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
giorgio@disi.unige.it

[2] Département d'Informatique, Université Libre de Bruxelles
Blvd Du Triomphe, 1050 Bruxelles, Belgium
{jraskin,lvbegin}@ulb.ac.be

Abstract. In this paper we investigate in the possible application of parameterized verification techniques (see [1, 12, 13, 16, 15, 25]) to synchronization skeletons of *multithreaded Java programs*. As first conceptual contribution, we identify a class of *infinite-state* abstract models that preserve the main features of the semantics of concurrent Java. We achieve this goal by exploting an interesting connection with the Broadcast Protocols of [15, 16], and by introducing a new communication primitive similar to the *non-blocking* rendez-vous of [14]. We formalize these connections via a special class of Petri Nets with marking-dependent cardinality arcs [5], we called Multi-transfer Nets (MTNs). As technical contribution, we extend the symbolic verification technique based on Covering Sharing Trees [12] and previously applied to Petri Nets [13] to the new class of MTNs. Specifically, we give algorithms to symbolically compute Pre^* for an MTN, and we show how to exploit structural invariants of Petri Nets with marking-dependent cardinality arcs [5] as heuristics to efficiently explore their state-space. As practical contribution, we have incorporated these technique in our CST-based library [13] and we have tested it on several parameterized safety problems for abstractions of multithreaded Java programs as well as for broadcast and client-server protocols modeled via MTNs. The resulting procedure outperforms HyTech, an existing infinite-state symbolic model checkers that can be applied to the same class of problems.

1 Introduction

Automated verification of properties of *parameterized* concurrent systems, i.e., systems consisting of an arbitrary number of processes, is an important and challenging research goal. Parameterized verification problems arise in several different contexts like multiprocessors systems [15, 16, 9], distributed databases, and client-server protocols [10]. Although in general the verification problem for parameterized concurrent systems is undecidable, decision procedures have been discovered for the verification of subclasses in which systems consist of many *identical*, *finite-state* processes. For instance, German and Sisla [20] proposed a verification method for infinite families of asynchronous CCS processes based on an encoding into Petri Nets. Safety properties of Petri Nets can be checked algorithmically, e.g., using Karp-Miller's coverability tree construction [24]. Recently, this idea has been applied to the verification of safety properties of *multithreaded C programs* [2]. Contrary to approaches based on finite-state abstractions of sofware programs [6, 7], parameterized verification techniques can handle infinite-state abstract models (e.g. unbounded Petri Nets as in [20, 2]), with a potential gain of precision in the analysis of the underlying systems.

Multithreaded programs are essential features of nowadays client- and server-side applications (e.g. in a webserver). Given the impact of Java on this field (think about *Applets* and *Servlets*), we believe that investigating the possible application of parameterized verification techniques to multithreaded Java programs is an important research goal. In this paper we will focus on what we think are problems propedeutic to further research in this direction.

We first address the problem of finding adequate *infinite-state abstract models* for synchronization skeletons of concurrent Java programs. The abstractions based on CCS and Petri Nets models adopted for

C programs in [2] are not powerful enough to obtain synchronization skeletons that naturally preserve the semantics of the concurrent primitives used in a typical multithreaded Java application. The main difficulty comes from the presence of special Java primitives that provide for global synchronization of threads, namely the pre-defined methods `wait`, `notify` and `notifyAll`, an operational aspect that cannot be modeled via rendez-vous communication. The semantics of `notifyAll`, however, is very similar to the notion of *broadcast* communication introduced by Emerson and Namjoshi in [15, 16] in order to model invalidation-based consistency protocols for multiprocessor systems (namely, *cache coherence protocols*). Furthermore, the semantics of `notify` can be given via a special type of *non-blocking* rendez-vous similar to the *asynchronous rendez-vous* introduced in [14].

Our conceptual contribution is to show that these connections can be formalized using a special sub-class of Ciardo's *Petri Nets with marking-dependend arc cardinality* [5]. We will call the resulting model Multi-transfer Nets (MTNs) and we will adopt them as *infinite-state* abstract models for multithreaded Java programs. Contrary to the Broadcast Protocols of [16], MTNs are *conservative* extensions of Petri Nets. Furthermore, all practical examples of specification via Broadcast Protocols (see [15, 16, 9]) can be naturally formulated as MTNs.

As a second step, we attack the problem of finding an adequate technology to *efficiently* model check this new class of infinite-state models. In [16], Esparza, Finkel and Mayr have shown that verification of a special class of safety properties is decidable for extensions of Petri Nets like Broadcast Protocols. This result can be naturally extended to the class of MTNs. Decidability holds for the model checking problem called *control state reachability* that consists of deciding if a state taken from a given *upward closed* set of (unsafe) states is reachable from the initial states. Several interesting verification problems for safety properties can be represented in terms of control state reachability. The algorithm that decides the problem is based on *backward reachability* [1], i.e., on the computation of the transitive closure Pre^* of the *pre-image* operator. In our previous work [12], we have proposed a *graph*-based assertional language called Covering Sharing Trees (CSTs) to compactly represent upward-closed sets (i.e. infinite sets) of markings of Petri Nets. In [13], we have defined a symbolic backward reachability algorithm based on CSTs and we have applied it to several parameterized problems for Petri Nets. The algorithm makes use of symbolic operations like a special predecessor operator Pre working on the graph structure of CSTs, and it uses heuristics based on structural invariants of Petri Nets in order to cut the state-space during the backward search.

The *technical contribution* of this paper is to show that CST-based verification techniques can be applied to the new class of MTNs. To achieve this goal, we proceed as follows. We first give new algorithms to symbolically compute the Pre operator associated to an MTN and apply them to build a symbolic backward reachability algorithm for MTNs. Furthermore, by exploiting the property that MTNs are a subclass of Petri Nets with marking dependent cardinality arcs, we can automatically extract *structural invariants* using the reduction to *Petri Nets* described by Ciardo (Theorem 1 of [5]). This way, we can extend the methodology suggested in [13] to MTNs and use the resulting invariants to efficiently cut the state-space (i.e. the CST that represent the partial results) during backward reachability. It is important to stress that the new verification algorithms can be applied to abstractions of multithreaded Java programs *as well as* to all examples of parameterized systems that can be modeled with MTNs (see discussion in Section 7).

As *practical contribution*, we have applied an extension of our CST-based model checking library with the new algorithms to a large set of parameterized safety problems taken from the literature [9, 15, 16]. Our CST-based search engine outperforms HyTech [23] a polyhedra-based model checker that provides backward reachability, and that can handle the same class of parameterized systems.

Plan of the paper. In Section 2 we introduce abstract models for concurrent Java based on communicating machines. In Section 3 we show to give a semantics of global machines in terms of a class of extended Petri Nets called Multi-transfer Nets. In Section 4 we discuss how to extende backward reachability to MTNs. In Section 5 we define the symbolic operations needed to handle MTNs with CSTs. In Section 6 we show what are the relation between MTNs and Petri Nets with marking dependent cardinality arcs and explain how to compute a static analysis for MTNs, so as to efficiently prune the set of backward

```
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 Dec extends Thread {
   private Point p;
   public Dec(Point P) { this.p = p; }

   private void decpoint() {
       P.decx();
       P.decy();
   }
   public void run() {
       while(true) decpoint();
   }
 }
```

Fig. 1. Example of Java thread classes.

reachable markings. In Section 7 we discuss some experimental results obtained with the CST-based search engine. In Section 8 we address related works and discuss future direction of research.

2 Abstract Models for Multithreaded Java Programs

A Java thread is basically an object of predefined Java classes like `Thread` and `Runnable` [7, 26]. 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 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. 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. Awakaned 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. It is important to note that both `notify` and `notifyAll` are *non-blocking*, i.e., the thread that invokes these primitives always proceeds in its execution. The previous features (together with the possibility of associating a lock to a given object using the *synchronized* instruction, and interrupting threads) are at the basis of the methodology used for programming using threads [26].

As an example, consider the thread declarations `Dec` and `Inc` of Fig. 1, and the class `Point` of Fig. 2. The class `Point` provides methods to increment and decrement the coordinates of a `Point` object. The methods `decx` and `decy` force a thread to `wait` for the counters to be *non zero*. Everytime 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 `Dec` repeatedly invokes the method `decpoint`, defined on the `Point` methods `decx` and `decy`, to decrement the coordinates of the point. Similarly, the thread `Inc` repeatedly increments the coordinates via the method `incpoint`. Suppose now that the `main` method creates n and m instances of `Inc` and `Dec`, respectively, working on a shared point. Suppose n is passed as a parameter to `main`. To ensure the consistency of data after invocation of methods like `incpoint` and `decpoint`, the modifications of the two coordinates should be performed atomically. The property should hold for *any value of* n *and* m, i.e., *any possible number of threads*. This property can be expressed as a the following *safety property*: there are no *reachable configurations* in which two different threads have modified only one of the two coordinates. In this paper we will attack this problem using parameterized verification technology as explained in Section 7.

For this purpose, let us first focus on the problem of finding *infinite-state* abstract models for our Java example.

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

Fig. 2. A Java class for point objects.

2.1 Abstract Models as Communicating Machines

Using *predicate abstraction* [21], we know how to extract the control skeleton of generic instances of the classes Inc and Dec. The idea is to associate a Boolean variable to each *guard in the program*, namely $zeroX$ will be associated to $x == 0$ and $zeroY$ to $y == 0$, and then extrapolate the effect of an instructions on them. Applying this technique and forgetting (for the moment) the synchronization primitives, we obtain two finite-automata. Following [20, 2], methods invocations can be represented using synchronization labels and rendez-vous à-la CCS. Furthermore, each state of the automata corresponds to a control point in the flattened code of the methods (i.e. the methods of the point class are unfolded into the definition of the threads. Our abstract model should also provide a natural way to restrict the applicability of a transition depending on the value of the lock associated to a 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 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).

An adequate abstract model can be obtained by merging the asynchronous CCS-like model of [20] (one monitor, and many clients with asynchronous communication), the broadcast protocols of [15] (synchronous communication) to model `notifyAll`, the global/local machines proposed in [2] (asynchronous communication with global and local variables) to model `locks`, and by enriching them with an *non-blocking* rendez-vous (similar to the *asynchronous rendez-vous* of [14]) to model `notify`. We define next a new model inheriting all previous features.

Let us first describe the syntax of the Boolean guards and actions associated to a transitions.

Definition 1 (Boolean Guards/Actions). Let $\mathbf{B} = \{b_1, \ldots, b_n\}$ be a finite set of *global boolean variables*, and let $\mathbf{B}'$ be their primed version. Then, a *Boolean guard* φ_g is either the formula *true* or the conjunction of literals $\ell_1 \wedge \ldots \wedge \ell_p$, where $p \leq n$, 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 is one of $\{true, false, b_i\}$ for $i : 1, \ldots, n$.

Boolean guards and actions are used to express pre-and post-conditions (using primed variables) on the global variables $\mathbf{B}$. Now, we define the notion of local machine.

Definition 2 (Local machine). 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. Specifically, let $\varphi \equiv \varphi_g \wedge \varphi_a$, where φ_g is a *satisfiable* Boolean guard, and φ_a a Boolean action, then actions have one of the following form:

- *Internal action.* $\ell : \varphi$.
- *Rendez-vous.* $\ell! : \varphi$ (send) and $\ell?$ (receive).
- *Asynchronous Rendez-vous.* $\ell\!\uparrow: \varphi$ (non blocking send) and $\ell\!\downarrow$ (receive).
- *Broadcast.* $\ell!! : \varphi$ (send) and $\ell??$ (receive).

We will clarify the semantics of actions in the next paragraphs.
– The behavior of the local machine is described via the transition relation $\delta : (Q \times \mathcal{A} \times Q)$.

In the following, we will use $s \xrightarrow{\alpha} s'$ to indicate that $\langle s, \alpha, s' \rangle \in \delta$. Having in mind the translation from Java programs, we will also apply the following restrictions:

– The *source* and *target states* of a given *asynchronous rendez-vous* must be all distinct each other.
– The states occurring in a *broadcast send* must be distinct from those in the corresponding *receives.*
– The *target state* of a *broadcast receive* does not belong to the set of its *source states.*
– *Broadcasts receives* associated to the same *send* can be partitioned so that each partition is defined over a distinct set of states.

These restriction derive from the following observations. We mainly 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 (see Section 7).

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

The operational semantics of a global machine is defined as follows.

Definition 4 (Global state). A global state is a tuple $G = \langle \rho, \mathbf{s} \rangle$, where $\rho = \langle v_1, \ldots, v_n \rangle$ with $v_i \in \{true, false\}$ $i : 1, \ldots, n$ is an evaluation for the variables in $\mathbf{B} = \langle b_1, \ldots, b_n \rangle$, and the tuple $\mathbf{s} = \langle \ldots, s_{i_1}, \ldots, s_{i_{k_i}}, \ldots \rangle$ of dimension $k = k_1 + \ldots + k_m$ represents the collection of process states, such that $s_{i_j} \in Q_i$ for $j : 1, \ldots, k_i$.

The execution of a machine is formalized through the relation $\Rightarrow$ defined next.

Definition 5 (Operational Semantics). Let $G = \langle \rho, \langle s_1 \ldots, s_k \rangle \rangle$, $G = \langle \rho', \langle s'_1 \ldots, s'_k \rangle \rangle$, and $\gamma = \rho \cup \rho'$. Then, $G \stackrel{\ell}{\Longrightarrow} G'$ iff one of the following conditions holds:

– if there exist i and u such that $s_i \xrightarrow{\ell:\varphi} u$, and $\gamma(\varphi) = true$, then $s'_i = u$ and $s'_j = s_j$ for all $j \neq i$.
– if there exist i, j, u and v such that $s_i \xrightarrow{\ell!:\varphi} u$, $s_j \xrightarrow{\ell?} v$, and $\gamma(\varphi) = true$, then $s'_i = u$, $s'_j = v$ and $s'_l = s_l$ for all $l \neq i, j$.
– if there exist i and u such that $s_i \xrightarrow{\ell\uparrow:\varphi} u$, and $\gamma(\varphi) = true$, then $s'_i = u$, and: if there exist j and v such that $s_j \xrightarrow{\ell\downarrow} v$, then $s'_j = v$ and $s'_l = s_l$ for any $l \neq i, j$; otherwise, $s'_l = s_l$ for any $l \neq i$.
– if there exist i and u such that $s_i \xrightarrow{\ell!!:\varphi} u$, and $\gamma(\varphi) = true$, then $s'_i = u$; furthermore, for all j such that there exist v and $s_j \xrightarrow{\ell??} v$, then $s'_j = v$; finally, $s'_l = s_l$ for all $l \neq i$ s.t. $\ell??$ is not defined in s_l.

A *run* of a global machine is a sequence of global states $G_0 G_1 \ldots$ such that $G_i \stackrel{\ell}{\Longrightarrow} G_{i+1}$ for $i \geq 0$. G_0 is the *initial* global state of the run. A global state G' is reachable from G, written $G \stackrel{*}{\Rightarrow} G'$, if and only if there exists a run from G to G'.

Example 1. Let us go back to the Java threads described in Section 2. The abstract model we extracted applying the technique of predicate abstraction [21] is described in Fig. 8.

3 Multi-Transfer Nets (MTNs)

Following [20, 9, 2], in order to study safety properties of global machines, we will apply a *counting* abstraction that maps global states into markings that keep track of the number of processes in very local state. 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 k tokens*. Formally, this model is defined as follows.

Definition 6 (Multi-Transfer Nets). A multi-transfer net is a tuple $\langle \mathcal{P}, \mathcal{B} \rangle$, where :

- $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places,
- $\mathcal{B} = \{M_1, M_2, \ldots, M_m\}$ is a finite set of *multi-transfers*.
- A multi-transfer M_i is a tuple $\langle T_i, \{B_{i1}, \ldots, B_{ir_i}\}, c_i \rangle$, where
  - $T_i = \langle \mathcal{I}_i, \mathcal{O}_i \rangle$ is a (Petri Net) *transition*: $\mathcal{I}_i, \mathcal{O}_i : \mathcal{P} \to \mathbb{N}$ are multisets of places;
  - $B_{ij} = \langle P_{ij}, p_{ij} \rangle$ is a *transfer block*: $P_{ij} \subseteq \mathcal{P}$ is the set of *source* places, and $p_{ij} \in \mathcal{P}$ is the *target*;
  - $c_i \in \mathbb{N} \cup \{+\infty\}$ is the *bound* on the number of tokens that can be transferred via M_i.
- Futhermore, in order to avoid cyclic transfers, $\mathcal{B}$ must satisfy the following conditions: $p_{ij} \notin P_{ij}$ for any i, j (cond. 1); $(P_{ij} \cup \{p_{ij}\}) \cap (P_{ik} \cup \{p_{ik}\}) = \emptyset$ for any j and k (cond. 2); and, finally, $(P_{ij} \cup \{p_{ij}\}) \cap (\mathcal{I}_i \cup \mathcal{O}_i) = \emptyset$ for any i, j (cond. 3). □

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 $\mathbf{m}(S) = \Sigma_{p \in S} \mathbf{m}(p)$. Then, we introduce the following definitions.

Firing a Multi-transfer. Let M_i be a multi-transfer with *transition* $\langle \mathcal{I}, \mathcal{O} \rangle$, blocks $\langle P_j, p_j \rangle$ for $j : 1, \ldots, q$, and bound c. We define Θ_i as the set of places occurring in the *transition* of M_i, i.e. $\Theta_i = \{p \mid \mathcal{I}(p) \geq 1\} \cup \{p \mid \mathcal{O}(p) \geq 1\}$; and Ω_i as the set of places that are not involved in any transfer block of M_i, i.e. $\Omega_i = \mathcal{P} \setminus (\Theta_i \cup P_1 \cup \ldots \cup P_q \cup \{p_1, \ldots, p_q\})$.

We say that M_i is *enabled* at $\mathbf{m}$ if $\mathcal{I} \geq \mathbf{m}$. If M_i is enabled at $\mathbf{m}$, firing it leads to the set of marking $\mathbf{m}'$ (written $\mathbf{m} \rightarrowtail_{M_i} \mathbf{m}'$) thta satisfy one of the following conditions:

- $\mathbf{m}'(p) = \mathbf{m}(p)$ for all $p \in \Lambda_i$;
- $\mathbf{m}'(p) = \mathbf{m}(p) - \mathcal{I}(p) + \mathcal{O}(p)$ for any $p \in \Theta_i$;
- if $q > 0$ and $\mathbf{m}(P_1 \cup \ldots \cup P_q) \geq c$, let $k_1, \ldots, k_q \in \mathbb{N}$ be such that $k_1 + \ldots + k_q = c$, then $\mathbf{m}'(p_i) = \mathbf{m}(p_i) + c$, and $\mathbf{m}'(P_i) = \mathbf{m}(P_i) - k_i$ with the additional constraint that $\mathbf{m}(p) \geq \mathbf{m}'(p)$ for any $p \in P_i$, for $i : 1, \ldots, q$.
- if $q > 0$ and $\mathbf{m}(P_1 \cup \ldots \cup P_q) < c$, then $\mathbf{m}'(p_i) = \mathbf{m}(p_i) + \mathbf{m}(P_i)$ and $\mathbf{m}'(P_i) = 0$ for all $i : 1, \ldots, q$.

Note that a *Petri Net transition* is obtained by considering multi-transfers without transfer blocks. A *transfer arc* (all tokens in the sources are transferred to the target) is obtained instead by a multi-transfer with bound $c = +\infty$. For a bound $c \in \mathbb{N}$, A multi-transfer may have a *non-deterministic* effect whenever the total number of tokens is greater or equal than c. The non-determinism is due to the number k_i of tokens that are transferred in each block, and from their distribution within the set P_i of sources. The operational semantics of an MTNs is defined below.

Definition 7 (Operational Semantics of an MTN). Let $\mathcal{M} = \langle \mathcal{P}, \mathcal{B} \rangle$ be an MTN with initial marking $\mathbf{m}_0$. Then, a *run* is a sequence $\mathbf{m}_0 \mathbf{m}_1 \ldots$ such that for any $i \geq 0$ there exists $M \in \mathcal{B}$ such that $\mathbf{m}_i \rightarrowtail_M \mathbf{m}_{i+1}$. The set of markings occurring in all possible run starting from $\mathbf{m}_0$ form the reachability set of $\mathcal{M}$.

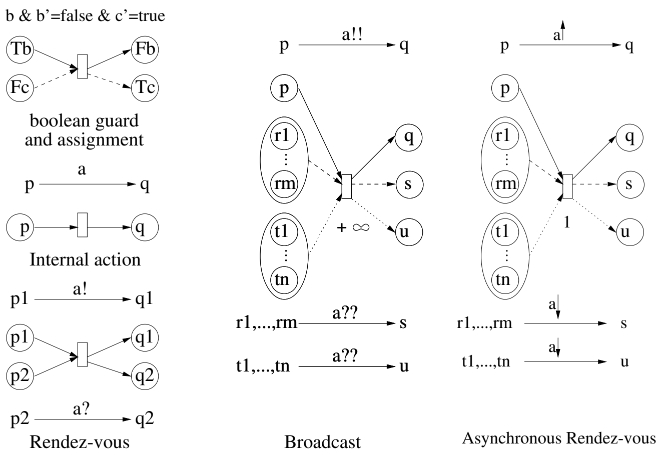

Fig. 3. From global machines rules to MTNs.

3.1 From Global Machines to MTNs

The counting abstraction that allows to reason about global machines using MTNs is defined as follows. 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 in which $\mathcal{L}_i$ has n_i local states, i.e., $Q_i = \{s_{i1}, \ldots, s_{in_i}\}$. Given a global state $G = \langle \rho, \mathbf{s} \rangle$, we define α as:

$$\alpha(G) = \langle \rho, \langle v_{11}, \ldots, v_{1n_1}, \ldots, v_{m1}, \ldots, v_{mn_m} \rangle \rangle,$$

where v_{ij}=number of *occurrences* of the state $s_{ij} \in Q_i$ in $\mathbf{s}$. Thus, α maps a global state in a tuple whose length is always $size(G) = |\mathbf{B}| + n_1 + \ldots + n_m$ for any value of $k_1, \ldots, k_m$ (i.e. independently from the number of processes). Furthermore, it maps all symmetric state into a unique representation. As an example, $\langle \langle true \rangle, \langle wait, wait, use \rangle \rangle$ and $\langle \langle true \rangle, \langle wait, use, wait \rangle \rangle$ are mapped into the abstract state $\langle \langle true \rangle, \langle 2, 1, \ldots \rangle \rangle$, where the first component is the number of *wait*, and so on. Clearly, using a Petri Net-like n which *processes* are abstracted into *black tokens* it will be possible to reason about properties of the original system that do not depend on process identifiers. Luckily, several interesting properties fall into this class.

Given a collection of local machine, we can automatically compute the corresponding MTN as sketched next (for brevity, we omit the formal translation and give only the key ideas). The translation is obtained by merging the multi-transfers associated to each boolean variables with those induced by the communication actions (the only expection is for *asynchronous rendez-vous* as we will explain later).

Boolean Guard/Actions. Let $\varphi = \varphi_g \wedge \varphi_a$ be a Boolean formula as in Def. 1. Given a Boolean variable b, we introduce the places $true_b$ and $false_b$ corresponding to its truth values. Initially, we put a token in $false_b$, none in $true_b$.

If b occurs in a literal ℓ of φ_g, and $b' = c$ occcurs in φ_a, we model the effect of φ on b as a transition going from the place naturally associated to ℓ to the place associated to c. E.g. for $\neg b \wedge b' = true$ we need a transition that tests and removes one token from $false_b$ and adds it to $true_b$. If b does not occur in φ_a, then we use a *transfer block* with bound $+\infty$ to transfer all tokens from one of the two places $true_b$ and $false_b$ to the other. E.g. for $b' = true$ we have a transfer that goes from $false_b$ to $true_b$. In the following we will denote M_φ the multi-transfer (with bound $c = +\infty$) coming out of this translation.

Internal action. We first associate a place to each local state. An internal action can be modeled then, by adding an extra input edge (from source state) and an extra output edge (to the target state) to the transition associated to the multi-transfer M_φ.

Rendez-vous. As for an internal action, but we need two edges (source and target states) for the sender and two edges for the receiver.

Asynchronous Rendez-vous. Asynchronous rendez-vous can be modeled by a multi-transfer with bound $c = 1$ (at most 1 process receives the message!). The transition of the multi-transfer goes from the source to the target state of the sender. Each trasfer block models the sources and target places of the set of receivers that move to the same state. Since M_φ is a multi-transfer with bound $+\infty$, the multi-transfer associated to an asynchronous rendez-vous (with bound $c = 1$) is *pipelined* to the multi-transfer M_φ. This solution is not particularly bad since in a Java program we usually need few `notify` actions. Alternatively, we could have generalized the definition of MTNs in order to allow different bounds on distinct *sets of transfer blocks*.

Broadcast. A broadcast can be naturally modeled via a multi-transfer with bound $c = +\infty$ (and thus merged with M_φ). Specifically, an input (from the source state of the sender) and an output (to the target state of the sender) edge are added to the transition of M_φ. Finally, a transfer block for each set of receivers with the same target state is added to the set of blocks of M_φ.

Fig. 3 summarizes the translation (according to α) from local machine to a single MTN: a transfer block is denoted here as a dashed arc going from the cluster of source places to the target place. Based on the previous ideas, we have the following proposition.

Proposition 1. Given a global machine $\mathcal{G}$, there exists an MTN $\mathcal{M}$ with multi-transfers whose bounds are either 1 or $+\infty$, and such that $\mathbf{m}_0\mathbf{m}_1\ldots$ is a run in $\mathcal{M}$ iff $G_0G_1\ldots$ is a run in $\mathcal{G}$ and $\mathbf{m}_i$ uniquely represents $\alpha(G_i)$. Thus, the reachability set of $\mathcal{M}$ is a *conservative approximation* of that of $\mathcal{G}$.

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, 18]. The goal here is to prove that none of the markings in a given infinite set U of *unsafe configurations* (representing unsafe configurations for arbitrary number of processes) can be reached from the initial (possibly parametric) marking $\mathbf{m}_0$. To achieve this goal, we first compute the closure $Pre^*(U)$ of Pre, and then check that no instance of $\mathbf{m}_0$ is in the resulting set of markings.

As shown in [1, 18], this algorithm is guaranteed to terminate on PNs and BPs whenever U is upward closed w.r.t. the componentwise ordering of tuples. Formally, let $cones(S) = \{\mathbf{m}' \mid \mathbf{m} \preccurlyeq \mathbf{m}',\ \mathbf{m} \in S\}$. Then, a set of markings S is upward closed if $cones(S) = S$. An upward closed set of markings U is always finitely generated by a set of minimal tuples, we will denote it as $gen(U)$. As an example, the marking $\langle 0,0,0,0,2,0\rangle$ generates the upward closed set consisting of all markings $\langle a,b,c,d,e,f\rangle$ such that $a,b,c,d,f \geq 0$, and $e \geq 2$. The termination of backward reachability for Petri Nets and Broadcast Protocols is based on the following facts: (1) the application of Pre (for Petri Nets and Broadcast Protocols) to an upward closed set of markings returns a set that is still upward closed; (2) the containment relation between upward closed sets of markings is a *well-quasi ordering* [1, 18]. Property (2) ensures that it is not possible to generate infinite chains of non comparable upward closed sets during the computation of Pre^*.

Since upward closed sets are infinite sets of states, one of the advantages of backward reachability is that it allows to prove safety properties for arbitrary system configurations, and to answer questions like "does the system ensures mutual exclusion for an arbitrary number of processes?". It is important to note that Karp-Miller's construction may fail from terminate for extension of Petri Nets with broadcast communication [16].

In our previous works [12], we have shown that a special data structure called Covering Sharing Trees (CSTs) can be used to compactly represent upward closed sets of markings, and to implement model checking procedures for Petri Nets. In this paper our goal is to extend CST-based model checking to the class of MTNs. To achieve this aim, let us first study the properties of the *precedessor* operator of an MTN. Its definition is given as follows.

Definition 8 (MTN Precedessor Operator). *Let $\mathcal{M} = \langle \mathcal{P}, \mathcal{B}\rangle$ be an MTN, and let $M \in \mathcal{B}$, then* $\mathsf{Pre}_M(S) = \{\mathbf{m}' \mid \mathbf{m}' \rightarrowtail_M \mathbf{m},\ \mathbf{m} \in S\}$.

By construction, the transition and transfer blocks of an MTNs involve distinct set of places each other. As a consequence, the following lemma holds.

Lemma 1 (Decomposition of Pre). *Let $B = \langle T, \{M_1, M_2, \ldots, M_n\}\rangle$ be a multi-transfer. Let $B_0 = \langle T, \emptyset\rangle$, and $B_i = \langle T_\emptyset, \{M_i\}\rangle$ where $T_\emptyset = \langle \emptyset, \emptyset\rangle$ for $i : 1, \ldots, n$. Then, the following holds* $\mathsf{Pre} = \mathsf{Pre}_{B_0} \circ \mathsf{Pre}_{B_1} \circ \mathsf{Pre}_{B_2} \circ \cdots \circ \mathsf{Pre}_{B_n}$.

This lemma will be useful later when we will describe the algorithms used to compute symbolically the Pre-operator of our MTN. Furthermore, we can easily prove that MTNs are monotonic w.r.t. the pointwise ordering on markings, i.e., if $\mathbf{m}_1 \rightarrowtail m_2$, then for any $\mathbf{m}'_1 \geq \mathbf{m}_1$ there exists $\mathbf{m}'_2 \geq \mathbf{m}_2$ such that $\mathbf{m}'_1 \rightarrowtail \mathbf{m}'_2$. According to [1, 18], from this property the following result holds.

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

In the next section we will use the previous properties to describe the algorithms used to compute symbolically the Pre-operator (of an MTN) over Covering Sharing Trees $CSTs$.

4.1 The Assertional Language of Covering Sharing Trees (CSTs)

Covering Sharing Trees are an extension of the Sharing Tree data structure introduced in [29] to efficiently store tuples of integers. A Sharing Tree $\mathbf{S}$ 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, $\mathbf{S}$ 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 $i = 1, \ldots, m$-1. Paths represent *tuples of size* k of natural numbers. We use $elem(\mathbf{S})$ to denote the *flat denotation* of a k-sharing tree $\mathbf{S}$:

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

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 $elem(\mathbf{S})$ can be exponentially larger than the size of $\mathbf{S}$. As shown in [29], given a set of tuples $\mathcal{A}$ of size k, there exists a unique sharing tree such that $elem(\mathbf{S}_\mathcal{A}) = \mathcal{A}$ (modulo isomorphisms of graphs). A Covering Sharing Tree (CST) is a sharing tree obtained by lifting the denotation as follows

$$cones(\mathbf{S}) = \{\mathbf{m} \mid \mathbf{n} \preccurlyeq \mathbf{m},\ \mathbf{n} \in elem(\mathbf{S})\}.$$ [1]

Given an upward closed set of markings U, we define the CST $\mathbf{S}_U$ as the k-sharing tree such that $elem(\mathbf{S}_U) = gen(U)$. Thus, $\mathbf{S}_U$ can be used to *compactly* represent $gen(U)$ (in the best case the size of $\mathbf{S}_U$ is *logarithmic* in the size of $gen(U)$) and to *finitely* represent U. A CST can also be viewed as a symbolic representation of the formula: $\vee_{\mathbf{m} \in gen(U)}\ (x_1 \geq m_1 \wedge \ldots \wedge x_n \geq m_n)$. An examples of CST is given in the point (a) of Fig. 5 (see Example 2).

[1] For a single marking $\mathbf{m}$, $cone(\mathbf{m})$ is defined in a similar way.

function Step2 ($\mathbf{S}$: *CST after step* (1), P : *source*, p_k : *target*) **return** $\mathbf{R}$
  $\mathbf{R} \longleftarrow$ EmptyCST
  forall *value c in the layer associated to place p_k* **do**
    $\mathbf{S_c} \longleftarrow \mathbf{S}$ where the nodes n of the layer associated to p_k such that $val(n) \neq c$ have been removed
    forall *layers of $\mathbf{S_c}$ corresponding to places $P \cup \{p_k\}$* **do**
      replace all node n by the set of nodes $\{n_0, n_1, \ldots, n_c\}$ having the same
      successors and predecessors than n and such that $val(n_k) = i$
    Normalize $\mathbf{S_c}$ according to rules (3) and (4)
    Compute $\mathbf{Q_c}$ such that $m \in cones(\mathbf{Q_c})$ iff $\sum_{p \in P \cup \{p_k\}} m(p) \geq c$
    $\mathbf{T_c} \longleftarrow \mathbf{S_c} \cap_{CST} \mathbf{Q_c}$
    $\mathbf{R} \longleftarrow \mathbf{R} \cup_{CST} \mathbf{Q_c}$

Fig. 4. Algorithm for Step (2).

5 CST-based Symbolic *Pre* Operator for MTNs

In [12], we presented an algorithm to computes the Pre associated to a Petri Net transition for an upward closed set of markings represented via a CST. To extend this technique to MTNs, we can exploit Lemma 1. That is, we simply have to define an algorithm for the Pre operator associated to a *transfer block*. For reason of space, in this section we will restrict ourselves to consider *multi-transfers* having bound $+\infty$. The algorithm can be extended however to any bound $c \in \mathbb{N}$, and in particular for $c = 1$ (we plan to describe the general in an extended version).

Let us consider then a multi-transfer M_B with bound $+\infty$ and with the only transfer block $B = \langle P, p_k \rangle$, $P \subseteq \mathcal{P}$, and $p_k \in \mathcal{P}$. Furthermore, let $I_P = \{ \ i \mid p_i \in P \ \}$ (the indexes of places in P).

> Given a CST $\mathbf{S}$, our aim is to build an algorithm to construct a CST $\mathbf{S}'$ such that $cones(\mathbf{S}') = \mathsf{Pre}_{M_B}(cones(\mathbf{S}))$.

We proceed in two steps.

(1) The first step consists in removing all paths of $\mathbf{S}$ that do not satisfy the following postcondition (induced by the semantics of transfer blocks with bound $+\infty$): *all source places in P must contain zero tokens after firing M_B* (in fact, they are all transferred to p_k!). Specifically, we first compute the CST $\mathbf{S_1}$ such that

$$elem(\mathbf{S_1}) \ = \ \{\langle c_1, c_2, \ldots, c_n\rangle \in elem(S) \mid c_i = 0, \ for \ any \ i \in I_P\}.$$

By construction, it follows that $\mathsf{Pre}_{M_B}(cones(\mathbf{S}_1)) = \mathsf{Pre}_{M_B}(cones(\mathbf{S}))$. This first step can be performed in *polynomial time* in the size of $\mathbf{S}$. We first 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 [29], a graph in which condition (4) of Section 4.1 might be violated. By applying the algorithm described in [29], the *pre*-Sharing Tree can be re-arranged into a Sharing Tree (and thus the desired CST) in polynomial time.

(2) As a second step, we compute the predecessors of the elements of $\mathbf{S_1}$, w.r.t. M_B (recall that M_B has only the block $B = \langle P, p_k \rangle$). Let us take $\mathbf{c} = \langle c_1, \ldots, c_n \rangle \in elem(\mathbf{S}_1)$. Then, we know that the tuple $\mathbf{c}$ represents the upward-closed set of markings $S = \{\mathbf{m} \mid \mathbf{m}(p_i) \geq c_i, \ i : 1, \ldots, n\}$. Applying Pre_{M_B} to $\mathbf{c}$, we should obtain a representation of the upward-closed set S' whose markings present one possible distribution of token *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, S' will be generated then by the set $gen(S')$ 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 $\mathbf{S}_1$ associated to the places in $P \cup \{p_k\}$, and replace them with nodes so that the sum of the values (associated to $P \cup \{p_k\}$) the along a path always gives c_k. As an example, consider a transfer from p_1 to p_2, and let $c_2 = 2$ be the constant in the constraint associated to p_2 in $\mathbf{S}_1$. 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 two

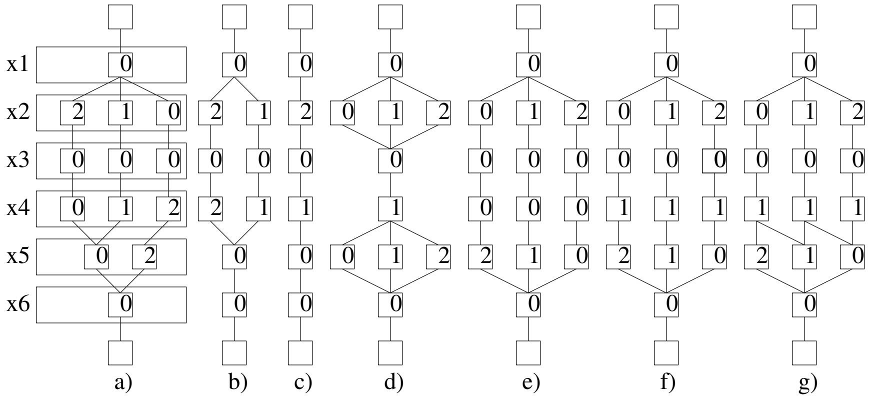

Fig. 5. Set of CST generated during the computation of the transfer $c_2' = c_2 + c_5, c_5' = 0$

(i.e. we put an edge from 0 (layer of p_1) to 2 (layer of p_2), and so on). In general, however, the algorithm is more complicate (e.g. we have to take into account places stored in non-adjacent layers, consider more than one value for c_k, etc.). To attack the general case, we split the process in two sub-steps. For each possible value of c_k in $\mathbf{S}_1$, we first compute an over-approximation (see 2 and Fig. 4), 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 algorithm is given in Fig. 4. The following example will help in undestanding this technique.

Example 2. Consider the CST $\mathbf{S}$ in Fig. 5(a) consisting of the elements $\langle 0,2,0,0,0,0 \rangle$, $\langle 0,1,0,1,0,0 \rangle$ and $\langle 0,0,0,2,2,0 \rangle$ representing the formula $\Phi = (c_2 \geq 2) \vee (c_2 \geq 1 \wedge c_4 \geq 1) \vee (c_4 \geq 2 \wedge c_5 \geq 2)$. 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-f). 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$\mathbf{Q_2}$ representing all the tuples satisfying $c_2 + c_5 \geq 2$ is shown in (e). The CST resulting from the intersection of (d) and $\mathbf{Q_2}$ 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).

6 Structural Heuristics for MTNs

When compared to forward state-space exploration, the great disadvantage of backward search is that it does take into account information about initial states. An an example, a given initial marking can forces some places (e.g. the places $true_b$ and $false_b$ associated to a Boolean variable) to be always bounded (e.g. by one). Suppose, however, we are given an over-approximation $\mathcal{R}$ of the set of reachable markings. Furthermore, let U be the seed of backward reachability. Then, we can prune the backward search space as follows:

- if $U \cap \mathcal{R} = \emptyset$, then we stop (no marking is reachable from the initial state);
- otherwise, after each application of Pre we remove all newly generated markings that lay outside $\mathcal{R}$.

In [13], we have specialized this idea to Petri Nets and CSTs by using the *structural theory* of Petri Nets, and, specifically, the notion of *place invariant*. Place invariants are solutions of the system of equation $\mathbf{y}^T \cdot C = 0$ where C is the flow matrix associated to a Petri Net, and $\mathbf{y}^T$ is the transpose of a vector of n variables (n=number of places). A place invariant $\mathbf{p}$ returns the following over-approximation of the reachability set: $\mathcal{R}_\mathbf{p} = \{\mathbf{m} \mid \mathbf{p}^T \cdot \mathbf{m} = \mathbf{p}^T \cdot \mathbf{m}_0\}$, where $\mathbf{m}_0$ is the initial marking. In the extended setting of MTNs, however, the flow matrix has a more complicated form than for Petri Nets. In fact, the cardinality of MTNs transitions is a *linear function* of the *current markings*.

This class of extended Petri Nets has been studied by Ciardo in [5] (the class $\mathcal{N}_s$ in [5]) in the context of the more general notion of *Petri Nets with marking dependent cardinality arcs*. Intuitively, in this formalism input and output edges associated to a given transition are labeled with functions like $f(\langle m_1, \ldots, m_n \rangle) = \alpha + \beta_1 \cdot m_1 + \ldots + \beta_n \cdot m_n$, where α and β_i are *natural numbers.* (Weighted) Petri Nets are a special case of this model in which functions are constants, i.e., $\beta_i = \ldots = \beta_m = 0$. To fire a transition the current marking must satisfy the constraint $\mathbf{m} \geq f(\mathbf{m})$ for every function associated to an input edge. The effect of firing a transition is obtained by the difference of the cardinality of input and output edges. The class of MTNs resulting can be viewed as a special case of Petri Nets with marking dependent cardinality arcs in virtue of the following observations.[2] As an example, for the MTNs coming out from the translation of a global machine the following observations hold.

- It is easy to see that a multi-transfer with bound 1 (e.g. an asynchronous rendez-vous) can be split into a *set of rendez-vous* (one for each possible source place), easily modeled via *constant* cardinality arcs, and an *internal move* guarded by an *inhibitor* arc (the case in which there are no tokens in the source places). However, as shown in [5], inhibitor arcs can also be modeled with linear cardinality arcs.
- As shown, e.g., in [11], broadcast operations can be modeled via *linear transformations* on counters that represent the current number of tokens in each place, and thus they can be naturally modeled using transitions with *linear functions* as cardinality arcs.

In [5], Theorem 1, Ciardo has shown that the computation of *place invariants* for a Petri Net with marking dependent cardinality arcs can be reduced to those of a Petri Net (with the same invariants). By the previous observation, we obtain the following proposition.

Proposition 3. *Given an MTN N, there exists a Petri Net N' having the same* place invariants *of N.*

By exploiting the previous corollary, we can use existing algorithms and tools like [22] to compute over-approximations of the reachable set of an MTN. In the following section we will briefly recall how we can use *place invariants* to prune CST-backward search (more details are given in [13]).

Structural Invariants and CSTs. The set $\mathcal{R}_\mathbf{p}$ associated to a place invariant $\mathbf{p}$ need not be upward closed. This implies that we cannot use CST-intersection to symbolically prune the state-space at each application of Pre_{CST}. The algorithm presented in [13] works instead in the following way. We first note that $\mathbf{p}^T \cdot \mathbf{m} > \mathbf{p}^T \cdot \mathbf{m}_0$ implies that $\mathbf{p}^T \cdot \mathbf{m}' \neq \mathbf{p}^T \cdot \mathbf{m}_0$ for all $\mathbf{m}' \in cone(\mathbf{m})$. Thus, checking $cone(\mathbf{m}) \cap \mathcal{R}_\mathbf{p}$ can be done by checking whether $\mathbf{p}^T \cdot \mathbf{m} > \mathbf{p}^T \cdot \mathbf{m}_0$ for some place invariant $\mathbf{p}$. To remove paths of the CSTs efficiently, the algorithm directly works on the structure of the CSTs. For each edge e of a CST $\mathbf{S}$, the algorithm checks if for all $\mathbf{m} \in elem(S)$ passing through e we have $cone(\mathbf{m}) \cap \mathcal{R}_\mathbf{p} = \emptyset$. An edge satisfying the test means that all the paths passing through this edge are useless and can be removed. This algorithm approximates the intersection with $\mathcal{R}_\mathbf{p}$, however works in polynomial time and preserves the upward-closure of the resulting (see [13] for major details).

7 Experimental results

We have extended the CST library of [13] with the new algorithms presented in this paper, and we have applied it to verification problems for MTNs of different type. In the next paragraph we report on these experiments.

---

[2] For brevity, we omit the formal translation that we plan to include in the extended version.

Multithreaded Java programs verification. The table in Fig. 6 describes some of the experimental results obtained on MTNs that represent abstractions of multithreaded Java programs. As an example, we have tested the MTN corresponding to the global machine of Fig. 8 starting from 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. 8, 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* using the PEP tool [22], and we have applied them to prune the search as shown in Fig. 6 (example Inc/Dec$_{bug}$). Here m and n represent the number of `Inc` and `Dec` threads in the initial marking. We considered both *parametric initial states* ($m \geq 1, n \geq 1$) as well as fixed initial markings ($m = 1, n = 1$). In the second case place invariants are much more effective (see [13]). In all cases our tool found a *potential bug*, 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` aren't. Thus, increments(decrements) on pair of coordinates are not executed atomically. Our tool finds the error after 15 iterations. To correct the error we can declare the derived methods *synchronized*, too. This way mutual exclusion is automatically guaranteed by the semantics of Java.

In Fig.6, we also consider for the Producer-Consumer example presented in [4, 6] (a classic introductory example on concurrent Java). After having fixed the number of producers and consumers, say m and n, in the initial marking, it was possible to prove a form of *freedom from deadlocks* by checking that it is not possible to reach a state in which all producers and all consumers are in the waiting state (i.e,. the unsafe states are such that the number of waiting consumers is greater or equal than m, and the number of waiting producers is greater or equal than n). The result of this experiment is given in Fig. 6 (example P/C). (Note that mutual exclusion for this example is directly ensured by the monitor-oriented semantics of Java (see [4]).) For big values of the constants involved in a constraint the compactness of CSTsis more and more effective (e.g. compare NNodes(=8509) and NElem*P(=2.434.518) for P/C $m, n = 50$ in Fig. 6, note that here the unsafe states depend on n, m).

Finally, we have analyzed a more complicated version of the example of [4] in which, by introducing new class declarations for *malicious* producers and consumers, we artifically inserted the possibility of violating mutual exclusion. In this example the presence of violations depends on the values of Boolean variables.[3] Our tool finds the bug after 14 iterations (Fig. 6, example 2P/2C$_{bug}$). We manage to verify the correct version in less than one second (Fig. 6, example 2P/2C$_{ok}$).

As shown in Fig. 6, we ran the same examples (using the same invariants) on the polyhedra-based model checker HyTech [23]. Our execution times are always better. Furthermore, in some case HyTech crashes before reaching a fixpoint or detecting the presence of the initial state because it runs out of memory.

Protocols To demonstrate the applicability of our techbnology to other classes of parameterized systems, we have considered the example called CSM in [11], and the Client-Server Protocol of [10], two examples of Broadcast Protocols. Fig. 7 shows the experimental results for these examples. In the CSM case, we considered a set of unsafe states expressed by a constraint having the form $x \geq n$, where we fixed different values for n (see Fig. 7). The CST-based search engine outperforms HyTech in all experiments. We note however that the ratio ET-CST/ET-HyTech (see Fig. 7) tends to decrease for increasing values of n. This is due to the fact that in our construction we introduce CSTs whose number of nodes depends on the constants in the corresponding constraints. Note however that for $n = 15$ our search engine is still 4 times faster than HyTech.

8 Conclusions and Related Works

In this work we focused on three main points. (1) Via a connection with previous works on parameterized verification of coherence and consistency protocols [15, 16, 9, 10], we show that there exists a class of

[3] The description of all examples are available at the URL: http://www.disi.unige.it/person/DelzannoG/CST/

MTN	Init	P	T	Prune	Bug	Steps	NNodes	NElems	ET-CST	ET-HyTech	Ratio
Inc/Dec$_{bug}$	$m,n\geq1$	32	28		$\surd$	10	1542	1823	9.77s	$\uparrow$	-
Inc/Dec$_{bug}$	$m,n\geq1$	32	28	$\surd$	$\surd$	10	538	209	0.82s s	40.24s	49
Inc/Dec$_{bug}$	$m,n=1$	32	28	$\surd$	$\surd$	10	279	68	0.22s	5.07s	23
P/C	$m,n=2$	18	14			25	833	10738	29.32s	$\uparrow$	-
P/C	$m,n=2$	18	14	$\surd$		7	109	35	0.04s	0.46s	11.5
P/C	$m,n=50$	18	14	$\surd$		151	8509	135251	22004s	$\uparrow$	-
2P/2C$_{bug}$	$k,l,m,n\geq1$	44	37		$\surd$	14	19547	14827	3002.68s	$\uparrow$	-
2P/2C$_{bug}$	$k,l,m,n\geq1$	44	37	$\surd$	$\surd$	14	1697	1330	2.82s	$\uparrow$	-
2P/2C$_{bug}$	$k,l,m,n=1$	44	37	$\surd$	$\surd$	14	1231	736	1.64s	1621.60s	989
2P/2C$_{ok}$	$k,l,m,n\geq1$	44	37			29	12479	8396	3350.05s	$\uparrow$	-
2P/2C$_{ok}$	$k,l,m,n\geq1$	44	37	$\surd$		1	46	1	0.02s	1.29s	65
2P/2C$_{ok}$	$k,l,m,n=1$	44	37	$\surd$		1	46	1	0.00s	0.06s	>6

Init=parameters in the initial markings; P=No. places; T=No. multi-transfers;
Prune=use of place invariants; Bug=potential bug detected;
Steps=No. iteration before reaching fixpoint or detecting a bug;
NNodes=Nodes in the last CST (fixpoint);
NElems=Paths in the last CST (fixpoint);
ET-CST=ex.time of the CST engine;
ET-HyTech=ex.time of HyTech ($\uparrow$ indicates that HyTech ran out of memory);
Ratio=ET-HyTech/ET-CST.

Fig. 6. Results using an AMD Athlon 900Mhz with 500 MBytes.

infinite-state abstract models that captures the essence of the concurrent model of Java and for which one can used *decision* procedures to verify automatically safety properties for arbitrary number of threads. (2) We have connected the abstract model to a special class of extended Petri Nets (called MTNs) for which we can apply *symbolic* reachability techniques as well as compute *structural invariants*. This goal is achieved by extending CST-based symbolic model checking algorithm previously defined for Petri Nets to the extended class of MTNs. (3) We have implemented all these techniques and experimented on examples of abstractions of Java programs and protocols, with promising practical results compared to other technology working on infinite-state systems.

We believe this approach is novel compared to other approaches of software verification via *finite models* (see e.g. [6, 7]) or with Petri Nets like model [2]. Our preliminary analysis of the problems, 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 (see e.g. [7]). 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. 10th IEEE Int. Symp. on Logic in Computer Science*, pages 313–321, 1996.
2. T. Ball, S. Chaki, S. K. Rajamani. Parameterized Verification of Multithreaded Software Libraries. MSR Technical Report 2000-116. To appear in Proc. *TACAS 2001*, 2001.
3. T. Bultan, R. Gerber, and W. Pugh. Symbolic Model Checking of Infinite-state Systems using Presburger Arithmetics. In *Proc. 9th Conf. on Computer Aided Verification (CAV'97)*, LNCS 1254, pages 400–411, 1997.
4. M. Campione, K. Walrath, and A. Huml. The Java Tutorial. Third Edition. A Short Course on the Basics. The Java Series. Addison Wesley, 2000.
5. G. Ciardo. Petri nets with marking-dependent arc multiplicity: properties and analysis. In *Proc. ICATPN 94*, LNCS 815, pages 179-198, 1994.

Case Study	P	T	Steps	Prune	Bug	NNodes	NElems	ET-CST	ET-HyTech	Ratio
$\mathrm{CSM}_{n=2}$	13	8	9			118	83	0.09s	1.05s	11.7
$\mathrm{CSM}_{n=2}$	13	8	7	$\surd$		73	33	0.02s	0.86s	43
$\mathrm{CSM}_{n=5}$	13	8	18			471	2058	1.36s	13.28s	9.8
$\mathrm{CSM}_{n=5}$	13	8	7	$\surd$		97	87	0.04s	0.86s	21.5
$\mathrm{CSM}_{n=10}$	13	8	33			1966	45045	41.04s	271.41s	6.6
$\mathrm{CSM}_{n=10}$	13	8	7	$\surd$		137	177	0.07s	0.86s	12.3
$\mathrm{CSM}_{n=15}$	13	8	48			5136	337212	680.76s	2786.17s	4.1
$\mathrm{CSM}_{n=15}$	13	8	7	$\surd$		177	267	0.16s	0.86s	5.38
$\mathrm{C/S}_1$	12	8	11			216	121	0.13s	5.81s	44.7
$\mathrm{C/S}_1$	12	8	5	$\surd$		62	9	0.02s	0.17s	8.5
$\mathrm{C/S}_2$	12	8	13			138	66	0.08s	3.02s	38.8
$\mathrm{C/S}_2$	12	8	2	$\surd$		22	2	0.00s	0.08s	>7
MOESI	9	11	1			11	1	0.00s	0.02s	> 2
MOESI	9	11	1	$\surd$		11	1	0.00s	0.02s	> 2
berkeley	6	8	7			31	26	0.03s	0.30	10
berkeley	6	8	1	$\surd$		8	1	0.00s	0.02s	> 2

Fig. 7. Results using an AMD Athlon 900Mhz with 500 MBytes.

6. J. C. Corbett. Constructing Compact Models of Concurrent Java Programs. In *Proc. ISSTA 1998*, pag. 1-10, 1998.
7. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proc. ICSE2000*, pag. 439-448, 2000.
8. G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. In *Performance Evaluation*, 24(1-2), pages 47-68, 1995.
9. G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *Proc. 12th Conf. on Computer Aided Verification (CAV 2000)*, LNCS 1855, pages 53–68, 1996.
10. G. Delzanno and T. Bultan. Constraint-based Verification of Client-server Protocols. Technical Report of University of Genova, 2001.
11. G. Delzanno, J. Esparza, and A. Podelski. Constraint-based Analysis of Broadcast Protocols. In *Proc. CSL '99*, LNCS 1683, pp. 50-66, 1999.
12. G. Delzanno, and J.-F. Raskin. Symbolic Representation of Upward Closed Sets. In *Proc. TACAS 2000*, LNCS 1785, pages 426-440, 2000. Also appeared as Technical Report MPI-I-1999-2-007, Max-Planck-Institut für Informatik, Saarbrücken,November 1999.
13. G. Delzanno, J.-F. Raskin, and L. Van Begin. Attacking Symbolic State Explosion. To appear in *Proc. CAV 2001*, Paris, July 2001. Also appeared as Technical Report 441, Universitè Libre de Bruxelles, January 2001.
14. A. Eliëns, and E. P. de Vink. Asynchronous Rendez-vous in Distributed Logic Programming. In . *Proc. of Foundations and Applications, REX Workshop*, LNCS, Vol. 666, pages 174-203, 1993.
15. E. A. Emerson and K. S. Namjoshi. On Model Checking for Non-deterministic Infinite-state Systems. In *Proc. of the 13th Annual Symp. on Logic in Computer Science (LICS '98)*, pages 70–80, 1998.
16. J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proc. 14th Annual Symp. on Logic in Computer Science (LICS'99)*, pages 352–359, 1999.
17. A. Finkel. The minimal coverability graph for Petri nets. In In *Advances in Petri Nets '93*, LNCS 674, pages 210-243. Springer, 1993.
18. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! TCS 256 (1-2):63–92, 2001.
19. S. M. German. Personal communication.
20. S. M. German, A. P. Sistla. Reasoning about Systems with Many Processes. *JACM* 39(3): 675–735 (1992)
21. S. Graf, H. Saïdi Construction of Abstract State Graphs with PVS. In *Proc. CAV '97*, pag. 72-83, 1997.
22. B. Grahlmann. The PEP Tool. In *Proc. CAV'97*, LNCS 1254, pages 440-443. Springer, 1997.
23. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a Model Checker for Hybrid Systems. In *Proc. 9th Conf. on Computer Aided Verification (CAV'97)*, LNCS 1254, pages 460–463, 1997.
24. R. M. Karp and R. E. Miller. Parallel Program Schemata. Journal of Computer and System Sciences, 3, pages 147-195, 1969.
25. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proc. CAV '97*, pages 424–435, 1997.

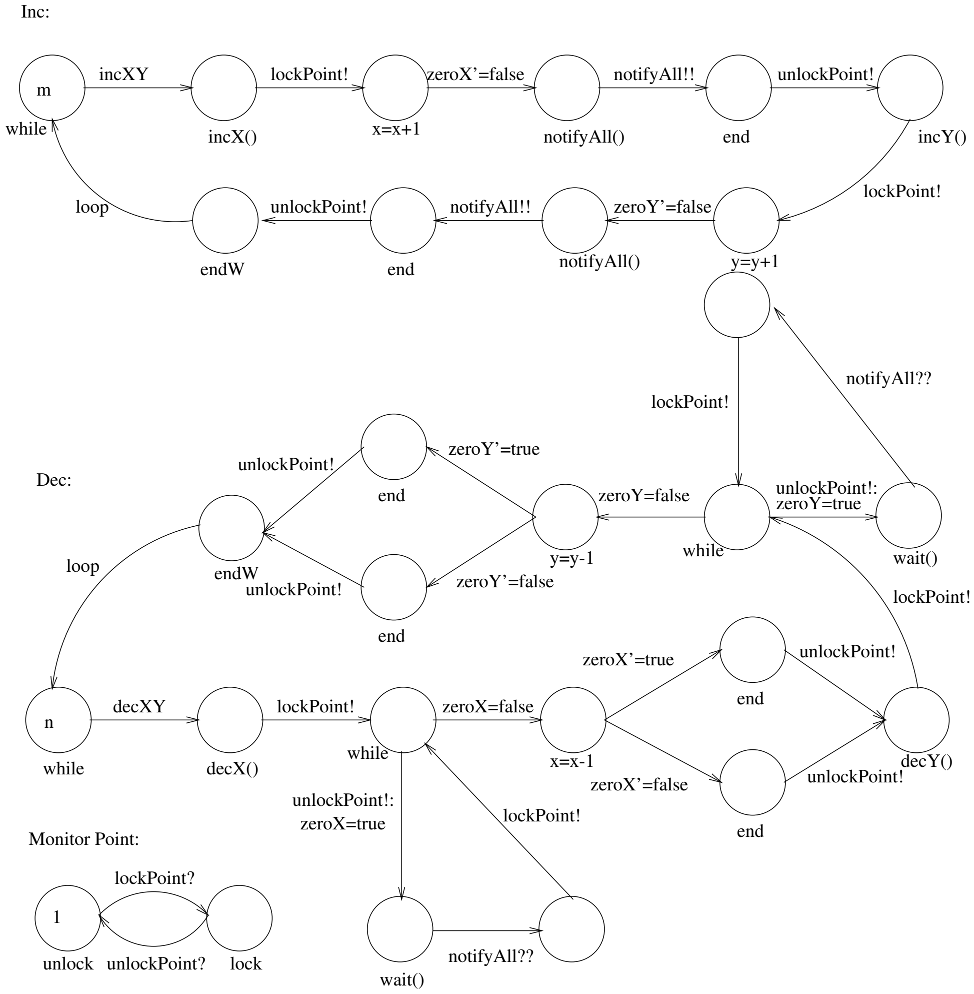

Fig. 8. The Global Machine for the `Inc` and `Dec` threads Fig. 1.

26. D. Lea. Concurrent Programming in Java. Design Principle and Patterns. Second Edition. The Java Series. Addison Wesley, 2000.
27. A. Pnueli, S. Ruah, L. D. Zuck Automatic Deductive Verification with Invisible Invariants. In *Proc. TACAS '01*, LNCS 2031, pp. 82-97, 2001.
28. M. Silva, E. Teruel, and J. M. Colom. Linear Algebraic and Linear Programming Techniques for Analysis of Place/Transition Net Systems. In W. Reisig and G. Rozenberg, editors. Lectures on Petri Nets I: Basic Models. *Advances in Petri Nets*, LNCS 1491, pages 308-309. Springer, 1998.
29. D. Zampuniéris, and B. Le Charlier. Efficient Handling of Large Sets of Tuples with Sharing Trees. In *Proceedings of the Data Compressions Conference (DCC'95)*, 1995.