Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 110 (2004) 55–74
www.elsevier.com/locate/entcs

Model Checking Multithreaded Programs by Means of Reduced Models

Sara Gradara, Antonella Santone, Maria Luisa Villani[1]

RCOST - Research Centre on Software Technology
University of Sannio, Italy

Gigliola Vaglini[2]

Dipartimento di Ingegneria della Informazione
University of Pisa, Italy

Abstract

Java is largely used to develop distributed and concurrent systems, but testing multithreaded systems cannot guarantee the quality of the software; in contrast, verification techniques give us a higher confidence about the system and, among these, model checking methods *automatically* establish properties of complex systems. Such techniques are usually applied to specification languages, and several environments exist to verify temporal properties of concurrent specifications. In this paper we present an attempt to apply model checking techniques for verifying a subset of multithreaded Java programs. In particular, we use a tool based on the selective mu-calculus logic to check systems described through the CCS specification language.

Keywords: state explosion, model checking, CCS, Java.

1 Introduction

Concurrent systems are becoming more and more interesting but such systems are usually quite complex. Also, as modern computing applications require highly reliable software systems, testing fails to assure an adequate level of

[1] Email: {gradara, santone, villani}@unisannio.it
[2] Email: g.vaglini@iet.unipi.it

doi:10.1016/j.entcs.2004.06.010

correctness. Formal verification techniques are a better choice, and, in particular, model checking is useful for automatically establishing properties of complex systems. Verification techniques are usually applied to concurrent specification languages, like the Calculus of Communicating Systems (CCS) [32] and the Language Of Temporal Ordering Specification (LOTOS) [9], due to the simplicity of their semantics. Some complete model checking environments exist in this context, see for example the Concurrency Workbench of the New Century (henceforth CWB-NC) [15] that performs verification of temporal properties expressed as mu-calculus formulae [35].

In this paper, we take into account several problems bounding the use of model checking techniques. The first problem is the lack of a formal link between the system specification and the code representing the final product. Thus, if we first formally specify the system behavior we can verify the correctness of the properties on this specification. But deriving in some way the *corresponding* program, we have no guarantee about its correctness. A further problem is that models used to represent the behavior of concurrent systems, for example transition systems, suffer of the *state explosion* problem, due to the representation of concurrency by interleaving; this problem curbs the effective use of model checking techniques to complex systems. Finally, beside the system behavior, also the system requirements must be expressed in a formal way when applying verification techniques: this is usually hard for ordinary developers.

In this paper we present an attempt to apply model checking techniques for verifying a subset of multithreaded Java programs. In particular, we use a tool based on:

- the Calculus of Communicating Systems (CCS) [32] that is a specification language widely used for concurrent and distributed systems; and
- the temporal logic called selective mu-calculus (see [5,6,7]) for describing system requirements. Such logic allows us to automatically reduce the CCS specification on the basis of the formula to be verified, so obtaining a less expensive model which is an abstraction of the system with respect to the formula (property driven abstractions are also used in [8,10,16], even if not syntactically deduced). Although equi-expressive to mu-calculus, the selective logic provides a more concise and intuitive way to express properties, and so stating the system requirements will be easier; and
- CWB-NC tool: since selective formulae can be automatically translated into mu-calculus formulae, it is possible to use the CWB-NC tool-kit for model checking.

In order to apply the above tool, we have defined a Java-to-CCS transform

operator. A similar approach is in [11]. Using this approach, as a result we obtain that the gap between the specification and its implementation is bridged.

The paper is organized as follows: in Section 2 we review the basic concepts of CCS and the methodology, based on selective mu-calculus, to attack the state explosion problem. In Section 3 we describe the transformation of multi-threaded Java programs into CCS specifications, while in Section 4 the usefulness of our approach in terms of the degree of the model reduction is shown through a simple synchronization problem. Finally, considerations and comparisons with related work are given in Section 5.

2 Background

In this section, we briefly recall the main concepts about CCS and selective mu-calculus. CCS [32] is a specification language widely used for concurrent and distributed systems and we assume the reader to be familiar with it. The selective mu-calculus is a branching temporal logic to express behavioral properties of systems, which has been introduced by the authors et al. in [6].

2.1 The Calculus of Communicating Systems

Consider the finite set of actions $\mathcal{A} = \{\tau, a, \overline{a}, b, \overline{b}, ...\}$. The action $\tau \in \mathcal{A}$ is called the *internal action*. The set of *visible actions*, $\mathcal{V}$, ranged over by l, is defined as $\mathcal{A} - \{\tau\}$. Each action $l \in \mathcal{V}$ ($\overline{l} \in \mathcal{V}$) has a *complementary action* $\overline{l}$ (l). The following syntax defines a CCS *process*.

$$p ::= nil \mid x \mid \alpha.p \mid p + p \mid p|p \mid p\backslash L \mid p[f]$$

where $\alpha \in \mathcal{A}$, $L \subseteq \mathcal{V}$ and the relabelling function f is a total function $f : \mathcal{A} \rightarrow \mathcal{A}$, such that the constraint $f(\tau) = \tau$ is respected. Also, x is a *constant* name: each constant x is defined by a constant definition $x \stackrel{\text{def}}{=} p$, where p is called the *body* of x.

The operational semantics of a process p is a labelled transition system, $\mathcal{S}(p)$, i.e., an automaton whose states correspond to processes (the initial state corresponds to p), and whose arcs are labelled by actions in $\mathcal{A}$ and correspond to transitions from state to state. Such structural operational semantics is precisely defined by means of the rules shown in Table 1.

Act	$\dfrac{}{\alpha.p \xrightarrow{\alpha} p}$	**Sum**	$\dfrac{p \xrightarrow{\alpha} p'}{p+q \xrightarrow{\alpha} p'}$ (and symmetric)
Con	$\dfrac{p \xrightarrow{\alpha} p'}{x \xrightarrow{\alpha} p'}\ x \stackrel{\text{def}}{=} p$	**Par**	$\dfrac{p \xrightarrow{\alpha} p'}{p\|q \xrightarrow{\alpha} p'\|q}$ (and symmetric)
Com	$\dfrac{p \xrightarrow{l} p',\ q \xrightarrow{\overline{l}} q'}{p\|q \xrightarrow{\tau} p'\|q'}$	**Rel**	$\dfrac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$
		Res	$\dfrac{p \xrightarrow{\alpha} p'}{p\backslash L \xrightarrow{\alpha} p'\backslash L}\ \alpha, \overline{\alpha} \notin L$

Table 1
Operational semantics of CCS.

2.2 Model checking and selective mu-calculus

A model checker [14] accepts two inputs, a transition system and a temporal formula, and returns "true" if the system satisfies the formula, "false" otherwise; in the last case a counter-example useful to locate and correct errors is produced. The major problem in model checking is the *state explosion*: in fact, systems are often described by transition systems with a prohibitively large number of states. The primary cause of this problem is the parallel composition of interacting processes. When n processes of size (number of states) m are composed in parallel, the resulting process can be of size m^n.

The selective mu-calculus, defined in [6], although equi-expressive to mu-calculus [35], differs from it in the definition of the modal operators. Given a set $\mathcal{A}$ of actions and a set *Var* of variables, selective mu-calculus formulae are obtained through the following definition:

$$\varphi ::= \texttt{tt} \mid \texttt{ff} \mid Z \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid [K]_R\, \varphi \mid \langle K \rangle_R\, \varphi \mid \nu Z.\varphi \mid \mu Z.\varphi$$

where $Z \in \mathit{Var}$ and $K, R \subseteq \mathcal{A}$. The operators $\mu Z.\varphi$ and $\nu Z.\varphi$ are fixed point operators: $\mu Z.\varphi$ is the least fixed point of the recursive equation $Z = \varphi$, while $\nu Z.\varphi$ is the greatest one. In the formula $\mu Z.\varphi$ ($\nu Z.\varphi$), μZ (νZ) *binds* the occurrences of Z in φ. A variable occurring outside the scope of fixed point operators is called *free*. A formula without free variables is called *closed*. From now on we only consider closed formulae.

The state s of a transition system satisfies the selective formula φ, written $s \models \varphi$, as follows:

- s always satisfies `tt` and never `ff`; and
- s satisfies $\varphi_1 \vee \varphi_2$ ($\varphi_1 \wedge \varphi_2$) if it satisfies φ_1 or (and) φ_2; and
- s satisfies $[K]_R\, \varphi$ if, for every possible sequence of actions not containing

actions in $R \cup K$, followed by an action in K, it will evolve to a state obeying to φ; and

- s satisfies $\langle K \rangle_R \varphi$ if it will evolve to a state obeying to φ after at least a sequence of actions not containing actions in $R \cup K$, followed by an action in K.

A transition system satisfies φ if and only if $s \models \varphi$, where s is its initial state. A CCS process p satisfies φ if $\mathcal{S}(p)$ satisfies φ. The precise definition of the satisfaction of the closed formula φ by the process p is given in Table 2: the transition relation $\Longrightarrow_I$, parametric with respect to $I \subseteq \mathcal{A}$, is defined as follows and ignores all non-interesting actions (i.e., those in $\mathcal{A} - I$).

Definition 2.1 [$\Longrightarrow_I$ relation] Let p be a CCS process and $I \subseteq \mathcal{A}$, the relation $\Longrightarrow_I$ is such that, for each $\alpha \in I$, $p \stackrel{\alpha}{\Longrightarrow}_I q$ iff $p \stackrel{\delta\alpha}{\longrightarrow} q$, where $\delta \in (\mathcal{A} - I)^*$.

Note that $\Longrightarrow_{\mathcal{A}} = \longrightarrow$.

$$
\begin{array}{lll}
p \not\models \texttt{ff} & & p \models \texttt{tt} \\
p \models \varphi \wedge \psi & \text{iff} & p \models \varphi \text{ and } p \models \psi \\
p \models \varphi \vee \psi & \text{iff} & p \models \varphi \text{ or } p \models \psi \\
p \models [K]_R \varphi & \text{iff} & \forall p'. \forall \alpha \in K. p \stackrel{\alpha}{\Longrightarrow}_{K \cup R} p' \text{ implies } p' \models \varphi \\
p \models \langle K \rangle_R \varphi & \text{iff} & \exists p'. \exists \alpha \in K. p \stackrel{\alpha}{\Longrightarrow}_{K \cup R} p' \text{ and } p' \models \varphi \\
p \models \nu Z. \varphi & \text{iff} & p \models \nu Z^n. \varphi \text{ for all } n \\
p \models \mu Z. \varphi & \text{iff} & p \models \mu Z^n. \varphi \text{ for some } n
\end{array}
$$

where, for each n, $\nu Z^n.\varphi$ and $\mu Z^n.\varphi$ are defined as:

$$
\begin{array}{ll}
\nu Z^0.\varphi = \texttt{tt} & \mu Z^0.\varphi = \texttt{ff} \\
\nu Z^{n+1}.\varphi = \varphi[\nu Z^n.\varphi/Z] & \mu Z^{n+1}.\varphi = \varphi[\mu Z^n.\varphi/Z]
\end{array}
$$

and $\varphi[\psi/Z]$ indicates the substitution of ψ for each free occurrence of Z in φ.

Table 2
Satisfaction of a closed formula by a process.

Example 2.2 We give some examples of selective mu-calculus formulae to explain the use of the selective operators.

$\varphi_1 = [b]_{\{a\}}\,\texttt{ff}$: *"b cannot be performed if a has not been performed before".*

$\varphi_2 = \langle b \rangle_\emptyset \, \mathtt{tt}$: *"it is possible to perform b preceded by any action".*

$\varphi_3 = \nu Z.\, [a]_\emptyset (Z \wedge [a]_{\{c\}} \, \mathtt{ff})$: *"it always holds that, after a has occurred, a successive a cannot occur if c has not occurred before".*

Consider the processes: $x \stackrel{\text{def}}{=} b.c.x. + a.b.c.x \quad y \stackrel{\text{def}}{=} a.b.c.y \quad z \stackrel{\text{def}}{=} b.z + a.b.z$ which transition systems are in Figure 1. It holds that:

$x \not\models \varphi_1 \;\; x \models \varphi_2 \;\; x \models \varphi_3 \;\; y \models \varphi_1 \;\; y \models \varphi_2 \;\; y \models \varphi_3 \;\; z \not\models \varphi_1 \;\; z \models \varphi_2 \;\; z \not\models \varphi_3 \quad \square$

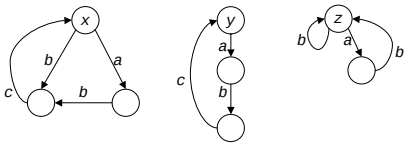

Fig. 1. Three transition systems.

As shown in [6], selective formulae can be easily translated into mu-calculus recursive formulae exploiting the following definitions.

$$[K]_R \, \varphi = \nu Z.\, [K] \, \varphi \wedge [\mathcal{A} - (K \cup R)] \, Z \qquad \langle K \rangle_R \, \varphi = \mu Z.\, \langle K \rangle \, \varphi \vee \langle \mathcal{A} - (K \cup R) \rangle \, Z$$

The main point is that, contrarily to mu-calculus, the selective mu-calculus allows us to immediately point out, from each formula, the parts of the transition system that do not alter the truth value of the formula itself. More precisely, the result of the checking only depends on *the actions explicitly mentioned in the modal operators used in the formula* (namely, *occurring actions*). For example, consider the process $p = b.c.a.nil$ and suppose that we want to verify the property "it is possible to perform an action a preceded by any action". The property can be expressed using the mu-calculus logic as:

$$\psi = \mu Z.\, \langle a \rangle \, \mathtt{tt} \vee \langle b, c \rangle \, Z$$

Using the selective mu-calculus logic, the above property can be equivalently expressed as:

$$\psi_{sel} = \langle a \rangle_\emptyset \, \mathtt{tt}$$

It is easy to see that the property holds on $\mathcal{S}(p)$ (see Figure 2(a)). Note that the same property holds also on a much smaller transition system, depicted in Figure 2(b), and the two systems are equivalent with respect to the formula. Such system may be obtained from $\mathcal{S}(p)$ by keeping only the transitions that are labelled by the action a and collapsing the states consequently. Thus, given a formula, the problem is to find the right set of actions that do not

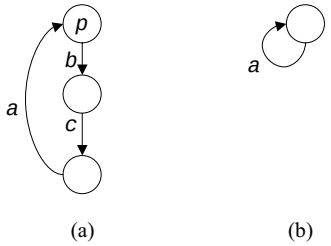

Fig. 2. Transition systems.

alter the truth value of the formula. Now, mu-calculus formulae are not always suitable to identify these actions. For example, the set of actions occurring in the formula ψ is $\{a, b, c\}$, while only a, which appears alone in ψ_{sel}, is a relevant action for preserving the truth value of the formula.

In [6] ρ-equivalence is defined to formally characterize the notion of "the same behavior with respect to a set ρ of actions": *two transition systems are ρ-equivalent if and only if they satisfy the same set of formulae with occurring actions in ρ* (see [8,10] for a similar equivalence). Thus, improvements in model checking a process p can be obtained by eliminating from $\mathcal{S}(p)$ the actions not in ρ. Obviously, the reduction degree mainly depends on the size of ρ with respect to the size of $\mathcal{A}$.

Example 2.3 Recall the formulae of Example 2.2 and the transition systems in Figure 1. $\mathcal{S}(x)$ is $\{a, b\}$-equivalent to $\mathcal{S}(z)$, on the contrary $\mathcal{S}(y)$ is not $\{a, b\}$-equivalent to $\mathcal{S}(z)$. Thus $\mathcal{S}(x)$ and $\mathcal{S}(z)$ give the same result for the checking of the formulae with occurring actions in $\{a, b\}$; in particular, they satisfy φ_2, while they do not satisfy φ_1. Note that the set of occurring actions of φ_1 is $\{a, b\}$, and that of φ_2 is $\{b\}$ which is a subset of $\{a, b\}$. □

In [5] a CCS process p is transformed into another process q on which the selective formula φ can be equivalently checked [3]. The method is based on a set of syntactic transformation rules, of the form $p \longmapsto q$; the rules are defined in Table 3 and consist of one *action deleting rule* and two *compacting rules*. The first one deletes (when possible) actions not belonging to a given set ρ, which includes the occurring actions of φ, from p; the actions must not be the first action of a choice operator in the scope of a parallel composition (this condition needs to preserve ρ-equivalence). The other two rules compact processes by eliminating redundant branches of choices. An algorithm has

[3] A similar method has been defined for LOTOS processes in [7].

Action deleting rule: $\alpha.p \longmapsto p$, if $\alpha \notin \rho$ and α is not the first action of a choice branch in the scope of a parallel composition

Compacting rules: $C_1 : \alpha.p + p \quad \longmapsto \alpha.p$ if $\alpha \notin \rho$

$$C_2 : \alpha.p + \beta.p \longmapsto \alpha.p \text{ if } \alpha, \beta \notin \rho$$

Table 3
The transformation rules.

been defined for reducing p by repeatedly applying the transformation rules until this is not possible. A prototype tool implements such algorithm, while another tool translates selective formulae in mu-calculus formulae. These tools have been integrated in a system named CCS Reduction Tool (see Figure 3), whose output are in the format of the CWB-NC environment, so that this environment can be used for building the transition system modeling the reduced CCS specification and for checking selective formulae on it.

3 Efficient Verification of Java Programs

The methodology proposed in the previous section has been applied by the authors in the design phase of a concurrent system (see for example [30]). From the positive results obtained there, we thought that the same methodology could be useful in the context of multi-threaded programs verification, through their translation into the specification language of an existing model checker, i.e., the CWB-NC tool. This is a more challenging area, as real programs are usually big and their abstraction to finite state models may be still intractable by model checking techniques that suffer from the state explosion problem. We believe that efforts to integrate reduction solutions should be made in this respect.

Our idea is illustrated in Figure 3. Tools for abstracting Java programs, applying data or formula-based abstraction techniques like slicing (see for example Bandera [16]), are used to provide us with a reasonable starting point for a translation into CCS specifications. We have defined a Java-to-CCS transform operator to this aim, described in Section 3.1. A similar approach is presented in [11]: however, our method seems to be more systematic, the translation rules are modular and have been thought in the perspective of being performed automatically. Following the same idea, transform operator from other programming languages can be easily defined.

The resulting CCS process is then analyzed by our CCS Reduction Tool that automatically removes all the non-relevant actions with respect to the formula to be checked. The main advantage is that our tool can be combined with, and not replace, other techniques developed to attack the state explosion problem, including *partial order* methods [21,36] which remove unnecessary interleaving of transitions, *abstraction* techniques [13] which ignore some state information, *compositional reasoning* [1,12,33] and methods based on *heuristic searches* [20,34]. In fact, once we have the CCS specification corresponding to the initial Java program, we can apply the above methods.

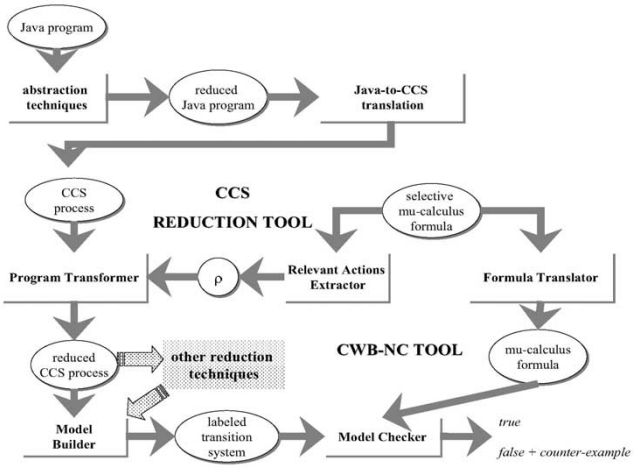

Fig. 3. The tool.

3.1 Transforming a Java program into a CCS process

The reader may refer to [27] for details about the essential Java constructs. Here we explain the semantics of such constructs through their translation into CCS processes.

The methodology is based on the following two assumptions:

- the number of objects in the program is statically defined; and
- each set of variable values is finite. This can be achieved through suitable abstraction schemes like for example data abstraction [16,19] or predicate abstraction [2,17]. Abstraction techniques have to be applied to make model checking feasible, since most software programs have an infinite number of states. In this paper we concentrate on the boolean data abstraction, which has been proven to be still quite efficient in this context.

As a consequence of the first assumption, (non-static) state variables and methods of a class are replicated for each object and translated singularly into CCS processes. The second constraint is to assure a finite number of states for both variables and methods that use them.

All objects, variables, and methods have an index. Namely,

- $\mathcal{V}$ is the index set for variables; and
- $\mathcal{O}$ is the index set for objects and methods.

Variables include object state variables and local variables passed to, or used in, a method. The methods and Java features for synchronization that we have formalized, such as the object lock and wait set, have the same identifier as the object to which they refer.

We have added the following operator to the CCS algebra to express the execution order of two processes.

Definition 3.1 [The sequence operator ;] Let p and q be two CCS processes. The sequence operator ; is defined as: $p \; ; q \stackrel{\text{def}}{=} p\{q/nil\}$, with the effect of replacing each occurrence of nil in p, and in the bodies of all the contained constants, with the process q.

The operator $\mathcal{T}$, defined in the remaining of this section, applies to the Java code of a program to translate it into CCS specifications. For lack of space, in this paper we only present a portion of the definition of $\mathcal{T}$, in particular what is needed to understand the application of our method on the example described in Section 4. More details about the definition of $\mathcal{T}$ are in [22].

3.2 Definition of variables and methods processes

The definitions of the processes corresponding to each variable and method are shown in Table 4.

Boolean variable definition. Each time, the boolean variable VAR_i may be seen as the process corresponding to its current value. We have defined two processes, $VAR\mathtt{ff}_i$ and $VAR\mathtt{tt}_i$, accordingly. For example, the branch $\overline{isFalse_i}$. $VAR\mathtt{ff}_i$ represents the false state of the variable; this state will change to true after the execution of the action $setTrue_i$ and the process will continue as $VAR\mathtt{tt}_i$.

Main definition. The method $main()$ is translated as the process $MAIN$, where $\mathcal{T}$ is applied to the instructions contained in I according to the rules of

$$\mathcal{T}(\textit{boolean } VAR_i) = VAR\mathtt{ff}_i \stackrel{\text{def}}{=} \overline{isFalse_i}.VAR\mathtt{ff}_i + setFalse_i.VAR\mathtt{ff}_i + setTrue_i.VAR\mathtt{tt}_i$$
$$VAR\mathtt{tt}_i \stackrel{\text{def}}{=} \overline{isTrue_i}.VAR\mathtt{tt}_i + setTrue_i.VAR\mathtt{tt}_i + setFalse_i.VAR\mathtt{ff}_i$$
$$\mathcal{T}(main()\{I\}) = MAIN \stackrel{\text{def}}{=} \mathcal{T}(I)$$
$$\mathcal{T}(methodName_i()\{I\}) = METHODNAME_i \stackrel{\text{def}}{=} callMethodName_i.\mathcal{T}(I);\ \overline{returnMethodName_i}.METHODNAME_i$$
$$\mathcal{T}(run_i()\{I\}) = RUN_i \stackrel{\text{def}}{=} callRun_i.\mathcal{T}(I)$$

Table 4
Definitions of variables and methods.

Table 5.

Method definition. Every method corresponds to a process waiting to be invoked through the action $callMethodName_i$. The action $\overline{returnMethodName_i}$ causes the control being returned to the method caller. Then the process goes back to the starting action so that it can be invoked again. A special case is the process RUN_i, corresponding to the method $run_i()$ of a thread object, which can only be executed once.

3.3 Instructions

Table 5 shows the translation of the main Java instructions.

$$\mathcal{T}(I_1; I_2) = \mathcal{T}(I_1); \mathcal{T}(I_2)$$
$$\mathcal{T}(VAR_i = true;) = \overline{setTrue_i}.nil$$
$$\mathcal{T}(while\ (VAR_i == true)\ I) = WHILE \quad \text{where} \quad WHILE \stackrel{\text{def}}{=} isTrue_i.\mathcal{T}(I); WHILE + isFalse_i.nil$$
$$\mathcal{T}(if\ (VAR_i == true)\ I_1; I_2) = IF \quad \text{where} \quad IF \stackrel{\text{def}}{=} isTrue_i.\mathcal{T}(I_1) + isFalse_i.\mathcal{T}(I_2)$$
$$\mathcal{T}(methodName_i();) = \overline{callMethodName_i}.returnMethodName_i.nil$$
$$\mathcal{T}(run_i();) = \overline{callRun_i}.nil$$
$$\mathcal{T}(synchronized\ (this)\{I\}) = \overline{lock_i}.\mathcal{T}(I); \overline{unlock_i}.nil$$

Table 5
Instructions.

Sequence of two Java instructions. The execution order of two Java instructions is expressed by the use of the operator ; (introduced in Definition 3.1) to sequentialize the translated processes. We assume that these instructions are contained in the definition of some $methodName_i()$.

The translation of the assignment operations and of the "while" and "if" constructs reflects their meaning in the programming theory. In particular, they are synchronized with some variable process.

Assignment operations. The assignment of the true value to the variable VAR_i is performed by a process that executes the action $\overline{setTrue_i}$ and then terminates. The assignment of the false value is defined similarly.

While construct. The *WHILE* process simulates the while control construct. Namely, the current value of VAR_i is checked through the actions $isTrue_i$ and $isFalse_i$: depending on what action is performed, the process corresponding to the block instruction I is activated or not.

If construct. The *IF* process simulates the if control construct interrogating the current value of VAR_i: either the translation of the block instruction I_1 or that corresponding to I_2 is executed, depending on the performed action ($isTrue_i$ or $isFalse_i$, respectively).

Method call. The call to the method "$methodName_i$" through the object i is performed by the action $\overline{callMethodName_i}$. The control to the caller process is returned through the action $returnMethodName_i$. Instead, for the $run_i()$ method the control is immediately returned to allow for a parallel execution of the processes.

Java synchronized block or method. Before translating I, the lock of the object i must be acquired, through the action $\overline{lock_i}$; the action $\overline{unlock_i}$ follows the translation of I to release the lock (see Table 6).

3.4 Concurrency constructs and methods

The translation of the Java constructs and methods for concurrency is shown in Table 6. Namely, the process *LOCK* is to describe the *monitor* concept, the processes *WAIT* and *NOTIFY* correspond to the *wait()* and *notify()* methods, and the processes *WAITSET* represent the $wait-set$ management associated with any object. A careful analysis of the Java language specifications has been made in order to represent these constructs.

Lock. This process simulates the lock and unlock mechanisms of the object

$LOCK_i \stackrel{\text{def}}{=} lock_i.unlock_i.LOCK_i$

$WAIT_i \stackrel{\text{def}}{=} callWait_i.\overline{insert_i}.\overline{unlock_i}.resume_i.\overline{lock_i}.\overline{returnWait_i}.WAIT_i$

$NOTIFY_i \stackrel{\text{def}}{=} callNotify_i.(\ someoneInQueue_i.\overline{resume_i}.\overline{remove_i}.\overline{returnNotify_i}.NOTIFY_i +$
$noneInQueue_i.\overline{returnNotify_i}.NOTIFY_i)$

$WAITSET(0)_i \stackrel{\text{def}}{=} insert_i.WAITSET(1)_i +$
$\overline{noneInQueue_i}.WAITSET(0)_i$

Let j be such that $0 < j < n-1$. Then

$WAITSET(j)_i \stackrel{\text{def}}{=} remove_i.WAITSET(j-1)_i +$
$\overline{someoneInQueue_i}.WAITSET(j)_i +$
$insert_i.WAITSET(j+1)_i$

$WAITSET(n-1)_i \stackrel{\text{def}}{=} remove_i.WAITSET(n-2)_i +$
$\overline{someoneInQueue_i}.WAITSET(n-1)_i$

Table 6
Definitions to handle synchronization.

i allowing concurrent threads to execute synchronized instructions after the action $lock_i$.

WAIT. The call to the method *wait()* always occurs in a synchronized block. This causes the thread being inserted in the wait-set of the object through the action $\overline{insert_i}$ that will be caught by one of the $WAITSET(j)_i$ processes, depending on the number j $(0 < j < n)$ of the waiting threads. Then the object lock is released through the action $\overline{unlock_i}$, so that the object i may be used again. Then $WAIT_i$ waits for the action $\overline{resume_i}$ from the process $NOTIFY_i$ and executes the action $\overline{lock_i}$ before returning the control and accepting a new invocation.

NOTIFY. The call to the method *notify()* sets free one of the threads in the wait-set of the object i, if any. The actions $someoneInQueue_i$ and $noneInQueue_i$, communicating with the $\overline{someoneInQueue_i}$ and $\overline{noneInQueue_i}$ actions of the $WAITSET(j)_i$ process definitions, show the state of the wait-set. If this is empty, nothing is done, whereas if the wait-set contains at least one thread, the actions $\overline{resume_i}$ (caught by the $WAIT_i$ process) and $\overline{remove_i}$ will resume and remove from the wait-set one of the waiting threads.

WAIT-SET. We assume that the program consists of at most n threads. So the wait-set associated with the shared object i may contain at most $n-1$ threads at the time. The translation of this concept consists of n $WAITSET$ processes. Namely, $WAITSET(0)_i$ represents an empty wait-set, while $WAITSET(n-1)_i$ represents a full wait-set. The insertion of a thread in the wait-set is required

through the action $\overline{insert_i}$ performed by the process $WAIT_i$.

Final CCS process

The final CCS process describing a Java program is obtained through the proposed methodology as a parallel of the *MAIN* process and all the *RUN*, *VAR*, *LOCK*, *WAIT*, *NOTIFY*, *WAITSET* and *METHODNAME* processes, restricted to all the actions used for communicating among such processes. Readers can refer to [22] for a more exhaustive explanation of the final process structure.

The proof of the correctness of the translation is ongoing and discussed in [22].

4 An example

We have applied our technique to check the correctness of a simple Java program; the Java code is presented in Tables 9 and 10 of the Appendix A. The program describes a tavern with a client (drinker) and two hosts (host1 and host2): both hosts may fill the client glass when empty, whereas the client waits for the glass being completely full to drink from it. This fact is expressed by the following two properties:

Property 1: filling the glass is not allowed if the glass is not empty.

Property 2: drinking from the glass is not allowed if the glass is not full.

In order to model check the program, the two properties above are expressed by selective formulae, as follows:

$\varphi_1 : [\,'filling]_{\{'empty\}}\,\mathtt{ff}$ [4]

$\varphi_2 : [\,'drinking]_{\{'full\}}\,\mathtt{ff}$

First we translated the Java program into a CCS process, named *prog*, according to the Java-to-CCS operator of Section 3 (see Table 11 of the Appendix B). Then we attempted to reduce the specification: we applied the transformation rules starting from the sets of actions ρ_i occurring in the formula φ_i, for each $i \in \{1, 2\}$. Finally, we checked the mu-calculus formulae corresponding to φ_i on the reduced specifications.

[4] Note that the action $'filling$ corresponds to the print instruction `System.out.println(‘‘filling")` in the method `fill` (similarly for $'full$, $'drinking$ and $'empty$). Moreover, the CWB-NC represents output actions as $'a$ instead of $\overline{a}$.

For the experiment we used the CWB-NC tool on a PC based on a 1,60 GHz Pentium 4 processor with 256 Mbytes of memory and Microsoft Windows XP operating system. The result was that the formula φ_1 is false, instead the formula φ_2 is true. In fact, the code presents a mistake as the operation of reading the empty status of the glass and filling it is not atomic and therefore one host may actually fill a non-empty glass.

Table 7 shows the number of states and transitions of $\mathcal{S}(prog)$ and of $\mathcal{S}(Tprog_{\rho_i})$, $i \in \{1, 2\}$, where $Tprog_{\rho_i}$ is the reduced CCS process obtained by applying the transformation rules to $prog$ with ρ_i. More precisely, $\rho_1 = \{'filling, 'empty\}$, and $\rho_2 = \{'drinking, 'full\}$. For lack of space, the reduced CCS specifications are not shown in this paper. It is worth noting that the state space reduction that we obtained is significant. This implies a corresponding reduction of the verification time: Table 8 shows the time employed by the CWB-NC model checker to check the formulae on the standard and on the reduced transition systems respectively, together with the corresponding time reduction.

$\mathcal{S}(prog)$		$\mathcal{S}(Tprog_{\rho_1})$			$\mathcal{S}(Tprog_{\rho_2})$		
states	transitions	states	transitions	state space reduction %	states	transitions	state space reduction %
6975	18386	4370	11200	38,6%	4316	11068	39,3%

Table 7
Results for the proposed program.

	standard transition system	reduced transition system	time reduction %
φ_1	14,9s	9,0s	39,5%
φ_2	14,6s	8,7s	40,4%

Table 8
Verification time of φ_1 and φ_1.

The results obtained with this simple example are encouraging. Obviously, this is only a starting point: further investigation will allow the validation of the results on relevant problems in software verification, and the comparison with other tools. In general, the usefulness of our tool depends both on the number of actions occurring in the formula and on the structure of the formula to be checked.

5 Conclusion and Related Work

Many works can be found in the recent literature aiming to verify real programs written in modern languages. As stated in [37], they can be roughly divided into two categories: *custom-made model checker* and *source-to-source translation*. An example of a custom-made model checker is described in [37], where a verification and testing environment for Java, Java PathFinder (JPF), has also been developed. The works in the second category (see, for example, [16,18,24,29]), translate software system descriptions to the input languages of verification tools. For example, in the Bandera tools [16,23] the Java source code, after some manipulation, is translated into either Promela (the input notation of Spin [28]) or SMV [31] model checker input notation. Bandera uses temporal patterns, to be instantiated to temporal logics and to specify properties, and uses program slicing and data abstraction (abstract interpretation) to customize models. With our methodology, we propose to add a further abstraction, which is driven by the formula being checked.

Interesting complete environments for checking non-Java programs are:

- the SLAM toolkit [3,4] which uses a predicate abstraction tool to transform C programs into boolean programs that respect the given set of predicates representing the required properties; then a suitable tool model checks such boolean programs; and
- BLAST [25,26], a model checker for C programs which uses counterexample-driven automatic abstraction refinement to construct an abstract model to be model checked for safety properties.

Our approach belongs to the source-to-source translation category. It consists of the translation of multithreaded Java programs into CCS specifications and of the use of a reduction algorithm, based on the selective mu-calculus logic, which allows us to face the state explosion problem and facilitates the writing of the properties. In order to use the existing model checking environment for concurrent specifications, namely CWB-NC, we use a prototype tool to implement the reduction algorithm and another tool to translate a selective formula into a mu-calculus one. This approach permits an easy detection of errors in a program, and this fact is relevant as it occurs with low probability on concurrent programs; moreover, the error detection is obtained at a lower cost, also on complex systems, compared to testing the program, because of the use of a reduced model.

References

[1] Andersen H.R., C. Stirling and G. Winskel. *A Compositional Proof System for the Modal mu-Calculus*, Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science (LICS'94), IEEE Computer Society Press, 1994, 144-153.

[2] Ball T., R. Majumdar, T. Millstein and S.K.Rajamani. Automatic *Predicate Abstraction of C Programs*, Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001), ACM, 2001, 203-213.

[3] Ball T., and S.K.Rajamani, *The SLAM Toolkit*, Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001), Lecture Notes in Computer Science **2102**, 2001, 260-264.

[4] Ball T., and S.K. Rajamani, *The SLAM Project: Debugging System Software via Static Analysis*, Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002), 2002, 1-3.

[5] Barbuti R., N. De Francesco, A. Santone and G. Vaglini, *Constructing Reduced Models for Temporal Properties Verification*, MOSAICO Project Technical Report PI-DII/5/98, July 1998.

[6] Barbuti R., N. De Francesco, A. Santone and G. Vaglini, *Selective mu-calculus and Formula-Based Abstractions of Transition Systems*, Journal of Computer and System Sciences, **59(3)**, 1999, 537-556.

[7] Barbuti R., N. De Francesco, A. Santone and G. Vaglini, *Loreto: a Tool for Reducing State Explosion in Verification of LOTOS programs*, Software-Practice and Experience, **29(12)**, 1999, 1123-1147.

[8] Bensalem S., A. Bouajjani, C. Loiseaux and J. Sifakis, *Property Preserving Simulations*, Proceedings of the Fourth Workshop on Computer Aided Verification (CAV'92), Lecture Notes in Computer Science **663**, 1992, 260-273.

[9] Bolognesi T., and E. Brinksma, *Introduction to ISO Specification Language LOTOS*, Comp. Networks and ISDN Systems, 14, 1987, 25-59.

[10] Bouajjani A., J.C. Fernandez and N. Halbwachs, *Minimal Model Generation*, Proceedings of the International Conference on Computer-Aided Verification (CAV'90), Lecture Notes in Computer Science **531**, 1990, 197-203.

[11] Chen J., *On Verifying Distributed Multithreaded Java Programs*, Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS-33), 2000.

[12] Clarke E.M., D.E. Long and K.L. McMillan, *Compositional Model Checking*, Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science, 1989, 353-362.

[13] Clarke E.M., O. Grumberg and D.E. Long, *Model Checking and Abstraction*, ACM Transactions on Programming Languages and Systems, **16(5)**, 1994, 1512-1542.

[14] Clarke E.M., O. Grumberg and D. Peled, "Model Checking", MIT press, 2000.

[15] Cleaveland R., and S.Sims, *The NCSU Concurrency Workbench*, Proceedings of the Eighth International Conference on Computer-Aided Verification (CAV'96), Lecture Notes in Computer Science **1102**, 1996, 394-397.

[16] Corbett J., M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach and H. Zheng, *Bandera: Extracting Finite-state Models from Java Source Code*, Proceedings of the 22nd Int. Conf. on Software Engineering 2000. ACM Press, 2000, 439-448.

[17] Das S., D.L. Dill and S. Park, *Experience with Predicate Abstraction*, Proceedings of the 11th International Conference on Computer-Aided Verification (CAV'99), Lecture Notes in Computer Science **1633**, 1999, 160-171.

[18] Demartini C., R. Iosif and R. Sisto, *A Deadlock Detection Tool for Concurrent Java Programs*, Software Practice and Experience, **29(7)**, 1999, 577-603.

[19] Dingel J., and T. Filkorn, *Model Checking for Infinite State Systems Using Data Abstraction, Assumption-Commitment Style reasoning and Theorem Proving*, Proceedings of the 7th International Conference on Computer Aided Verification (CAV'95), Lecture Notes in Computer Science **939**, 1995, 54-69.

[20] Edelkamp S., A. Lluch-Lafuente and S. Leue, *Directed Explicit Model Checking with HSF-SPIN.* Proceedings of the 8th International SPIN Workshop on Model Checking Software, Lecture Notes in Computer Science **2057**, 2001, 57-79.

[21] Godefroid P., and P. Wolper, *A Partial Approach to Model Checking*, Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS), 1991.

[22] Gradara S., A. Santone and M.L.Villani, *Efficient Verification of Java Programs*, Technical Report, TR-RCOST 12/03, April 2003.

[23] Hatcliff J., M.B. Dwyer and H. Zheng, *Slicing Software for Model Construction*, Higher-Order and Symbolic Computation **13(4)**, 2000, 315-353.

[24] Havelund K., and T. Pressburger, *Model Checking Java Programs using Java PathFinder*, Int. Journal on Software Tools for Technology Transfer (STTT), **2(4)**, 2000, 366-381.

[25] Henzinger T.A., R. Jhala, R. Majumdar and G. Sutre, *Lazy Abstraction*, Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002), 2002, 58-70.

[26] Henzinger T. A., R. Jhala, R. Majumdar, G.C. Necula, G. Sutre and W. Weimer, *Temporal-Safety Proofs for Systems Code*, Proceedings of the 14th International Conference on Computer Aided Verification (CAV'2002), Lecture Notes in Computer Science **2404**, 2002.

[27] Holub A., "Taming Java Threads", APress, June 2000.

[28] Holzmann G.J., *The Model Checker Spin*, IEEE Trans. on Software Engineering, **23(5)**, 1997, 279-295. Special issue on Formal Methods in Software Practice.

[29] Holzmann G.J., and M.H. Smith. *Software Model Checking: Extracting Verification Models from Source Code*, Proceedings of FORTE/PSTV'99, Publ. Kluwer, 1999, Beijing China, 481-497

[30] Mazzocca N., A. Santone, G. Vaglini and V. Vittorini, *Efficient Model Checking of Properties of a Distributed Application: a Multimedia Case Study*, Software Testing, Verification & Reliability, **12(1)**, 2002. 3-21.

[31] McMillan K.L., "Symbolic Model Checking", Kluwer Academic Publishers, 1993.

[32] Milner R., "Communication and Concurrency", Prentice-Hall, 1989.

[33] Santone A., *Automatic Verification of Concurrent Systems using a Formula-Based Compositional Approach*, Acta Informatica, **38(2)**, 2002, 531-564.

[34] Santone A., *Heuristic Search + Local Model Checking in Selective mu-Calculus*, IEEE Transactions on Software Engineering, **29(6)**, 2003, IEEE Computer Society, California, USA, 510-523.

[35] Stirling C., *An Introduction to Modal and Temporal Logics for CCS*, In Concurrency: Theory, Language, and Architecture, Lecture Notes in Computer Science **391**, 1989.

[36] Valmari A., *A Stubborn Attack on State Explosion*, Proceedings of the Second International Conference on Computer-Aided Verification (CAV'90), Lecture Notes in Computer Science **531**, 1990, 156–165.

[37] Visser W., K. Havelund, G.P. Brat and S. Park, *Model Checking Programs*, Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00), IEEE Computer Society, 2000, 3-12.

Appendix A

```
class ProcessDrinker extends Thread {
        private Glass gl;

        public ProcessDrinker(Glass s)
        {
            gl=s;
        }

        public void run()
        {
            while(true){
            System.out.println(‘‘drinker");
                try
                {
                    gl.drink();
                }catch(Exception ee)
                {
                }
            }
        }
}

class ProcessHost extends Thread {

        private Glass gl;
        private final String name;

        public ProcessHost(Glass s, String n)
        {
            gl=s;
            name=n;
        }

        public void run()
        {
            while(true){
            System.out.println(name);
                try
                {
                    if(gl.checkEmpty())
                        gl.fill();
                }catch(Exception e)
                {
                }
            }
        }
}
```

Table 9
The Java program (I).

```
class Glass {
        private boolean is_full=false;
        private boolean is_empty=true;

        public synchronized void fill() throws Exception
        {
                is_empty=false;
                System.out.println("filling");
                is_full=true;
                System.out.println("full");
                notify();
        }

        public synchronized void drink()
        {
                while(is_empty)
                {
                    try
                    {
                        wait();
                    }catch(InterruptedException e)
                        {
                        }
                }
                System.out.println("drinking");
                is_full=false;
                is_empty=true;
                System.out.println("empty");
        }

        public synchronized boolean checkEmpty()
        {
            System.out.println("checked");
            return is_empty;
        }
    }

public class SharedGlass {
    public static void main(String [] args) throws Exception
    {
        Glass gl=new Glass();
        ProcessDrinker drinker=new ProcessDrinker(gl);
        drinker.start();
        ProcessHost host1=new Process(gl, "host1");
        host1.start();
        ProcessHost host2=new Process(gl, "host2");
        host2.start();
    }
}
```

Table 10
The Java program (II).

Appendix B

We present the CCS translation of the Java program in Table 11. Note that the print instruction of the name attribute of the ProcessHost class has been simplified by directly including visible actions (i.e. 'host1, 'host2) in the corresponding thread processes. Moreover we have omitted all the VAR_i processes.

```
proc DRINK1= callDrink1.'lock1.X11
proc X11 = isTrue1.'callWait1.returnWait1.X11+
          isFalse1.'drinking.'setFalse2.'setTrue1.'empty.'unlock1.'returnDrink1.DRINK1

proc FILL1 = callFill1.'lock1.X12
proc X12 = 'setFalse1.'filling.'setTrue2.'full.'callNotify1.
           returnNotify1.'unlock1.'returnFill1.FILL1

proc CHECKEMPTY1 = callCheckEmpty1.'lock1.X13
proc X13 = isTrue1.'setTrue3.Y13 +
           isFalse1.'setFalse3.Y13
proc Y13 = 'checked.'unlock1.'returnCheckEmpty1.CHECKEMPTY1

proc RUN2 = callRun2.WHILETrue2 proc WHILETrue2 =
'drinker.'callDrink1.returnDrink1.WHILETrue2

proc RUN3 = callRun3.WHILETrue3
proc WHILETrue3 ='host1.'callCheckEmpty1.returnCheckEmpty1.K
proc K  = isTrue3.'callFill1.returnFill1.WHILETrue3 +
          isFalse3.WHILETrue3

proc RUN4 = callRun4.WHILETrue4
proc WHILETrue4 ='host2.'callCheckEmpty1.returnCheckEmpty1.K4
proc K4  = isTrue3.'callFill1.returnFill1.WHILETrue4 +
           isFalse3.WHILETrue4

proc MAIN = 'callRun2.'callRun3.'callRun4.nil

proc FINAL =(MAIN|RUN2|RUN3|VARtt1|VARff2|VARff3|LOCK1|WAIT1|
             NOTIFY1|WAITSET10|DRINK1|FILL1|RUN4|CHECKEMPTY1|CHECKEMPTY1)\
             {isTrue1,isFalse1,setTrue1,setFalse1,isTrue2,isFalse2,setTrue2,setFalse2,
              isTrue3,isFalse3,setTrue3,setFalse3,lock1,unlock1,callWait1,returnWait1,
              callNotify1,returnNotify1,insert1,resume1,someoneInQueue1,noneInQueue1,
              callRun2,callRun3,callRun4,callDrink1,returnDrink1,callFill1,returnFill1,
              callCheckEmpty1,returnCheckEmpty1,remove1}
```

Table 11
The CCS specification.