the symbol $f$ is at position $\epsilon$ (the empty sequence) and each subterm $t_i$ is at position $i$, $1 \leq i \leq k$. Then it goes on by induction: considering $t_i$ as a term, a position $p$ in $t_i$ corresponds to the position $i, p$ in $t$. We denote by $Pos(t)$ the set of all positions of the term $t$. A subterm of $t$ at position $p$ is denoted by $t/p$. If a term $t'$ is obtained from a term $t$ by replacing the subterm $t/p$ by a term $u$, we write $t' = t[p \leftarrow u]$.

A term in $T_\Sigma[\mathcal{X}]$ can be viewed as a rooted labeled tree where an internal node with $n$ sons is labeled by a symbol from $\Sigma_n$, and the leaves are labeled with variables and constants (symbols in $\Sigma_0$).

2.2 Tree automata

Definition 2.1 [[11]] A finite tree automaton is a tuple $\mathcal{A} = (Q, \Sigma, F, \delta)$ where $Q$ is a finite set of states, $\Sigma$ is a ranked alphabet, $F \subseteq Q$ is a set of final states, and $\delta$ is a set of rules of the form (1) $f(q_1, \ldots, q_n) \rightarrow q$, or (2) $a \rightarrow q$, where $a \in \Sigma_0$, $n \geq 0$, $f \in \Sigma_n$, and $q_1, \ldots, q_n, q \in Q$. $\mathcal{A}$ is deterministic if there are no two rules with the same left-hand side.
Let \( \rightarrow_\delta \) be the move relation of \( \mathcal{A} \) defined as follows: Given \( t \) and \( t' \) two terms of \( T_{\Sigma \cup Q} \), \( t \rightarrow_\delta t' \) iff:

- there exist a context \( C \in T_{\Sigma \cup Q}[\mathcal{X}] \), \( n \) ground terms \( t_1, \ldots, t_n \in T_\Sigma \), and a rule \( f(q_1, \ldots, q_n) \rightarrow q \) in \( \delta \), such that \( t = C[f(q_1, \ldots, q_n)] \) and \( t' = C[q] \);
- or, there exist a context \( C \in T_{\Sigma \cup Q}[\mathcal{X}] \) and a rule \( a \rightarrow q \) in \( \delta \), such that \( t = C[a] \) and \( t' = C[q] \).

Let \( \rightarrow^*_\delta \) be the reflexive-transitive closure of \( \rightarrow_\delta \). A term \( t \) is accepted by a state \( q \in Q \) iff \( t \rightarrow^*_\delta q \). Let \( L_q \) be the set of terms accepted by \( q \). The language accepted by the automaton \( \mathcal{A} \) is \( \mathcal{L}(\mathcal{A}) = \bigcup \{ L_q \mid q \in F \} \). A tree language is regular if it is accepted by a finite tree automaton.

### 2.3 Rewriting Systems

A term rewriting system is a set of rules of the form \( l \rightarrow r \), where \( l \) and \( r \) are terms in \( T_{\Sigma}[\mathcal{X}] \). Let \( R \) be a term rewriting system and \( t \) and \( t' \) be two terms. We write \( t \rightarrow_R t' \) if there exist a position \( p \in Pos(t) \), a rule \( l \rightarrow r \) in \( R \), a substitution \( \sigma \), and two terms \( t_1 \) and \( t_2 \) such that \( t_1 \equiv t, t_2 \equiv t' \), \( t_1/p = l\sigma \) and \( t_2 = t_1[p \leftarrow r\sigma] \).

We say that \( t' \) is a successor of \( t \). More generally, the set of successors of \( t \) is the set of all terms that are obtained after a single rewriting step applied to \( t \):

\[
\text{successors}(t) = \{ t' \in T_{\Sigma}[\mathcal{X}] \mid t \rightarrow_R t' \}.
\]

Let \( \rightarrow^*_R \) be the reflexive-transitive closure of \( \rightarrow_R \). \( t' \) is reachable from \( t \) if \( t \rightarrow^*_R t' \).

### 3 Term to Automaton Reachability

Let us fix a rewriting system \( R \), a (possibly non ground) term \( t \), and a regular tree language \( L, \equiv \)-compatible, given by a deterministic tree automaton \( \mathcal{A} \). We propose in this section a semi-decision procedure that checks whether there exists a term \( t' \in L \) such that \( t' \) is reachable from \( t \) (i.e., there exists a term \( t'' \) and a substitution \( \sigma \) such that \( t \rightarrow^*_R t'' \) and \( t' \equiv t''\sigma \)). The idea of our procedure is to iteratively apply the rules of \( R \) to the successors of \( t \) until either we reach a term equivalent to another term with an instance in \( L \), or we reach a step where all the successors computed so far cannot be rewritten by \( R \). To do so, there are mainly two difficulties:

(i) Since terms contain variables, sometimes we need to instantiate them before being able to apply a rewriting step. For example, if \( t = f(x, y) \) and \( R \) is the rule \( f(g(z), y) \rightarrow g(y) \); then before applying \( R \) to \( t \) to obtain \( g(y) \), we need to instantiate the variable \( x \) by \( g(z) \). This instanciation step is known as narrowing (see [15] for a survey). It has been implemented in tools such as MAUDE [10] to perform reachability between two terms. We use MAUDE in our implementation to perform this narrowing step.

(ii) The second difficulty is to determine whether a given non ground term \( t \) has an equivalent term with a ground instance in the infinite set of terms \( L \). Since \( L \) is \( \equiv \)-compatible, this amounts to checking whether \( t \) has a ground instance
in $L$ (this is the reason why we need $L$ to be $\equiv$-compatible). We show in the next section how to solve this problem.

3.1 Does a non ground term have an instance in a regular language?

The aim of this section is to show how to determine whether a term $t$ has a ground instance in a given regular tree language $L$. This is not obvious because there is an infinite number of ground instantiations of $t$ that need to be checked. We give hereafter a procedure that solves this problem if we are given a deterministic tree automaton $A$ that recognizes $L$. This is not restrictive since any regular language has a deterministic tree automaton that recognizes it. Let then $t$ be a non ground term with $n$ variables $x_1, \ldots, x_n$, and let $A = (Q, \Sigma, F, \delta)$ be a deterministic tree automaton that recognizes $L$. Then:

**Theorem 3.1** $t$ has a ground instance in $L$ iff there exists a sequence of states $q_1, \ldots, q_n$ (the $q_i$'s need not be different) such that $t[q_1, \ldots, q_n]$ has an accepting run in $A$, i.e., there exists $q \in F$ such that $t[q_1, \ldots, q_n] \xrightarrow{*} \delta q$.

**Proof.** Suppose $t$ has a ground instance in $L$, then there exist $n$ ground terms $u_1, \ldots, u_n$, and a final state $q \in F$ such that $t[u_1, \ldots, u_n] \xrightarrow{*} \delta q$. Therefore, there exist states $q_1, \ldots, q_n$ in $Q$ (that are not necessarily different) such that $u_i \xrightarrow{\rightarrow} \delta q_i$ for every $i$, $1 \leq i \leq n$, and $t[u_1, \ldots, u_n] \xrightarrow{*} \delta t[q_1, \ldots, q_n] \xrightarrow{*} \delta q$. Note that in this case, $u_i = u_j$ implies that $q_i = q_j$ because $A$ is deterministic.

Note that the fact that $A$ is deterministic is crucial. To see this, let us consider the example where $t$ is $f(x, x)$, and suppose that $\delta$ consists of the following rules: $a \rightarrow \delta q_1$, $a \rightarrow \delta q_2$, and $f(q_1, q_2) \rightarrow \delta q$. Then, $f(a, a)$ is a ground instance of $t$ that is in $L$ since $f(a, a) \xrightarrow{*} \delta f(q_1, q_2) \rightarrow \delta q$, whereas neither $f(q_1, q_1)$ (obtained by substituting $x$ by $q_1$) nor $f(q_2, q_2)$ (obtained by substituting $x$ by $q_2$) have accepting runs in $A$.

3.2 Our Procedure

Let $T$ be a finite set of (non-ground) terms, $L$ a $\equiv$-compatible regular tree language given by a deterministic tree automaton $A$, and $R$ a term rewriting system. We would like to check whether after applying the rules of $R$ starting from $T$, it is possible to reach a term that has a ground instance in $L$. This problem being undecidable, we propose a semi-decision procedure. The idea is to apply iteratively the rules of $R$ until either we reach a term that has a ground instance in $L$, or we reach a point where all the terms that need to be rewritten cannot be reduced by $R$.

The procedure is given in Figure 1. $X$ is the set of reachable terms whose successors have already been computed. In line (4), we use the algorithm underlying Theorem 3.1. Line (8) corresponds to a rewriting step. In our implementation, the tool MAUDE [10] takes care of such steps. This is the main reason why we built our tool on top of MAUDE. Line (6) also is done by MAUDE.
Theorem 3.2 The above procedure terminates if \( L \) is reachable from \( T \).

4 From a program to a rewriting system

We are interested in the analysis of programs that present the following complex features: (1) dynamic creation of parallel threads, (2) synchronisation between parallel threads via global variables, (3) (possibly recursive) procedure calls, and (4) integer variables. We present in this section our model, and show how it describes such programs.

4.1 Integers as terms

The key idea of our modeling is to use terms to represent integers. Let \( s \) be a symbol of arity 1 or 0. \( \text{Nat} \) is the set of terms defined as follows:

- \( s \in \text{Nat} \);
- if \( t \in \text{Nat} \), then \( s(t) \in \text{Nat} \).

\( \text{Nat} \) defines the set of strictly positive integers. For example, the term \( s \) represents the integer 1, and the term \( s(s(s())) \) represents the integer 4.

Let \( t \) be a term in \( \text{Nat} \), \( C \) be a context such that \( t = C[s] \) and \( x \) be a variable. \( t(x) \) is defined as \( t(x) = C[s(x)] \).

4.2 Configurations as terms

We distinguish between three kinds of variables in programs: visible variables are the variables that are visible to all the threads, local-global variables are the ones that are local to one thread but global to all the procedures of this thread, and local variables are local to one procedure. For simplicity, we assume that all the
procedures of our program have the same local variables, say $a_1, \ldots, a_k$; and that all the threads have the same local-global variables, say $b_1, \ldots, b_m$. Suppose there are $l$ visible variables $c_1, \ldots, c_l$. Let $\Gamma$ be the set of control points of the program. Then the program is modeled by a rewriting system $R$ over the signature $\Sigma = \Gamma \cup \{ \square, \sharp, ||, \odot, s, a_1, \ldots, a_k, b_1, \ldots, b_m, c_1, \ldots, c_l \}$, where each element $n$ of $\Gamma$ is of arity $k$, $s$ can be of arity 1 or 0, $\odot$ and $||$ of arity 2, the $a_i$’s, $b_i$’s, and $c_i$’s are of arity 0 or 1, $\sharp$ of arity $m + 1$, and $\square$ of arity $l + 1$. Moreover, $||$ is associative and commutative ($||$ models parallel composition). Let us explain the intuition behind these symbols. As explained in the previous subsection, “$s$” is the successor operator for integers. We use a subterm of the form $a_i(v)$, where $v \in \text{Nat}$ to express that the variable $a_i$ has the integer value corresponding to $v$. For example, $a_i(s(s(s))))$ means that the variable $a_i$ has value 3. A subterm $a_i$ (i.e., when $v$ is empty) means that $a_i = 0$. We use a subterm of the form $n(a_1(v_1), \ldots, a_k(v_k))$, where the $v_i$’s are in $\text{Nat}$ to express that the program is at control point $n$ with local variables $a_1, \ldots, a_k$ having respectively the values corresponding to $v_1, \ldots, v_k$. Similarly, we use subterms of the form $b_i(v), v \in \text{Nat}$, to express that the local-global variable $b_i$ has value $v$. $\odot$ represents sequential composition, i.e., for two terms $t_1$ and $t_2$, $\odot(t_2, t_1)$ means that $t_1$ is executed first, and that $t_2$ starts its execution when $t_1$ finishes. This operator models the execution stack of the thread, i.e., the order in which the statements are executed. The top of the stack (the right-most task) has to be executed first. We use the “$\sharp$” operator as a root of the subterms corresponding to single (sequential) threads to put together the values of all the local-global variables and the execution stack of the program. For example, suppose a given thread has only two local variables $a_1, a_2$ and two local-global variables $b_1, b_2$. Then the term $\sharp \left( \odot \left( n_2(a_1, a_2(s)), n_1(a_1(s(s)), a_2) \right), b_1, b_2(s) \right)$ represents a configuration where the thread is at control point $n_1$ with local variables $a_1 = 2$ and $a_2 = 0$; and local-global variables $b_1 = 0$ and $b_2 = 1$, and such that when the execution at point $n_1$ finishes, the thread goes back to the calling point $n_2$ with local variables $a_1 = 0$ and $a_2 = 1$.

The parallel composition is modeled by $||$, i.e., $||(t_2, t_1)$ represents the parallel execution of $t_1$ and $t_2$. The operator “$\square$” is used as the root of our terms to put the values of all the visible variables together with the parallel threads of the program. For example, if the program has three visible variables, then the term $\square \left( ||(n(t_1, t_2), t_3), c_1(s), c_2(s(s)), c_3(s) \right)$ represents a configuration where the visible variables $c_1, c_2, c_3$ have respectively the values 1, 2, and 1; and where there are three sequential threads $t_1, t_2$, and $t_3$ running in parallel (these terms have $\sharp$ as root and have the form of sequential thread terms described above).

### 4.3 The instructions as rewriting rules

Each instruction of the program can be modeled by rewriting rules.

- **Assignments:** We can model assignments of the form $x := c$ and $x := x + c$, where $x$ is a variable and $c$ is an integer constant. Suppose that $x$ is a local...
variable $a_i$ and that the assignment is of the form

$$n : a_i := a_i + c$$

where $i$ is in $\{1, \ldots, k\}$ and $n$ is the control point of the program corresponding to this instruction. We suppose that $c \neq 0$. (the other cases can be treated similarly). Let $t_c$ be the term in Nat that represents the value $c$ ($t_c$ is the term $s(s(s))$ if $c$ is 3). Intuitively, this assignment can be represented by the following rule:

$$n(x_1, \ldots, x_{i-1}, a_i(x_i), x_{i+1}, \ldots, x_k) \rightarrow n(x_1, \ldots, x_{i-1}, a_i(t_c(x_i)), x_{i+1}, \ldots, x_k),$$

where $x_1, \ldots, x_k$ are term variables that represent the values of the local variables $a_1, \ldots, a_k$. However, to make sure that this rule cannot be applied anywhere in the term, and that it can be applied only to the points $n$ that are on the top of the stack, we need to represent it by these two rules:

(i)  $$\sharp\left( n(x_1, \ldots, x_{i-1}, a_i(x_i), x_{i+1}, \ldots, x_k), y_1, \ldots, y_m \right) \rightarrow \sharp\left( n(x_1, \ldots, x_{i-1}, a_i(t_c(x_i)), x_{i+1}, \ldots, x_k), y_1, \ldots, y_m \right).$$

This rule is applied when $n$ is the only control point in the stack.

(ii)  $$\sharp\left( \bigcirc (x, n(x_1, \ldots, x_{i-1}, a_i(x_i), x_{i+1}, \ldots, x_k)), y_1, \ldots, y_m \right) \rightarrow \sharp\left( \bigcirc (x, n(x_1, \ldots, x_{i-1}, a_i(t_c(x_i)), x_{i+1}, \ldots, x_k)), y_1, \ldots, y_m \right).$$

In this rule, $n$ is on the top of the stack, and $x$ represents the other content of the stack. This rule ensures that the assignment applies only if the control point $n$ is on the top of the stack.

In these two rules, $y_1, \ldots, y_m$ are term variables that represent the values of the local-global variables $b_1, \ldots, b_m$.

**Procedure calls:** Suppose that at control point $n_1$, there is a call to a procedure $P$, and that the next control point of the program after $P$ returns is $n_2$. Let $e_P$ be the entry point of the procedure $P$. This call is represented by the following rule:  $$\sharp\left( \bigcirc (x, n_1(x_1, \ldots, x_k)), y_1, \ldots, y_m \right) \rightarrow \sharp\left( \bigcirc (x, n_2(x_1, \ldots, x_k), e_P(a_1, \ldots, a_k)), y_1, \ldots, y_m \right).$$

As previously, $x$ represents the rest of the stack; $x_1, \ldots, x_k$ represent the values of the local variables $a_1, \ldots, a_k$ at point $n_1$, and $y_1, \ldots, y_m$ represent the values of the local-global variables $b_1, \ldots, b_m$.

This rule expresses that when the procedure $P$ is called, its entry point $e_P$ is put on the top of the stack and its local variables are initially all set to 0. When this procedure terminates, the control goes back to point $n_2$. There is a similar rule that corresponds to the case where $x$ is empty, i.e., to the case where $n_1$ is the only control point in the stack.

**Dynamic Thread Creation:** Dynamic thread creation can be modeled by
rules of the form
\[ \lbrack\lbrack x, \sharp \circ (x', n(x_1, \ldots, x_k)), y_1, \ldots, y_m \rbrack \rbrack \rightarrow \lbrack\lbrack x, \sharp \circ (x', n_1(x_1, \ldots, x_k)), y_1, \ldots, y_m \rbrack, \sharp(n_2(a_1, \ldots, a_k), b_1, \ldots, b_m) \rbrack. \]

This rule expresses that when a thread is at control point \( n \), it goes to control point \( n_1 \) and launches a new thread at control point \( n_2 \). This new thread has null local and local-global variables. There is a similar rule that corresponds to the case where \( x' \) is empty.

- **if-then-else conditions**: if-then-else conditions are represented by rules as previously, but that are applicable only to terms satisfying the condition (or not; depending on whether we consider the “if” or the “else” branch). We restrict ourselves to tests between variables of the form \( c \sim d \) where \( \sim \in \{<, >, \leq, \geq, =\} \). We use MAUDE to perform these tests.

Note that our model handles synchronisation between parallel processes since (1) synchronisation is performed using visible variables, (2) we can model if-then-else branches where the conditions are on the visible variables accurately, and (3) \( \lbrack\lbrack \rbrack \) is associative and commutative, which allows to test locally the values of the visible variables.

### 4.4 Example

Let us illustrate our modeling using a small toy example. For simplicity, we take the following small part of a program:

\[
\begin{align*}
n & \colon b := 3 \\
n' & \colon \text{Call } P \\
n'' & : \ldots \ldots
\end{align*}
\]

We consider two instances of the following program running in parallel. In this program, there are no local variables, and there is a single local-global variable \( b \). Suppose the two parallel instances share a visible variable \( c \). Initially, the system is in configuration \( \Box \lbrack\lbrack \sharp(n(b), \sharp(n, b)) \rbrack \rbrack \) where the two instances are in control point \( n \), with \( b = 0 \), and \( c = 0 \). If the first component (instance) applies the first instruction, the system moves to \( \Box \lbrack\lbrack \sharp(n', b(s(s(s)))) \rbrack, \sharp(n, b) \rbrack \), where \( e_P \) is the entry point of the procedure \( P \), etc.

### 5 Experimental Results

We implemented our techniques in a tool based on MAUDE [10], and applied it to several examples. Our tool takes a rewrite system modeling a program, a tree
automaton modeling the bad configurations \( L \), and a set of (non-ground) terms \( T \) modeling the initial set of configurations. In case of termination, the tool answers whether \( L \) is reachable from \( T \). The performances are given in Table 1. The experiments were obtained on a 1.6GHz processor with 2GB of memory. The last column gives the number of rewrite steps needed to find the bugs or to prove the correctness of the program. All the programs were analysed in a completely automatic manner. We needed the whole power of our model to handle these programs since they contain dynamic thread creation, synchronisation, integer variables, procedure calls, and if-then-else conditions.

<table>
<thead>
<tr>
<th>Example</th>
<th>Time(ms)</th>
<th>Nbre of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlueTooth v1</td>
<td>15917</td>
<td>84</td>
</tr>
<tr>
<td>BlueTooth v2</td>
<td>17456</td>
<td>143</td>
</tr>
<tr>
<td>Lock/unlock</td>
<td>15509</td>
<td>118</td>
</tr>
<tr>
<td>Philosophers</td>
<td>24501</td>
<td>1250</td>
</tr>
<tr>
<td>Prod/Cons</td>
<td>12767</td>
<td>108</td>
</tr>
<tr>
<td>Toy Example</td>
<td>3274</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 1
Performances of our tool

The BlueTooth v1 is our model of the BlueTooth driver program used by Windows NT and given in [19]. The BlueTooth v2 is a corrected version of BlueTooth v1 proposed by the authors of [19]. We were able to find two bugs in these programs. SPADE [17] found these bugs as well, however, the SPADE model of these versions encodes the values of the integer variables in the stack of a pushdown system, whereas with our new model, the integer variables are explicitly and accurately represented. As far as we know, this is the first time that the integer variables of these programs are explicitly modeled.

The Lock/unlock example is a system that handles an arbitrary number of concurrent insertions on a binary search tree. We were able to find a bug in a version of the program given in [9].

The Philosophers example is a buggy algorithm that solves the dining philosophers problem. Our tool was able to find an execution of the program that reaches a deadlock point where all the philosophers have the fork.

The Prod/Cons example is a version of a Producer/Consumer algorithm, where the producer puts data in a buffer, and the consumer consumes this data. Our tool shows that the program reaches a buggy configuration where the producer is not aware that the buffer is empty.

Finally, Toy Example is a toy example that we have written ourselves to test the power of our technique. This program includes several assignments, loops, and procedure calls.
References


