Towards the Automated Verification of Multithreaded Java Programs

Giorgio Delzanno[1] Jean-François Raskin[2] * Laurent Van Begin[2]**

[1] Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, via Dodecaneso 35, 16146 Genova, Italy

[2] Département d'Informatique, Université Libre de Bruxelles,
Blvd Du Triomphe, 1050 Bruxelles, Belgium

Abstract. In this paper we investigate the possible application of *parameterized verification* techniques to synchronization skeletons of *multithreaded Java programs*. As conceptual contribution, we identify a class of *infinite-state* abstract models, called Multi-Transfer Nets (MTNs), that preserve the main features of the semantics of concurrent Java. We achieve this goal by exploiting an interesting connection with the Broadcast Protocols of [7], and by introducing the notion of *asynchronous rendez-vous*. As technical contribution, we extend the symbolic verification techniques of [6] based on Covering Sharing Trees and structural invariants to MTNs. As practical contribution, we report on experimental results for verification of examples of multithreaded Java programs.

1 Introduction

In the last years there has been an increasing interest in automated verification techniques for parameterized systems. Contrary to approaches based on finite-state abstractions, in parameterized verification it is possible to handle infinite-state abstract models, with a potential gain of precision in the analysis of the underlying systems. Recently, this idea has been applied to the verification of safety properties of *multithreaded C programs* [2], a natural field of application for techniques related to Petri Nets. Concurrent Java, however, is going to become the standard language for multithreaded programming. Its success is due to technology like Applets and Servlets, widely used in the context of client-server applications for the World Wide Web. For this reason, we consider the specialization of parameterized verification techniques to concurrent Java programs an important (and challenging) research goal.

In this paper we will focus on problems that we think are propedeutic to further research in this direction. Specifically, we will first address the problem of finding adequate *infinite-state abstract models* for synchronization skeletons

* This author was partially supported by a "Crédit aux chercheurs", Belgian National Fund for Scientific Research.

** Supported by a "First Europe" grant, Walloon Region, Belgium. This work was partially done when this author was visiting Università di Genova.

of concurrent Java programs. The Petri Net model adopted for C programs in [2] is no more adequate here. The problem here is due to the presence of special Java built-in methods, namely `wait`, `notify` and `notifyAll`, whose semantics cannot be given via rendez-vous communication. We will solve this problem by resorting to the following connections: the semantics of `notifyAll` (a primitive that awakens all processes waiting on a lock) can be expressed in terms of the *broadcast* primitive introduced by Emerson and Namjoshi in [7]; the semantics of `notify` (a primitive that awakens only one of the waiting processes) can be expressed via *asynchronous rendez-vous*, i.e., a rendez-vous that is *non-blocking* for the sender. We will formalize these intuitions by introducing the new model of Multi-Transfer Nets (MTNs), a formalism that incorporates and extends the main feature of Petri Nets and Broadcast Protocols.

As a second step, we will study the problem of finding an adequate technology to *efficiently* model check this new class of infinite-state systems. We will first show that the backward reachability algorithm of Esparza, Finkel and Mayr used for Broadcast Protocols in [8] can be naturally extended to MTNs. Decidability still holds for the *control state reachability* problem that consists of deciding if a state taken from a given *upward closed* set of unsafe states is reachable from the initial states. In [5], we have define a graph-based data structure called Covering Sharing Trees (CSTs) to compactly represent upward closed sets of markings of Petri Nets. In this paper we will show that the CST-based verification techniques defined in [6] can be extended to MTNs. Specifically, we will define a CST-based symbolic algorithm to compute the pre-image operator associated to an MTN, and we will apply it to build a symbolic backward reachability algorithm to check parameterized safety properties. In [5, 6] we proposed several heuristics for the analysis of Petri Nets. Most of them are based on the Structural Theory of Petri Nets, a theory that allows to statically compute over-approximations of the reachability set. Interestingly, MTNs can be viewed as a subclass of *Petri Nets with marking dependent cardinality arcs*, a class of models for which Ciardo [3] an extension of the Structural Theory of Petri Nets. As a nice consequence of this connection, we can still use the pruning techniques based on structural invariants proposed in [6] to efficiently cut the search space of an MTN during backward reachability.

As practical experiments, we have applied the extended CST-library to several parameterized safety problems taken from the literature, see e.g., [7, 8]. In this paper we will report on bechmarks performed over abstractions of multithreaded programs and we will compare the results with HyTech [11], a polyhedra-based model checker that provides backward reachability and that can handle the same class of parameterized systems.

2 Abstract Models for Multithreaded Java Programs

A Java thread is an object of the predefined Java classes `Thread` and `Runnable`. The code of a thread must be specified in the method `run` that is invoked to start its execution. Threads are executed in parallel and can share variables and

```
public class Inc extends Thread
{private Point p;
 public Inc(Point p)
   { this.p = p; }
 private void incpoint()
   { p.incx();
     p.incy(); }
 public void run()
   { while (true) incpoint(); }
}
 ....
```

```
public class Point
{private int x = 0;
 private int y = 0;
 public synchronized void incx()
   { x = x + 1;
     notifyAll(); }
 public synchronized void decx()
   { while (x == 0) wait();
     x = x - 1; }
  ...
 }
```

Fig. 1. An example of declaration of threads and synchronized methods.

objects. Every object comes with a *lock* that can be used to control concurrent accesses to its methods in a multithreaded program. Methods declared as `synchronized` compete for the lock on the corresponding instance object. Via the *synchronized* instruction, it is also possible to associate a lock to a given object. To avoid starvation and deadlocks, threads can suspend their activity and relinquish the lock on a given object while being inside a synchronized method using the `wait` primitive. Waiting processes can be awakened using the `notifyAll` primitive. Awakened processes compete for the locks they relinquish using `wait`. The primitive `notify` can be used to awake a thread arbitrarily chosen between the ones that are waiting. Both `notify` and `notifyAll` are *non-blocking*, i.e., the thread that invokes one of them does not wait for an acknowledgment.

As an example, consider the thread declarations of Fig. 1. The class `Point` provides methods to increment and decrement the coordinates of a `Point` object (the code of the methods `incy` and `decy` are omitted for brevity). The method `decx` enforces a thread to `wait` for the value of coordinate x to be *non zero*. Every time a coordinate is incremented a notification is broadcast to awake all suspended threads. Suspended threads will compete for the lock on the `Point` object. The thread `Inc` repeatedly invokes the method `incpoint` based on `incx` and `incy`. Symmetrically, the thread `Dec` (whose definition is omitted for brevity) repeatedly invokes the method `decpoint` based on `decx` and `decy`.

A possible way to use these definitions would be to declare a `main` method in which we create a `Point` object, n `Inc` threads, m `Dec` threads (all working on the same object), and then to let them run all threads in parallel. Note that, in order to ensure the consistency of the data of the shared object, increments and decrements should be performed atomically. This property should hold for *any possible number* of `Inc` and `Dec` threads, i.e. for any value of n and m. Our goal is to attack this kind of problems using *parameterized verification*. However, we first need to study adequate *infinite-state* abstract models for multithreaded Java programs.

2.1 Global Machines

Let us focus on our example and forget for a moment all synchronization primitives. Following [2], in order to extract the control skeleton of the classes `Inc` and `Dec`, we can apply the technique of *predicate abstraction*. We first unfold the methods of the `Point` class into the thread declarations. Then, we associate a boolean variable to each guard in the program (e.g. $x == 0$ will be represented via a boolean variable $zeroX$), and extrapolate the effect of the instructions on the resulting boolean program. This way, we obtain two finite-state automata. Each state of the automata corresponds to a control point in the flattened code of the methods of our threads. Method invocations can be represented using synchronization labels. Global boolean variables can be used to model the monitor that controls a shared object.

Let us now consider the synchronization skeleton of threads. Unfortunately, communication via rendez-vous cannot be used to model the operational semantics of the interplay between the built-in methods `wait`, `notify` and `notifyAll`. The type of synchronization we need here involves, in fact, a number of processes that depends on the *current global state* of the system (all processes that are *currently* waiting to be awakened). To solve this problem, we will resort to a new model, called global machines, obtained by merging concepts coming from the broadcast protocols of [7] and from the global/local machines of [2] with the new notion of *asynchronous rendez-vous*.

We start the description of global machines from the operations needed to handle global boolean variables.

Definition 1. Let $\mathbf{B} = \{b_1, \ldots, b_n\}$ be a finite set of *global boolean variables*, and let $\mathbf{B}'$ be their primed version. A *boolean guard* φ_g is either the formula *true* or the conjunction of literals $L_1 \wedge \ldots \wedge L_p$, $p \leq n$, such that L_i is either b or $\neg b$ for some $b \in \mathbf{B}$. A *boolean action* φ_a is a formula $b'_1 = v_1 \wedge \ldots \wedge b'_n = v_n$, where $v_i \in \{true, false, b_i\}$ for $i : 1, \ldots, n$.

Boolean guards and actions are used to express pre-and post-conditions on the variables in $\mathbf{B}$. The behaviour of a thread will be modeled via the notion of local machine introduced below.

Definition 2. A *local machine* is a tuple $\langle Q, \Sigma, \delta \rangle$, where: Q is a finite set of states; Σ is the set of synchronization labels used to build the set of possible actions $\mathcal{A}$ of a process (defined later); and $\delta \subseteq (Q \times \mathcal{A} \times Q)$ is the local transition relation. In the following, we will use $s \xrightarrow{\alpha} s'$ to indicate that $\langle s, \alpha, s' \rangle \in \delta$.

The *actions* of a local machines are defined as follows (in the following φ represent the conjunction of a boleean guard with an action (Def. 1), and $\ell \in \Sigma$):

- *Internal action:* $\ell : \varphi$;
- *Rendez-vous:* $\ell! : \varphi$ (sending), and $\ell?$ (reception);
- *Asynchronous Rendez-vous:* $\ell\!\uparrow\!: \varphi$ (sending), and $\ell\!\downarrow$ (reception);
- *Broadcast.* $\ell!! : \varphi$ (sending), and $\ell??$ (reception).

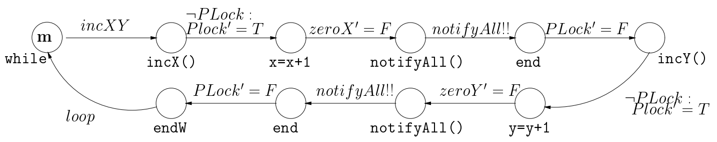

Fig. 2. The Global Machine for the `Inc` thread of Fig. 1.

Having in mind the translation from Java programs, we will also apply the following restrictions: the set of source and target states of a broadcast (asynchronous rendez-vous) reception must be distinct; broadcasts receptions associated to the same sending can be partitioned so that each partition is defined over a distinct set of states. Note, in fact, tha we will use asynchronous rendez-vous and broadcast to model the semantics of `notify` and `notifyAll`. Our restriction avoids *cyclic rules* like $sloc_1 \xrightarrow{n!!} sloc_2$, $rloc_1 \xrightarrow{n??} rloc_2$, and $rloc_2 \xrightarrow{n??} rloc_1$ that have no meaning if *sloc* and *rloc* are control points in the code of the sender and of the receiver, respectively, and n corresponds to a `notifyAll`. Furthermore, all the interesting examples of Broadcast Protocols we are aware of satisfy these conditions.

As an example, the abstract model that we extracted from thread `Inc` applying the technique of predicate abstraction can be represented via the local machine of Fig. 2.

Definition 3. A *global machine* is a tuple $\mathcal{G} = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_m, k_m \rangle \rangle$, where: $\mathbf{B} = \{b_1, \ldots, b_n\}$ is the set of *global boolean variables* for $i : 1, \ldots, m$; $\mathcal{L}_i = \langle Q_i, \Sigma_i, \delta_i \rangle$ is the i-th local machine; and k_i the number of its copies. Futhermore, we have that $Q_i \cap Q_j = \emptyset$ for any $i, j : 1, \ldots, m$ with $i \neq j$.

Definition 4. A *global state* of $\mathcal{G} = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_m, k_m \rangle \rangle$ is a tuple $G = \langle \rho, \mathbf{s} \rangle$ where $\rho : \mathbf{B} \to \{true, false\}$ is a valuation for the global boolean variables, and the tuple $\mathbf{s} = \langle s_{11}, \ldots, s_{1k_1}, \ldots, s_{m1}, \ldots, s_{mk_m} \rangle$ of dimension $k = k_1 + \ldots + k_m$ is such that $s_{ij} \in Q_i$ denotes the current local state of the j-th copy of the local machine $\mathcal{L}_i$.

The runs of a global machine are defined via the relation $\Rightarrow$ defined below.

Definition 5. Let $\mathcal{G} = \langle \mathbf{B}, \langle \mathcal{L}_1, k_1 \rangle, \ldots, \langle \mathcal{L}_m, k_m \rangle \rangle$ be a global machine, $G = \langle \rho, \langle s_1 \ldots, s_k \rangle \rangle$ and $G = \langle \rho', \langle s'_1 \ldots, s'_k \rangle \rangle$ be two global states, and $\gamma = \rho \cup \rho'$. Then, $G \stackrel{\ell}{\Longrightarrow}_{\mathcal{G}} G'$ iff one of the following conditions holds:

- there exist i and u such that $s_i \xrightarrow{\ell:\varphi} u$, and $\gamma(\varphi) = true$, $s'_i = u$, and $s'_j = s_j$ for all $j \neq i$.
- there exist i, j, u and v such that $s_i \xrightarrow{\ell!:\varphi} u$, $s_j \xrightarrow{\ell?} v$, $\gamma(\varphi) = true$, $s'_i = u$, $s'_j = v$, and $s'_r = s_r$ for all $r \neq i, j$.
- there exist i and u such that $s_i \xrightarrow{\ell\uparrow:\varphi} u$, and $\gamma(\varphi) = true$, $s'_i = u$, and: either there exist j and v such that $s_j \xrightarrow{\ell\downarrow} v$, $s'_j = v$ and $s'_r = s_r$ for any $r \neq i, j$,

or there are no j and v such that $s_j \xrightarrow{\ell\downarrow} v$ is defined, and $s'_r = s_r$ for any $r \neq i$.
- there exist i and u such that $s_i \xrightarrow{\ell!!:\varphi} u$, $\gamma(\varphi) = true$, $s'_i = u$, and for all j such that there exist v and $s_j \xrightarrow{\ell??} v$, we have $s'_j = v$; finally, $s'_r = s_r$ for all $r \neq i$ such that $\ell??$ is not defined in s_r.

A *run* of a global machine is a sequence of global states $G_0 G_1 \ldots G_n$ such that $G_i \xRightarrow{\ell}_{\mathcal{G}} G_{i+1}$ for $0 \leq i < n$. G_0 is the *initial* global state of the run and G_n is the *target* global state of the run. A global state G' is reachable from G, written $G \xRightarrow{*}_{\mathcal{G}} G'$, if and only if there exists a run with initial global state G and target global state G'. The set of reachable global states from a set of initial global states $\mathbf{G}$, noted $\mathsf{Reach}_{\mathcal{G}}(\mathbf{G})$, is equal to $\{G' \mid \exists G \in \mathbf{G} : G \xRightarrow{*}_{\mathcal{G}} G'\}$.

3 Multi-Transfer Nets

Following [8, 10], in order to study safety properties of global machines, we will apply a *counting* abstraction that maps global states into markings (of a Petri Net model) that keep track of the number of processes in each one of the local states of the local machines. To be able to model the communication mechanisms of Def. 2, we need however an extended Petri Net-like model, we will call Multi-Transfer Nets (MTNs). MTNs have all the features of Petri Nets. In addition, MTNs allow us to capture the semantics of rendez-vous, asynchronous rendez-vous and of broadcast as an instance of a general notion of *transfer of at most c tokens*. Formally, this model is defined as follows.

Definition 6 (Multi-Transfer Net). A *Multi-Transfer Net* is a pair $\langle \mathcal{P}, \mathcal{B} \rangle$ where: $\mathcal{P} = \{p_1, \ldots, p_n\}$ is a set of places, and $\mathcal{B} = \{M_1, \ldots, M_m\}$ is a set of *multi-transfers*.

A *multi-transfer* M is a tuple $\langle T, \{S_1, \ldots, S_u\} \rangle$ such that

- $T = \langle \mathcal{I}, \mathcal{O} \rangle$ is the Petri Net *transition* of the *multi-transfer*, i.e. $\mathcal{I}, \mathcal{O} : \mathcal{P} \to \mathbb{N}$;
- each $S_k = \langle \{B_{k1}, \ldots, B_{kr_k}\}, c_k \rangle$ is a *transfer block*, where $c_k \in \mathbb{N} \cup \{+\infty\}$ is the *bound*, and each $B_{kj} = \langle P_{kj}, p_{kj} \rangle$ with $P_{kj} \subseteq \mathcal{P}$ and $p_{kj} \in \mathcal{P}$ is a *transfer*.

In order to avoid cyclic transfers, a multi-transfer M with set of transfer blocks $\{S_1, \ldots, S_u\}$ must satisfy the following conditions:

1. for any *transfer block* S_k with set of *transfers* $\{B_{k1}, \ldots, B_{kr_k}\}$, and for any such B_{kj}, we require that $p_{kj} \notin P_{kj}$;
2. for any *transfer* B_{ki} in the transfer block S_k and B_{lj} in the transfer block S_l with $B_{ki} \neq B_{lj}$, we require that $(P_{ki} \cup \{p_{ki}\}) \cap (P_{lj} \cup \{p_{lj}\}) = \emptyset$.

A *marking* is a mapping $\mathbf{m} : \mathcal{P} \to \mathbb{N}$ (a vector of natural numbers). Given $\mathcal{I} : \mathcal{P} \to \mathbb{N}$, we use $\mathcal{I} \geq \mathbf{m}$ to indicate that $\mathcal{I}(p) \geq \mathbf{m}(p)$ for all $p \in \mathcal{P}$. Furthermore, given $S \subseteq \mathcal{P}$ we define $\mathbf{m}(S) = \Sigma_{p \in S} \mathbf{m}(p)$.

Definition 7 (Enabling a Multi-Transfer). Let M be a multi-transfer with *transition* $\langle \mathcal{I}, \mathcal{O} \rangle$. We say that M is *enabled* in $\mathbf{m}$ if $\mathcal{I} \geq \mathbf{m}$.

Definition 8 (Firing a Multi-Transfer). Let $M = \langle T, \{S_1, \ldots, S_u\} \rangle$ be a multi-transfer enabled in $\mathbf{m}$. *Firing* M in $\mathbf{m}$ leads to any marking $\mathbf{m}'$ (written $\mathbf{m} \rightarrowtail_M \mathbf{m}'$) computed in accord to the following *sequence* of steps (in which we use the two intermediate markings $\mathbf{m}_1$ and $\mathbf{m}_2$):

1. let $T = \langle \mathcal{I}, \mathcal{O} \rangle$, $\mathbf{m}_1(p) = \mathbf{m}(p) - \mathcal{I}(p)$ for all $p \in \mathcal{P}$;
2. $\mathbf{m}_2$ can be any marking such that the following constraints are satisfied:
   – for each transfer block $S_k = \langle \{B_{k1}, \ldots, B_{kr_k}\}, c_k \rangle$:
     - if $r_k > 0$ and $\mathbf{m}_1(P_{k1} \cup \ldots \cup P_{kr_k}) > c_k$, then for all $d_1, \ldots, d_{r_k} \in \mathbb{N}$ be such that $d_1 + \ldots + d_{r_k} = c_k$ and for all j, $1 \leq j \leq r_k$:
       - $\mathbf{m}_2(p_{kj}) = \mathbf{m}_1(p_{kj}) + d_j$,
       - and $\mathbf{m}_2(P_{kj}) = \mathbf{m}_1(P_{kj}) - d_j$,
       - with the additional constraint that $\mathbf{m}_1(p) \geq \mathbf{m}_2(p)$ for any $p \in P_{kj}$.
     - if $r_k > 0$ and $\mathbf{m}_1(P_{k1} \cup \ldots \cup P_{kr_k}) \leq c_k$, then for all j, $1 \leq j \leq r_k$:
       - $\mathbf{m}_2(p_{kj}) = \mathbf{m}_1(p_{kj}) + \mathbf{m}_1(P_{kj})$ and
       - $\mathbf{m}_2(P_{kj}) = 0$;
   – $\mathbf{m}_2(p) = \mathbf{m}_1(p)$ for all $p \in \mathcal{P}$ not involved in transfer blocks;
3. $\mathbf{m}'(p) = \mathbf{m}_2(p) + \mathcal{O}(p)$ for all $p \in \mathcal{P}$.

Note that a *Petri Net transition* is obtained by considering multi-transfers with an empty set of transfer blocks. A *transfer arc* (all tokens in the sources are transferred to the target) is obtained instead by a multi-transfer with a single set of transfers with bound $c = +\infty$. A transfer block has a *non-deterministic* effect if the total number of tokens is strictly greater than c with $c \in \mathbb{N}$. The non-determinism is due to the numbers $d_1, \ldots, d_{r_k}$ of tokens that are transferred by each transfer, and from their distribution within the set P_{kj} of sources for $j : 1, \ldots, r_k$.

Definition 9 (Operational Semantics). Let $\mathcal{M} = \langle \mathcal{P}, \mathcal{B} \rangle$ be an MTN. A *run* of $\mathcal{M}$ is a sequence of markings $\mathbf{m}_0\mathbf{m}_1 \ldots \mathbf{m}_n$ such that for any $i, 0 \leq i < n$, there exists $M \in \mathcal{B}$ such that $\mathbf{m}_i \rightarrowtail_M \mathbf{m}_{i+1}$, $\mathbf{m}_0$ is the *initial* marking of the run and $\mathbf{m}_n$ the *target* marking of the run. A marking $\mathbf{m}'$ is *reachable* from a marking $\mathbf{m}$, written $\mathbf{m} \rightarrowtail^* \mathbf{m}'$, if and only if there exists a run with initial marking $\mathbf{m}_0$ and target marking $\mathbf{m}'$. The *set of reachable markings* of $\mathcal{M}$ from a set of markings I, written $\mathsf{Reach}_{\mathcal{M}}(I)$, is defined as the set $\{\mathbf{m}' \mid \exists \mathbf{m} \in I : \mathbf{m} \rightarrowtail^* \mathbf{m}'\}$.

3.1 From Global Machines to MTNs

Global machines can be naturally translated into MTNs, as we will informally explain in this section. In the following we will often refer to Fig. 3 to illustrate the intuition behind the translation. First of all, each *global boolean variable* b is modeled in the MTN by the two places T_b and F_b. Each local state s_{ij} appearing in the *local machine* $\mathcal{L}_i$ is modelled with a place with the same name

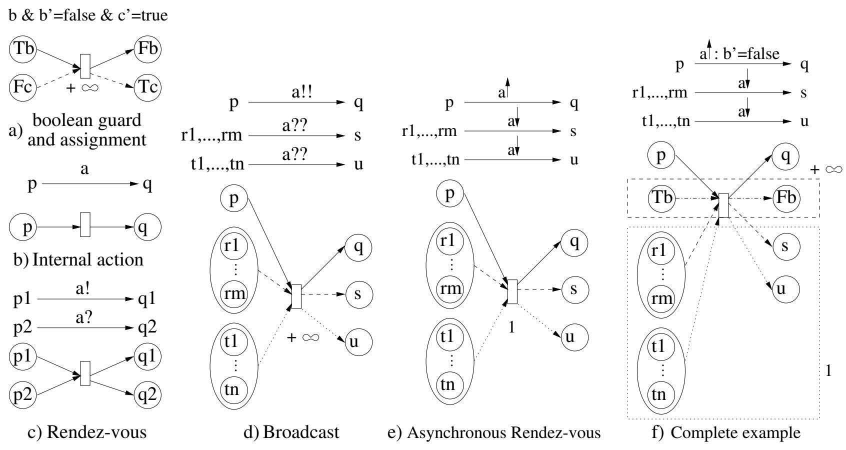

Fig. 3. From global machines rules to MTNs. $r_1, \ldots, r_n \xrightarrow{\ell??} s$ is an abbreviation for the set of transitions $r_1 \xrightarrow{\ell??} s$, ..., $r_n \xrightarrow{\ell??} s$, all of them having the same destination state s, similarly for $r_1, \ldots, r_n \xrightarrow{\ell\downarrow} s$.

s_{ij}. A global state $G = \langle \rho, \mathbf{s} \rangle$ is translated into the marking $\mathbf{m}_G$ such that: (1) for each $b \in B$, if $\rho(b) = true$ then $\mathbf{m}_G(T_b) = 1$ and $\mathbf{m}_G(F_b) = 0$, if $\rho(b) = false$ then $\mathbf{m}_G(T_b) = 0$ and $\mathbf{m}_G(F_b) = 1$; (2) for each local state s_{ij}, $\mathbf{m}_G(s_{ij}) = v_{ij}$, where v_{ij} is the number of occurrences of state s_{ij} in $\mathbf{s}$. In other words, the place s_{ij} contains as many tokens as the number of copies of the local machine $\mathcal{L}_i$ whose current state is s_{ij}. Let us now briefly explain how an action is translated into a *multi-transfer*. We first focus our attention on the boolean part of actions. Consider the boolean formula $\varphi = b \wedge b' = false \wedge c' = true$ with $d' = d$ for all other variables. Let M_φ be the multi-transfer that results from the translation of φ. Intuitively, M_φ has to check the presence of one token in place T_b, and ensure that, when the transition is taken, the token is removed from T_b and added to F_b. Furthermore, the only token shared by T_c and F_c must be in F_c after the firing. This can be achieved as shown in Fig. 3(a). Using the Petri Net transition of M_φ (thick lines in in Fig. 3), we ensure the presence of the token in T_b before the firing and in F_b after. To ensure the presence of a token in T_c after firing, we use an unbounded transfer block with a single transfer from the singleton $\{F_c\}$ to the place T_c (dashed lines in Fig. 3). If the token was already in T_c before the firing, the transfer block has no effect, on the other hand if the token was in F_c then it is transfered to T_c.

Starting from the previous idea, we incrementally add new components to M_φ so as to completely action of a global machine. An *internal action* a from state p to state q is simply modeled by adding an input arc from place p and an output arc to place q to the transition of the transfer (see Fig. 3(b)). A *rendez-vous* is treated similarly but with pairs of places (see Fig. 3(c)). A *broadcast* sending is modelled as an internal action. *Broadcast* receptions are modeled via an unbounded transfer block that contains a transfer for each possible destination state (see Fig. 3(d)).

Finally, an *asynchronous rendez-vous* sending is modeled as an internal action, whereas the corresponding receptions are modeled via a transfer block with bound 1 that contains a transfer for each possible destination state (see Fig. 3(e)). A complete example treating a boolean formula and asynchronous rendez-vous is given in Fig. 3(f). The asynchronous rendez-vous update the boolean formula b to false. The boolean part is modeled by an unbounded transfer block with one single transfer, the sender part by the petri net transition and the receivers part by a 1-bounded transfer block with two transfers. The following proposition formally relates global machines and MTNs.

Proposition 1. For any global machine $\mathcal{G}$ and any set of global states $\mathbf{G}$, we can construct automatically a MTN $\mathcal{M}_{\mathcal{G}}$ with only unbounded and 1-bounded multi-transfers such that $\alpha(\mathsf{Reach}_{\mathcal{G}}(\mathbf{G})) = \mathsf{Reach}_{\mathcal{M}_{\mathcal{G}}}(\alpha(\mathbf{G}))$.

4 Verification of MTN-based Abstract Models

It is well-known that the class of safety properties of Petri Nets whose negation can be expressed in terms of upward closed sets of markings can be decided using backward reachability [1, 9]. In this setting the goal is to prove that none of the markings in a given infinite set $\mathbf{U}$ of *unsafe configurations* can be reached from the set of initial markings $\mathbf{M}_0$. To achieve this goal, we can first compute the transitive closure of the *pre-image* operator, and then check that no elements of $\mathbf{M}_0$ is in the resulting set of markings. As shown in [1, 8, 9], this algorithm is guaranteed to terminate for Petri Nets and Broadcast Protocols whenever $\mathbf{U}$ is upward closed w.r.t. the componentwise ordering of tuples. Formally, let $cones(\mathbf{S}) = \{\mathbf{m}' \mid \mathbf{m} \preccurlyeq \mathbf{m}',\ \mathbf{m} \in \mathbf{S}\}$. Then, a set of markings $\mathbf{S}$ is upward closed if $cones(\mathbf{S}) = \mathbf{S}$. An upward closed set of markings $\mathbf{U}$ is always finitely generated by a set of minimal tuples, we will denote it as $gen(\mathbf{U})$. The termination of the algorithm is ensured by the following properties. The application of the pre-image operator (associated to Petri Nets and Broadcast Protocols) to an upward closed set of markings returns a set that is still upward closed. The containment relation between upward closed sets of markings is a *well-quasi ordering*. It is important to note that Karp-Miller's construction may fail to terminate for extensions of Petri Nets with broadcast communication [8].

In order to extend the algorithm of [8], we first need to study the properties of the pre-image operator of MTNs.

Definition 10 (Pre-image of an MTN). Let $\mathcal{M} = \langle \mathcal{P}, \mathcal{B} \rangle$ be an MTN, and let $M \in \mathcal{B}$, then $\mathsf{Pre}_M(\mathbf{S}) = \{\mathbf{m}' \mid \mathbf{m}' \rightarrowtail_M \mathbf{m},\ \mathbf{m} \in \mathbf{S}\}$.

We can easily prove that MTNs are monotonic with respect to the pointwise ordering on markings, i.e., if $\mathbf{m}_1 \rightarrowtail \mathbf{m}_2$, then for any $\mathbf{m}_1' \geq \mathbf{m}_1$ there exists $\mathbf{m}_2' \geq \mathbf{m}_2$ such that $\mathbf{m}_1' \rightarrowtail \mathbf{m}_2'$.

Proposition 2. Let $\mathcal{M} = \langle \mathcal{P}, \mathcal{B} \rangle$ be an MTN, and $M \in \mathcal{B}$. If $\mathbf{S}$ is an upward closed set of markings, then $\mathsf{Pre}_M(\mathbf{S})$ is still upward closed.

As a consequence, backward reachability for MTNs is guaranteed to terminate when taking an upward closed sets of markings as input. In the next section we will exploit the previous property to define a CST-based symbolic backward reachability for MTNs.

5 The Assertional Language of Covering Sharing Trees

Covering Sharing Trees (CSTs) are an extension of the sharing tree data structure introduced in [13] to efficiently store tuples of integers. A sharing tree $\mathcal{S}$ is a rooted acyclic graph with nodes partitioned in k-*layers* such that: all nodes of layer i have successors in the layer $i+1$; a node cannot have two successors with the same label; finally, two nodes with the same label in the same layer do not have the same set of successors. Formally, $\mathcal{S}$ is a tuple $(N, V, root, end, val, succ)$, where $N = \{root\} \cup N_1 \cup \ldots \cup N_k \cup \{end\}$ is the finite set of *nodes* (N_i is the set of nodes of *layer* i and, by convention, $N_0 = \{root\}$ and $N_{k+1} = \{end\}$), $V = \{x_1, x_2, \ldots, x_k\}$ is a set of variables. Intuitively, N_i is associated to x_i. $val : N \rightsquigarrow \mathbb{Z} \cup \{\top, \bot\}$ is a labeling function for the nodes, and $succ : N \rightsquigarrow 2^N$ defines the successors of a node. Furthermore, (1) $val(n) = \top$ iff $n = root$, $val(n) = \bot$ iff $n = end$, $succ(end)=\emptyset$; (2) for $i : 0, \ldots, k$, $\forall n \in N_i$, $succ(n) \subseteq N_{i+1}$ and $succ(n) \neq \emptyset$; (3) $\forall n \in N$, $\forall\, n_1, n_2 \in succ(n)$, if $n_1 \neq n_2$ then $val(n_1) \neq val(n_2)$. (4) $\forall i, 0 \leq i \leq k$, $\forall n_1, n_2 \in N_i$, $n_1 \neq n_2$, if $val(n_1) = val(n_2)$ then $succ(n_1) \neq succ(n_2)$. A path of a k-sharing tree is a sequence of nodes $\langle n_1, \ldots, n_m \rangle$ such that $n_{i+1} \in succ(n_i)$ for $i = 1, \ldots, m\text{-}1$. Paths represent *tuples of size* k of natural numbers. We use $elem(\mathcal{S})$ to denote the *flat denotation* of a k-sharing tree $\mathcal{S}$:

$$elem(\mathcal{S}) = \{\ \langle val(n_1), \ldots, val(n_k) \rangle \mid \langle \top, n_1, \ldots, n_k, \bot \rangle \text{ is a path of } \mathcal{S}\ \}.$$

Conditions (3) and (4) ensure the maximal sharing of prefixes and suffixes among the tuples of the flat denotation of a sharing tree. The *size* of a sharing tree is the number of its *nodes* and *edges*. The number of tuples in $elem(\mathcal{S})$ can be exponentially larger than the size of $\mathcal{S}$. As shown in [13], given a set of tuples $\mathcal{A}$ of size k, there exists a unique sharing tree such that $elem(\mathcal{S}_{\mathcal{A}}) = \mathcal{A}$ (modulo isomorphisms of graphs). A CST is obtained by lifting the denotation of a sharing tree $\mathcal{S}$ as follows

$$cones(\mathcal{S}) = \{\mathbf{m} \mid \mathbf{n} \preccurlyeq \mathbf{m},\ \mathbf{n} \in elem(\mathcal{S})\}.$$

($cone(\mathbf{m})$ is defined in a similar way on a single marking $\mathbf{m}$). Given an upward closed set of markings $\mathbf{U}$, we define the CST $\mathcal{S}_{\mathbf{U}}$ as the k-sharing tree such that $elem(\mathcal{S}_{\mathbf{U}}) = gen(\mathbf{U})$. Thus, $\mathcal{S}_{\mathbf{U}}$ can be used to *compactly* represent $gen(\mathbf{U})$ (in the best case the size of $\mathcal{S}_{\mathbf{U}}$ is *logarithmic* in the size of $gen(\mathbf{U})$) and to *finitely* represent $\mathbf{U}$. A CST can also be viewed as a compact representation of the formula $\vee_{\mathbf{m} \in gen(\mathbf{U})}\ (x_1 \geq m_1 \wedge \ldots \wedge x_n \geq m_n)$. An examples of CST is given in the of Fig. 5(a).

function Step2 ($\mathcal{S}$ *: CST after step* (1), P *: source*, p_k *: target*) **return** R
$\quad\mathcal{R} \longleftarrow Empty_{CST}$
$\quad$**forall** *value* v *in the layer associated to place* p_k **do**
$\quad\quad\mathcal{S}_v \longleftarrow \mathcal{S}$ such that $elem(\mathcal{S}) = elem(\mathcal{S}_v) \setminus \{\langle c_1, c_2, \ldots, c_n\rangle | c_k \neq v\}$
$\quad\quad$**forall** *layers of* $\mathcal{S}_v$ *corresponding to places* $P \cup \{p_k\}$ **do**
$\quad\quad\quad$replace all the nodes n of the current layer by the set of nodes
$\quad\quad\quad\{n_0, n_1, \ldots, n_v\}$ having the same successors and predecessors
$\quad\quad\quad$than n and such that $val(n_i) = i$
$\quad\quad$Apply algorithm of [13] on $\mathcal{S}_v$ to ensure conditions (3)-(4)
$\quad\quad$Compute $\mathcal{Q}_v$ such that $m \in cones(\mathcal{Q}_v)$ iff $\sum_{p \in P \cup \{p_k\}} m(p) \geq v$
$\quad\quad\mathcal{T}_v \longleftarrow \mathcal{S}_v \cap_{CST} \mathcal{Q}_v$
$\quad\quad\mathcal{R} \longleftarrow \mathcal{R} \cup_{CST} \mathcal{T}_v$

Fig. 4. Algorithm for Step (2).

6 CST-based Symbolic *Pre* Operator for MTNs

In the context of Petri Nets, in [5], we presented an algorithm to symbolically compute the pre-image of a set of markings represented via a CST. In this section we will restrict ourselves to consider *transfer blocks* having bound $+\infty$. The algorithm can be extended however to any bound $c \in \mathbb{N}$, and in particular to $c = 1$. Let us first note that the Pre operator enjoys the following properties.

Remark 1. Consider the multi-transfer $M = \langle T, \{S_1, \ldots, S_k\}\rangle$ where $T = \langle \mathcal{I}, \mathcal{O}\rangle$. We define $\mathsf{Pre}_{\mathcal{I}}$, $\mathsf{Pre}_{\mathcal{O}}$, and Pre_{S_i} as the pre-image operator associated to the multi-transfers $M_{\mathcal{I}} = \langle\langle \mathcal{I}, \emptyset\rangle, \emptyset\rangle$, $M_{\mathcal{O}} = \langle\langle \emptyset, \mathcal{O}\rangle, \emptyset\rangle$, and $M_i = \langle\langle \emptyset, \emptyset\rangle, \{S_i\}\rangle$, respectively. From Def. 6, we have that $\mathsf{Pre}_M = \mathsf{Pre}_{\mathcal{I}} \circ \mathsf{Pre}_{S_1} \circ \ldots \circ \mathsf{Pre}_{S_k} \circ \mathsf{Pre}_{\mathcal{O}}$. Furthermore, let S be the transfer block $\langle\{B_1, \ldots, B_r\}, +\infty\rangle$, then $\mathsf{Pre}_S = \mathsf{Pre}_{B_1} \circ \ldots \circ \mathsf{Pre}_{B_r}$, where Pre_{B_i} is associated to the multi-transfer $N_i = \langle T', \{S'_i\}\rangle$ such that $T' = \langle \emptyset, \emptyset\rangle$ and $S'_i = \langle B_i, c\rangle$. The previous properties hold because transfers are defined on distinct set of places each other.

Based on the previous remark, let us consider then a *transfer block* M_B with bound $+\infty$ and with the single *transfer* $B = \langle P, p_k\rangle$, $P \subseteq \mathcal{P}$, and $p_k \in \mathcal{P}$. Furthermore, let $I_P = \{\ i \mid p_i \in P\ \}$ be the set of indexes of places in P. Given a CST $\mathcal{S}$, our aim is to build an algorithm to construct a CST $\mathcal{S}'$ such that $cones(\mathcal{S}') = \mathsf{Pre}_{M_B}(cones(\mathcal{S}))$. We proceed in two steps.

The first step consists in removing all paths of $\mathcal{S}$ that do not satisfy the the post-condition induced by the semantics of transfer blocks with bound $+\infty$: *all source places in* P *must contain zero tokens after firing* M_B. Specifically, we compute the CST $\mathcal{S}_1$ such that

$$elem(\mathcal{S}_1) \ = \ \{\langle c_1, c_2, \ldots, c_n\rangle \in elem(\mathcal{S}) \mid c_i = 0,\ for\ any\ i \in I_P\}.$$

By construction, it follows that $\mathsf{Pre}_{M_B}(cones(\mathcal{S}_1)) = \mathsf{Pre}_{M_B}(cones(\mathcal{S}))$. This step can be performed in *polynomial time* in the size of $\mathcal{S}$: we simply have to remove all nodes n of the layers associated to places in P such that $val(n) \neq 0$. This will give us what is called a *pre*-Sharing Tree in [13], a graph in which condition (4) of the definition of sharing tree might be violated. By applying the algorithm described in [13], the *pre*-Sharing Tree can be re-arranged into a Sharing Tree in polynomial time.

As a second step, we compute the predecessors of the elements of $\mathcal{S}_1$, w.r.t. M_B (recall that M_B has only the *transfer* $B = \langle P, p_k \rangle$). Suppose $\mathbf{c} = \langle c_1, \ldots, c_n \rangle \in elem(\mathcal{S}_1)$. Then, we know that the tuple $\mathbf{c}$ represents the upward-closed set of markings $\mathbf{S} = \{\mathbf{m} \mid \mathbf{m}(p_i) \geq c_i,\ i : 1, \ldots, n\}$. Applying Pre_{M_B} to $\mathbf{c}$, we should obtain a representation of the upward-closed set $\mathbf{S}'$ whose markings present one possible distribution of tokens *before* the transfer from P to p_k took place. Note that the relation between the number of tokens in P (say $\sum_{i \in I_P} x_i$) and in p_k (say x_k) before and after firing M_B is as follows: $x'_k = x_k + \sum_{i \in I_P} x_i$ and $x'_i = 0$ for any $i \in I_P$. Thus, $\mathbf{S}'$ will be generated by the set $gen(\mathbf{S}')$ consisting of the markings $\mathbf{d} = \langle d_1, d_2, \ldots, d_n \rangle$ having the following properties: $d_k + \sum_{i \in I_P} d_i = c_k$; whereas $d_j = c_j$ for any $j \notin (I_P \cup \{k\})$. Intuitively, all we need here is to forget about the labels of the nodes of $\mathcal{S}_1$ associated to the places in $P \cup \{p_k\}$, and replace them with nodes so that the sum of the values (associated to $P \cup \{p_k\}$) along a path always gives c_k.

For instance, consider a transfer $\langle \{p_1\}, p_2 \rangle$, and let $c_2 = 2$ be the constant in the constraint associated to p_2 in $\mathcal{S}_1$. Furthermore, suppose that p_1 and p_2 are associated to adjacent layers in our CST. Then, we simply have to add the labels $0, 1, 2$ in both layers, and then connect the resulting nodes so that the sum of the connected values is always equal to 2.

In the general case, we also have to take into account places stored in non-adjacent layers, consider more than one value for c_k, etc. To attack these problems, we split the algorithm in two sub-steps. For each possible value of c_k in $\mathcal{S}_1$, we first compute an over-approximation (see the example below), and then, we select the exact paths by intersecting the resulting CST with a CST whose generators are the markings $\mathbf{m} = \langle m_1, \ldots, m_n \rangle$ in which $m_k + \sum_{i \in I_P} m_i = c_k$ and $m_j = 0$ for $j \notin I_P \cup \{k\}$. The complete algorithm is given in Fig. 4. The following example will help in understanding this technique.

As an example, consider the CST $\mathcal{S}$ in Fig. 5(a) consisting of the three elements $\langle 0, 2, 0, 1, 0, 0 \rangle$, $\langle 0, 1, 0, 2, 0, 0 \rangle$ and $\langle 0, 0, 0, 2, 2, 0 \rangle$ representing the formula $\Phi = (c_2 \geq 2 \wedge c_4 \geq 1) \vee (c_2 \geq 1 \wedge c_4 \geq 2) \vee (c_4 \geq 2 \wedge c_5 \geq 2)$. Now consider the transfer that moves the tokens from place p_5 into place p_2, defined through the equation $c'_2 = c_2 + c_5, c'_5 = 0$. When applied to the CST (a) the algorithm of Fig. 4 performs the steps shown in (b-g). Specifically, it first computes (b) by removing the tuple $\langle 0, 0, 0, 2, 2, 0 \rangle$ that do not satisfy $c'_5 = 0$. At the second iteration of the loop (line 3, Fig.4), it computes (c). By adding new nodes in the second and fifth layers, we obtain (d), an over-approximation of the backward reachable markings starting from (c). The CST $\mathcal{Q}_2$ representing all the tuples satisfying $c_2 + c_5 \geq 2$ is shown in (e). The CST resulting from the intersection of (d) and $\mathcal{Q}_2$ corresponding to the exact set of backward reachable markings starting from (c) is shown in (f). Finally, the result of the algorithm is the CST (g).

Structural Heuristics The MTNs that results from the translations from global machines enjoy the following interesting property. They can be viewed as *Petri*

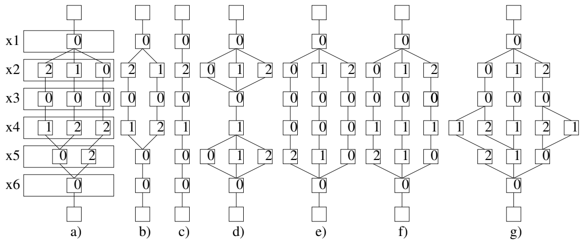

Fig. 5. Set of CST generated during the computation of the transfer $c_2' = c_2 + c_5, c_5' = 0$

Nets with arc dependent cardinality arcs [3], an extension of Petri Nets in which edges are labeled with functions linear in the current marking. Ciardo has shown that *place invariants* for *Petri Nets with arc dependent cardinality arcs* can be computed by first reducing them to Petri Nets (Theorem 1 of [3]). When specialized to a given initial marking, place invariants can be used to infer structural invariants that must hold in all reachable markings (i.e. they represent an over-approximation of the reachability set). Structural invariants can then be used to prune the search space explored during backward reachability. On the basis of the previous observation, we can enhance the MTN-backward reachability algorithm with the efficient pruning techniques working on CSTs we proposed in [6]. In this technique we use the information coming from the *statically computed* invariants as follows. At each step during the search we remove paths of the CST representing the partial search space that do not satisfy the invariant. This operations can be efficiently performed by working directly on the structure of a CST. In the next section we will report on practical experiments we obtained with the resulting method.

7 Experimental results

The table in Fig. 6 describes some of the results we obtained by applying the extended CSTs-library to examples of MTNs modeling multithreaded programs. As an example, we have tested the MTN corresponding to the global machine partially described in Fig. 2. We considered the following unsafe set of states: at least two threads have modified only one of the two coordinates of the `Point` object. This *upward closed set* of states can be represented by CSTs in which either at least two tokens are in the place corresponding to the `incY()` state of Fig. 2, or at least one token is in state `incX()` and at least one token is in state `decY()`, or at least two tokens are in state `decY()`. For this example, we have automatically computed *place invariants*, and we have applied them to prune the search as shown in Fig. 6 (I/D). Here m and n represent the number of `Inc` and `Dec` threads in the initial marking. We considered both *parametric initial markings* (e.g. $m \geq 1, n \geq 1$) as well as fixed initial markings (e.g. $m = 1, n = 1$). In the second case the idea of pruning the search space via structural invariants

MTN	$\mathbf{m}_0$	P	MT	I	Err	S	NN	NE	ETC	ETH	R
I/D	$m,n \geq 1$	32	28		√	10	1542	1823	9.77s	↑	–
I/D	$m,n \geq 1$	32	28	√	√	10	538	209	0.82s	40.24s	49
I/D	$m,n=1$	32	28	√	√	10	279	68	0.22s	5.07s	23
P/C	$m,n=2$	18	14			25	860	12264	44.49s	↑	–
P/C	$m,n=2$	18	14	√		8	111	43	0.04s	0.46s	11.5
P/C	$m,n=50$	18	14	√		152	8511	179451	1h36m	↑	–
P/C_1	$k,l,m,n \geq 1$	44	37		√	14	20421	15543	37m55s	↑	–
P/C_1	$k,l,m,n \geq 1$	44	37	√	√	14	1710	1384	2.86s	↑	–
P/C_1	$k,l,m,n=1$	44	37	√	√	14	1231	736	1.88s	↑	–
P/C_2	$k,l,m,n \geq 1$	44	37			29	12479	8396	55m50s	↑	–
P/C_2	$k,l,m,n \geq 1$	44	37	√		1	46	1	0.02s	1.73s	86.5
P/C_2	$k,l,m,n=1$	44	37	√		1	46	1	0.00s	0.48s	> 48

Fig. 6. Benchmarks on an AMD Athlon 900Mhz 500Mbytes: $\mathbf{m}_0$=initial marking, **P**=n. places, **MT**=n. multi-transfers; **I**=pruning via invariants; **Err**=bug found; **S**=n. iterations before reaching the fixpoint/finding a bug; **NN**=nodes of the CST-fixpoint; **NE**=paths in the CST-fixpoint; **ETC**=ex. time using CSTs; **ETH**=ex. time using HyTech (↑ indicates that HyTech ran out of memory); **R**=**ETH/ETC**.

is much more effective (see [6]). In all cases our tool found a *potential bug* (see Fig.6), that (after looking at the abstract trace) turns out to be a mistake in the Java program. In fact, though the primitive methods `incx`, `incy`, `decx`, and `decy` are protected by a monitor (they are declared as *synchronized*), the derived methods `incpoint` and `decpoint` are not. Thus, increments(decrements) on pair of coordinates are not executed atomically. To correct the error, we can declare the derived methods *synchronized*, too. This way mutual exclusion is automatically guaranteed by the semantics of Java.

We have also analyzed the Producer-Consumer example of [4] (P/C in Fig. 6) and a modified version of it (P/C_1 in Fig. 6) built as follows: We introduced new class declarations for *malicious* producers and consumers, and we artifically inserted the possibility of violating mutual exclusion. In this example the presence of violations depends on the values of boolean variables. Our tool finds the bug after 14 iterations (see Fig. 6). We also managed to verify the corrected version in less than one second (P/C_2 in Fig. 6). As shown in Fig. 6, we ran the same examples (using the same invariants) on HyTech [11]. Our execution times are always better. Furthermore, in some case HyTech ran out of memory before reaching a fixpoint or detecting the presence of the initial state.

8 Conclusions and Related Works

In this work we focused on the following points. Via a connection with previous works on parameterized verification [7, 8], we have shown that there exists an extension of Petri Nets that captures the essential features of the concurrent model of Java. For this class, we can use *decision* procedures to automatically verify safety properties for arbitrary number of threads. This goal is achieved by extending the CST-based symbolic model checking algorithm previously defined for Petri Nets.

The use of parameterized verification via backward reachability is the main novelty of this approach over other approaches to software verification via *finite models* (see e.g. [4]) or with Petri Nets like model [2]. The verification approach of [2] is based on the Karp-Miller's coverability tree that amount to forward reachability with accelerations (see also [7]). Contrary to backward reachability, the automated construction of Karp-Miller coverability tree is not guaranteed to terminate for extensions of Petri Nets with transfer arcs as shown by the counterexample of Esparza, Finkel, and Mayr [8].

Our preliminary analysis of the problem, tune of the techniques, and experimental results indicate the potential interest of a second research phase aimed at producing automatically *infinite-state* skeletons of Java programs, a task that is in an advanced stage in the *finite*-case verification approach. This will be one of our main future directions of research. As we explain in the paper, our techniques find other interesting applications for the automated verification of Broadcast Protocols. Concerning this point, we are not aware of other tools designed to attack *symbolic state explosion* for this class of extended Petri Nets.

References

1. P. A. Abdulla, K. Cerāns, B. Jonsson and Y.-K. Tsay. General Decidability Theorems for Infinite-State Systems. In *Proc. LICS'96*, pages 313–321, 1996.
2. T. Ball, S. Chaki, S. K. Rajamani. Parameterized Verification of Multithreaded Software Libraries. In *Proc. TACAS'01*, LNCS 2031, pages 158-173, 2001.
3. G. Ciardo. Petri Nets with marking-dependent arc multiplicity: properties and analysis. In *Proc. ICATPN'94*, LNCS 815, pages 179-198, 1994.
4. J. C. Corbett. Constructing Compact Models of Concurrent Java Programs. In *Proc. ISSTA'98*, pages 1-10, 1998.
5. G. Delzanno, and J.-F. Raskin. Symbolic Representation of Upward Closed Sets. In *Proc. TACAS 2000*, LNCS 1785, pages 426-440, 2000.
6. G. Delzanno, J.-F. Raskin, and L. Van Begin. Attacking Symbolic State Explosion. In *Proc. CAV'01*, LNCS 2102, pages 298-310, 2001.
7. E. A. Emerson and K. S. Namjoshi. On Model Checking for Non-deterministic Infinite-state Systems. In *Proc. of LICS '98)*, pages 70-80, 1998.
8. J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proc. LICS'99*, pages 352–359, 1999.
9. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! TCS 256 (1-2):63–92, 2001.
10. S. M. German, A. P. Sistla. Reasoning about Systems with Many Processes. *JACM* 39(3): 675–735 (1992)
11. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a Model Checker for Hybrid Systems. In *Proc. CAV'97*, LNCS 1254, pages 460-463, 1997.
12. D. Lea. Concurrent Programming in Java. Design Principle and Patterns. Second Edition. The Java Series. Addison Wesley, 2000.
13. D. Zampuniéris, and B. Le Charlier. Efficient Handling of Large Sets of Tuples with Sharing Trees. In *Proc. DCC'95*, 1995.