TLP **7** (1&2): 67–91, 2007. © 2007 Cambridge University Press
doi:10.1017/S1471068406002821 Printed in the United Kingdom

Constraint-based automatic verification of abstract models of multithreaded programs

GIORGIO DELZANNO

Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, via Dodecaneso 35, 16146 Genova, Italy
(*e-mail:* giorgio@disi.unige.it)

submitted 17 December 2003; revised 13 April 2005; accepted 5 January 2006

Abstract

We present a technique for the automated verification of abstract models of multithreaded programs providing fresh name generation, name mobility, and unbounded control. As high level specification language we adopt here an extension of communication finite-state machines with local variables ranging over an infinite name domain, called TDL programs. Communication machines have been proved very effective for representing communication protocols as well as for representing abstractions of multithreaded software. The verification method that we propose is based on the encoding of TDL programs into a low level language based on multiset rewriting and constraints that can be viewed as an extension of Petri Nets. By means of this encoding, the symbolic verification procedure developed for the low level language in our previous work can now be applied to TDL programs. Furthermore, the encoding allows us to isolate a decidable class of verification problems for TDL programs that still provide fresh name generation, name mobility, and unbounded control. Our syntactic restrictions are in fact defined on the internal structure of threads: In order to obtain a complete and terminating method, threads are only allowed to have at most one local variable (ranging over an infinite domain of names).

KEYWORDS: constraints, multithreaded programs, verification

1 Introduction

Gordon (Gordon 2001) defines a *nominal calculus* to be a computational formalism that includes a set of *pure names* and allows the dynamic generation of *fresh*, *unguessable* names. A name is *pure* whenever it is only useful for comparing for identity with other names. The use of pure names is ubiquitous in programming languages. Some important examples are memory pointers in imperative languages, identifiers in concurrent programming languages, and nonces in security protocols. In addition to pure names, a *nominal process calculus* should provide mechanisms for *concurrency* and *inter-process communication.* A computational model that provides all these features is an adequate abstract formalism for the analysis of *multithreaded* and *distributed* software.

The problem Automated verification of specifications in a nominal process calculus becomes particularly challenging in presence of the following three features: the possibility of generating fresh names (*name generation*); the possibility of transmitting names (*name mobility*); the possibility of dynamically adding new threads of control (*unbounded control*). In fact, a calculus that provides all the previous features can be used to specify systems with a state-space infinite in *several dimensions*. This feature makes difficult (if not impossible) the application of finite-state verification techniques or techniques based on abstractions of process specifications into Petri Nets or CCS-like models. In recent years there have been several attempts of extending automated verification methods from finite-state to infinite-state systems (Abdulla and Nylén 2000; Kesten et al. 2001). In this paper we are interested in investigating the possible application of the methods we proposed in (Delzanno 2001) to verification problems of interest for nominal process calculi.

Constraint-based Symbolic Model Checking In (Delzanno 2001) we introduced a specification language, called MSR($\mathscr{C}$), for the analysis of communication protocols whose specifications are parametric in several dimensions (e.g. number of servers, clients, and tickets as in the model of the ticket mutual exclusion algorithm shown in (Bozzano and Delzanno 2002)). MSR($\mathscr{C}$) combines multiset rewriting over first order atomic formulas (Cervesato et al. 1999) with constraints programming. More specifically, multiset rewriting is used to specify the control part of a concurrent system, whereas constraints are used to symbolically specify the relations over local data. The verification method proposed (Delzanno 2005) allows us to symbolically reason on the behavior of MSR($\mathscr{C}$) specifications. To this aim, following (Abdulla et al. 1996; Abdulla and Nylén 2000) we introduced a symbolic representation of infinite collections of global configurations based on the combination of multisets of atomic formulas and constraints, called constrained configurations.[1] The verification procedure performs a symbolic backward reachability analysis by means of a symbolic *pre-image* operator that works over constrained configurations (Delzanno 2005). The main feature of this method is the possibility of automatically handling systems with an arbitrary number of components. Furthermore, since we use a symbolic and finite representation of possibly infinite sets of configurations, the analysis is carried out without loss of precision.

A natural question for our research is whether and how these techniques can be used for verification of abstract models of multithreaded programs.

Our Contribution In this paper we propose a sound, and fully automatic verification method for abstract models of multithreaded programs that provide *name generation*, *name mobility*, and *unbounded control*. As a high level specification language we adopt here an extension with value-passing of the formalism of (Ball et al. 2001) based on families of state machines used to specify abstractions of multithreaded software libraries. The resulting language is called Thread Definition Language (TDL). This

[1] Notice that in (Abdulla et al. 1996; Abdulla and Nylén 2000) a *constraint* denotes a symbolic state whereas we use the word *constraint* to denote a symbolic representation of the relation of data variables (e.g. a linear arithmetic formula) used as part of the symbolic representation of sets of states (a constrained configuration).

formalism allows us to keep separate the finite control component of a thread definition from the management of local variables (that in our setting range over a infinite set of names), and to treat in isolation the operations to generate fresh names, to transmit names, and to create new threads. In the present paper we will show that the extension of the model of (Ball et al. 2001) with value-passing makes the model Turing equivalent.

The verification methodology is based on the encoding of TDL programs into a specification in the instance MSR_{NC} of the language scheme $\mathrm{MSR}(\mathscr{C})$ of (Delzanno 2001). MSR_{NC} is obtained by taking as constraint system a subclass of linear arithmetics with only $=$ and $>$ relations between variables, called name constraints (NC). The low level specification language MSR_{NC} is not just instrumental for the encoding of TDL programs. Indeed, it has been applied to model consistency and mutual exclusion protocols in (Bozzano and Delzanno 2002; Delzanno 2005). Via this encoding, the verification method based on symbolic backward reachability obtained by instantiating the general method for $\mathrm{MSR}(\mathscr{C})$ to NC-constraints can now be applied to abstract models of multithreaded programs. Although termination is not guaranteed in general, the resulting verification method can succeed on practical examples as the Challenge-Response TDL program defined over binary predicates we will illustrated in the present paper. Furthermore, by propagating the sufficient conditions for termination defined in (Bozzano and Delzanno 2002; Delzanno 2005) back to TDL programs, we obtain an interesting class of decidable problems for abstract models of multithreaded programs still providing name generation, name mobility, and unbounded control.

Plan of the paper In Section 2 we present the Thread Definition Language (TDL) with examples of multithreaded programs. Furthermore, we discuss the expressiveness of TDL programs showing that they can simulate Two Counter Machines. In Section 3, after introducing the MSR_{NC} formalism, we show that TDL programs can be simulated by MSR_{NC} specifications. In Section 4 we show how to transfer the verification methods developed for $\mathrm{MSR}(\mathscr{C})$ to TDL programs. Furthermore, we show that safety properties can be decided for the special class of monadic TDL programs. In Section 5 we address some conclusions and discuss related work.

2 Thread Definition Language (TDL)

In this section we define TDL programs. This formalism is a natural extension with value-passing of the communicating machines used by (Ball et al. 2001) to specify abstractions of multithreaded software libraries.

Terminology Let $\mathscr{N}$ be a denumerable set of *names* equipped with the relations $=$ and $\neq$ and a special element $\bot$ such that $n \neq \bot$ for any $n \in \mathscr{N}$. Furthermore, let $\mathscr{V}$ be a denumerable set of variables, $\mathscr{C} = \{c_1, \ldots, c_m\}$ a finite set of constants, and $\mathscr{L}$ a finite set of *internal action* labels. For a fixed $V \subseteq \mathscr{V}$, the set of *expressions* is defined as $\mathscr{E} = V \cup \mathscr{C} \cup \{\bot\}$ (when necessary we will use $\mathscr{E}(V)$ to explicit the set of variables V upon which expressions are defined). The set of *channel expressions* is defined as $\mathscr{E}_{\mathrm{ch}} = V \cup \mathscr{C}$. Channel expressions will be used as synchronization labels so as to establish communication links only at execution time.

A *guard over* V is a conjunction $\gamma_1, \ldots, \gamma_s$, where γ_i is either *true*, $x = e$ or $x \neq e$ with $x \in V$ and $e \in \mathscr{E}$ for $i : 1, \ldots, s$. An *assignment* α from V to W is a conjunction like $x_i := e_i$ where $x_i \in W$, $e_i \in \mathscr{E}(V)$ for $i : 1, \ldots k$ and $x_r \neq x_s$ for $r \neq s$. A *message template m over V* is a tuple $m = \langle x_1, \ldots, x_u \rangle$ of variables in V.

Definition 1
A *TDL program* is a set $\mathscr{T} = \{P_1, \ldots, P_t\}$ of *thread definitions* (with distinct names for local variables and control locations). A *thread definition P* is a tuple $\langle Q, s_0, V, R \rangle$, where Q is a finite set of *control locations*, $s_0 \in Q$ is the initial location, $V \subseteq \mathscr{V}$ is a finite set of *local variables*, and R is a set of rules. Given $s, s' \in Q$, and $a \in \mathscr{L}$, a *rule* has one of the following forms[2]:

- *Internal move*: $s \xrightarrow{a} s'[\gamma, \alpha]$, where γ is a *guard over* V, and α is an *assignment* from V to V;
- *Name generation*: $s \xrightarrow{a} s'[x := new]$, where $x \in V$, and the expression *new* denotes a *fresh name*;
- *Thread creation: s* $\xrightarrow{a} s'[run\ P'\ with\ \alpha]$, where $P' = \langle Q', t, W, R' \rangle \in \mathscr{T}$, and α is an *assignment from V to W* that specifies the initialization of the local variables of the new thread;
- *Message sending: s* $\xrightarrow{e!m} s'[\gamma, \alpha]$, where e is a *channel expression*, m is a *message template over V* that specifies which names to pass, γ is a *guard over* V, and α is an *assignment* from V to V.
- *Message reception: s* $\xrightarrow{e?m} s'[\gamma, \alpha]$, where e is a channel expression, m is a message template over a *new* set of variables V' ($V' \cap V = \emptyset$) that specifies the names to receive, γ is a *guard over* $V \cup V'$ and α is an *assignment* from $V \cup V'$ to V.

Before giving an example, we formally introduce the operational semantics of TDL programs.

2.1 Operational Semantics

In the following we will use N to indicate the subset of *used names* of $\mathscr{N}$. Every constant $c \in \mathscr{C}$ is mapped to a distinct name $n_c \neq \bot \in N$, and $\bot$ is mapped to $\bot$.

Let $P = \langle Q, s, V, R \rangle$ and $V = \{x_1, \ldots, x_k\}$. A *local configuration* is a tuple $p = \langle s', n_1, \ldots, n_k \rangle$ where $s' \in Q$ and $n_i \in N$ is the current value of the variable $x_i \in V$ for $i : 1, \ldots, k$.

A *global configuration* $G = \langle N, p_1, \ldots, p_m \rangle$ is such that $N \subseteq \mathscr{N}$ and $p_1, \ldots, p_m$ are local configurations defined over N and over the thread definitions in $\mathscr{T}$. Note that there is no relation between indexes in a global configuration in G and in $\mathscr{T}$; G is a *pool* of active threads, and *several active threads* can be instances of the same *thread definition.*

[2] In this paper we keep assignments, name generation, and thread creation separate in order to simplify the presentation of the encoding into MSR.

Given a local configuration $p = \langle s', n_1, \ldots, n_k \rangle$, we define the *valuation* ρ_p as $\rho_p(x_i) = n_i$ if $x_i \in V$, $\rho_p(c) = n_c$ if $c \in \mathscr{C}$, and $\rho_p(\perp) = \perp$. Furthermore, we say that ρ_p satisfies the guard γ if $\rho_p(\gamma) \equiv true$, where ρ_p is extended to constraints in the natural way ($\rho_p(\varphi_1 \wedge \varphi_2) = \rho_p(\varphi_1) \wedge \rho_p(\varphi_2)$, etc.).

The execution of $x := e$ has the effect of updating the local variable x of a thread with the current value of e (a name taken from the set of used values N). On the contrary, the execution of $x := new$ associates a *fresh unused name* to x. The formula *run P with* α has the effect of adding a new thread (in its initial control location) to the current global configuration. The initial values of the local variables of the generated thread are determined by the execution of α, an assignment with local variables of the parent thread. The channel names used in a rendez-vous are determined by evaluating the channel expressions tagging sender and receiver rules. Value passing is achieved by extending the evaluation associated to the current configuration of the receiver so as to associate the output message of the sender to the variables in the input message template. The operational semantics is given via a binary relation $\Rightarrow$ defined as follows.

Definition 2
Let $G = \langle N, \ldots, \mathbf{p}, \ldots \rangle$, and $\mathbf{p} = \langle s, n_1, \ldots, n_k \rangle$ be a local configuration for $P = \langle Q, s, V, R \rangle$, $V = \{x_1, \ldots, x_k\}$, then:

- If there exists a rule $s \xrightarrow{a} s'[\gamma, \alpha]$ in R such that $\rho_\mathbf{p}$ satisfies γ, then $G \Rightarrow \langle N, \ldots, \mathbf{p}', \ldots \rangle$ (meaning that only $\mathbf{p}$ changes) where $\mathbf{p}' = \langle s', n'_1, \ldots, n'_k \rangle$, $n'_i = \rho_\mathbf{p}(e_i)$ if $x_i := e_i$ is in α, $n'_i = n_i$ otherwise, for $i : 1, \ldots, k$.
- If there exists a rule $s \xrightarrow{a} s'[x_i := new]$ in R, then $G \Rightarrow \langle N', \ldots, \mathbf{p}', \ldots \rangle$ where $\mathbf{p}' = \langle s', n'_1, \ldots, n'_k \rangle$, n_i is an unused name, i.e., $n'_i \in \mathscr{N} \setminus N$, $n'_j = n_j$ for every $j \neq i$, and $N' = N \cup \{n'_i\}$;
- If there exists a rule $s \xrightarrow{a} s'[run\ P'\ with\ \alpha]$ in R with $P' = \langle Q', t_0, W, R' \rangle$, $W = \{y_1, \ldots, y_u\}$, and α is defined as $y_1 := e_1, \ldots, y_u := e_u$ then $G \Rightarrow \langle N, \ldots, \mathbf{p}', \ldots, \mathbf{q} \rangle$ (we add a new thread whose initial local configuration is $\mathbf{q}$) where $\mathbf{p}' = \langle s', n_1, \ldots, n_k \rangle$, and $\mathbf{q} = \langle t_0, \rho_\mathbf{p}(e_1), \ldots, \rho_\mathbf{p}(e_u) \rangle$.
- Let $\mathbf{q} = \langle t, m_1, \ldots, m_r \rangle$ (distinct from $\mathbf{p}$) be a local configuration in G associated with $P' = \langle Q', t_0, W, R' \rangle$.
  Let $s \xrightarrow{e!m} s'[\gamma, \alpha]$ in R and $t \xrightarrow{e'?m'} t'[\gamma', \alpha']$ in R' be two rules such that $m = \langle x_1, \ldots, x_u \rangle$, $m' = \langle y_1, \ldots, y_v \rangle$ and $u = v$ (message templates match). We define σ as the *value passing* evaluation $\sigma(y_i) = \rho_\mathbf{p}(x_i)$ for $i : 1, \ldots, u$, and $\sigma(z) = \rho_\mathbf{q}(z)$ for $z \in W'$.
  Now if $\rho_\mathbf{p}(e) = \rho_\mathbf{p}(e')$ (channel names match), $\rho_\mathbf{p}$ satisfies γ, and σ satisfies γ', then $\langle N, \ldots, \mathbf{p}, \ldots, \mathbf{q}, \ldots \rangle \Rightarrow \langle N, \ldots, \mathbf{p}', \ldots, \mathbf{q}', \ldots \rangle$ where $\mathbf{p}' = \langle s', n'_1, \ldots, n'_k \rangle$, $n'_i = \rho_\mathbf{p}(v)$ if $x_i := v$ is in α, $n'_i = n_i$ otherwise and for $i : 1, \ldots, k$; $\mathbf{q}' = \langle t', m'_1, \ldots, m'_r \rangle$, $m'_i = \sigma(v)$ if $u_i := v$ is in α', $m'_i = m_i$ otherwise and for $i : 1, \ldots, r$.

Definition 3
An *initial global configuration* G_0 has an *arbitrary (but finite) number* of threads with local variables all set to $\perp$. A *run* is a sequence $G_0 G_1 \ldots$ such that $G_i \Rightarrow G_{i+1}$ for $i \geqslant 0$. A global configuration G is *reachable* from G_0 if there exists a run from G_0 to G.

$$
\begin{array}{l}
\mathit{Thread\ Init}(\mathit{local}\ id_A, n_A, m_A);\\
\quad init_A \xrightarrow{fresh} gen_A \qquad [n_A := new]\\
\quad gen_A \xrightarrow{c!\langle n_A\rangle} wait_A \qquad [true]\\
\quad wait_A \xrightarrow{n_A?\langle y\rangle} stop_A \qquad [m_A := y]\\
\\
\mathit{Thread\ Resp}(\mathit{local}\ id, n_B, m_B);\\
\quad init_B \xrightarrow{c?\langle x\rangle} gen_B \qquad [n_B := x]\\
\quad gen_B \xrightarrow{fresh} ready_B \qquad [m_B := new]\\
\quad ready_B \xrightarrow{n_B!\langle m_B\rangle} stop_B \qquad [true]\\
\\
\mathit{Thread\ Main}(\mathit{local}\ x);\\
\quad init_M \xrightarrow{id} create \qquad [x := new]\\
\quad create \xrightarrow{new_A} init_M \qquad [run\ Init\ with\ id_A := x, n_A := \perp, m_A := \perp, x := \perp]\\
\quad create \xrightarrow{new_B} init_M \qquad [run\ Resp\ with\ id_B := x, n_B := \perp, m_B := \perp, x := \perp_B]
\end{array}
$$

Fig. 1. Example of thread definitions.

Example 1

Let us consider a *challenge and response* protocol in which the goal of two agents Alice and Bob is to exchange a pair of new names $\langle n_A, n_B\rangle$, the first one created by Alice and the second one created by Bob, so as to build a composed secret key. We can specify the protocol by using new names to dynamically establish *private channel names* between instances of the initiator and of the responder. The TDL program in Figure 1 follows this idea. The thread *Init* specifies the behavior of the initiator. It first creates a new name using the internal action *fresh*, and stores it in the local variable n_A. Then, it sends n_A on channel c (a constant), waits for a name y on a channel with the same name as the value of the local variable n_A (the channel is specified by variable n_A) and then stores y in the local variable m_A. The thread *Resp* specifies the behavior of the responder. Upon reception of a name x on channel c, it stores in the local variable n_B, then creates a new name stored in local variable m_B and finally sends the value in m_B on a channel with the same name as the value of n_B. The thread *Main* non-deterministically creates new thread instances of type *Init* and *Resp*. The local variable x is used to store new names to be used for the creation of a new thread instance. Initially, all local variables of threads *Init*/*Resp* are set to $\perp$. In order to allow process instances to participate to several sessions (potentially with different principals), we could also add the following rule

$$stop_A \xrightarrow{restart} init_A[n_A := \perp, m_A := \perp]$$

In this rule we require that *roles* and *identities* do not change from session to session.[3] Starting from $G_0 = \langle N_0, \langle init, \perp\rangle\rangle$, and running the *Main* thread we can generate any number of copies of the threads *Init* and *Resp* each one with a unique

[3] By means of thread and fresh name creation it is also possible to specify a restart rule in which a given process takes a different role or identity.

identifier. Thus, we obtain global configurations like

$$\begin{array}{ll}\langle N, & \langle init_M, \perp\rangle, \\ & \langle init_A, i_1, \perp, \perp\rangle, \ldots, \langle init_A, i_K, \perp, \perp\rangle, \\ & \langle init_B, i_{K+1}, \perp, \perp\rangle, \ldots, \langle init_B, i_{K+L}, \perp, \perp\rangle \ \rangle\end{array}$$

where $N = \{\perp, i_1, \ldots, i_K, i_{K+1}, \ldots, i_{K+L}\}$ for $K, L \geqslant 0$. The threads of type *Init* and *Resp* can start parallel sessions. For $K = 1$ and $L = 1$ one possible session is as follows.
Starting from

$$\langle\{\perp, i_1, i_2\}, \langle init_M, \perp\rangle, \langle init_A, i_1, \perp, \perp\rangle, \langle init_B, i_2, \perp, \perp\rangle\rangle$$

if we apply the first rule of thread *Init* to $\langle init_A, i_1, \perp, \perp\rangle$ we obtain

$$\langle\{\perp, i_1, i_2, a^1\}, \langle init_M, \perp\rangle, \langle gen_A, i_1, a^1, \perp\rangle, \langle init_B, i_2, \perp, \perp\rangle\rangle$$

where a^1 is the generated name (a^1 is distinct from $\perp$, i_1, and i_2). Now if we apply the second rule of thread *Init* and the first rule of thread *Resp* (synchronization on channel c) we obtain

$$\langle\{\perp, i_1, i_2, a^1\}, \langle init_M, \perp\rangle, \langle wait_A, i_1, a^1, \perp\rangle, \langle gen_B, i_2, a^1, \perp\rangle\rangle$$

If we apply the second rule of thread *Resp* we obtain

$$\langle\{\perp, i_1, i_2, a^1, a^2\}, \langle init_M, \perp\rangle, \langle wait_A, i_1, a^1, \perp\rangle, \langle ready_B, i_2, a^1, a^2\rangle\rangle$$

Finally, if we apply the last rule of thread *Init* and *Resp* (synchronization on channel a^1) we obtain

$$\langle\{\perp, i_1, i_2, a^1, a^2\}, \langle init_M, \perp\rangle, \langle stop_A, i_1, a^1, a^2\rangle, \langle stop_B, i_2, a^1, a^2\rangle\rangle$$

Thus, at the end of the session the thread instances i_1 and i_2 have both a local copy of the fresh names a^1 and a^2. Note that a copy of the main thread $\langle init_M, \perp\rangle$ is always active in any reachable configuration, and, at any time, it may introduce new threads (either of type *Init* or *Resp*) with fresh identifiers. Generation of fresh names is also used by the threads of type *Init* and *Resp* to create nonces. Furthermore, threads can restart their life cycle (without changing identifiers). Thus, in this example the set of possible reachable configurations is infinite and contains configurations with arbitrarily many threads and fresh names. Since names are stored in the local variables of active threads, the local data also range over an infinite domain. □

2.2 Expressive Power of TDL

To study the expressive power of the TDL language, we will compare it with the Turing equivalent formalism called Two Counter Machines. A Two Counters Machine configurations is a tuple $\langle \ell, c_1 = n_1, c_2 = n_2\rangle$ where ℓ is control location taken from a finite set Q, and n_1 and n_2 are natural numbers that represent the values of the counters c_1 and c_2. Each counter can be incremented or decremented (if greater than zero) by one. Transitions combine operations on individual counters with changes of control locations. Specifically, the instructions for counter c_i

are as follows

Inc: ℓ_1: $c_i := c_i + 1$; goto ℓ_2;
Dec: ℓ_1: if $c_i > 0$ then $c_i := c_i - 1$; goto ℓ_2; else goto ℓ_3;

A Two Counter Machine consists then of a list of instructions and of the initial state $\langle \ell_0, c_1 = 0, c_2 = 0\rangle$. The operational semantics is defined according to the intuitive semantics of the instructions. Problems like control state reachability are undecidable for this computational model.
The following property then holds.

Theorem 1
TDL programs can simulate Two Counter Machines.

Proof
In order to define a TDL program that simulates a Two Counter Machine we proceed as follows. Every counter is represented via a *doubly linked list* implemented via a collection of threads of type *Cell* and with a unique thread of type *Last* pointing to the head of the list. The i-th counter having value zero is represented as the *empty list* $Cell(i,v,v), Last(i,v,w)$ for some name v and w (we will explain later the use of w). The i-th counter having value k is represented as

$$Cell(i,v_0,v_0), Cell(i,v_0,v_1), \ldots, C(i,v_{k-1},v_k), Last(i,v_k,w)$$

for distinct names $v_0, v_1, \ldots, v_k$. The instructions on a counter are simulated by sending messages to the corresponding *Last* thread. The messages are sent on channel *Zero* (zero test), *Dec* (decrement), and *Inc* (increment). In reply to each of these messages, the thread *Last* sends an acknowledgment, namely *Yes*/*No* for the zero test, *DAck* for the decrement, *IAck* for the increment operation. *Last* interacts with the *Cell* threads via the messages *tstC*, *decC*, *incC* acknowledged by messages *z*/*nz*, *dack*. *iack*. The interactions between a *Last* thread and a *Cell* thread is as follows.

Zero Test Upon reception of a message $\langle x\rangle$ on channel *Zero*, the *Last* thread with local variables $id, last, aux$ checks that its identifier id matches x – see transition from *Idle* to *Busy* – sends a message $\langle id, last\rangle$ on channel *tstC* directed to the cell pointed to by *last* (transition from *Busy* to *Wait*), and then waits for an answer. If the answer is sent on channel *nz*, standing for non-zero, (resp. *z* standing for zero) – see transition from *Wait* to *AckNZ* (resp. *AckZ*) – then it sends its identifier on channel *No* (resp. *Yes*) as an acknowledgment to the first message – see transition from *AckNZ* (resp. *Z*) to *Idle*. As shown in Fig. 3, the thread *Cell* with local variables *idc*, *prev*, and *next* that receives the message *tstC*, i.e., pointed to by a thread *Last* with the same identifier as *idc*, sends an acknowledgment on channel *z* (zero) if $prev = next$, and on channel *nz* (non-zero) if $prev \neq next$.

Decrement Upon reception of a message $\langle x\rangle$ on channel *Dec*, the *Last* thread with local variables $id, last, aux$ checks that its identifier id matches x (transition from *Idle* to *Dbusy*), sends a message $\langle id, last\rangle$ on channel *decC* directed to the cell pointed to by *last* (transition from *Busy* to *Wait*), and then waits for an answer. If the answer is sent on channel *dack* (transition from *DWait* to *DAck*) then it updates the local

Thread $Last(local\ id, last, aux)$;

(Zero test)

$$Idle \xrightarrow{Zero?\langle x\rangle} Busy \qquad [id = x]$$

$$Busy \xrightarrow{tstC!\langle id,last\rangle} Wait$$

$$Wait \xrightarrow{nz?\langle x\rangle} AckNZ \qquad [id = x]$$

$$Wait \xrightarrow{z?\langle x\rangle} AckZ \qquad [id = x]$$

$$AckZ \xrightarrow{Yes!\langle id\rangle} Idle$$

$$AckNZ \xrightarrow{No!\langle id\rangle} idle$$

(Decrement)

$$Idle \xrightarrow{Dec?\langle x\rangle} Dbusy \qquad [id = x]$$

$$DBusy \xrightarrow{decC!\langle id,last\rangle} DWait$$

$$DWait \xrightarrow{dack?\langle x,u\rangle} DAck \qquad [id = x, last := u]$$

$$DAck \xrightarrow{DAck!\langle id\rangle} Idle$$

(Increment)

$$Idle \xrightarrow{Inc?\langle x\rangle} INew \qquad [id = x]$$

$$INew \xrightarrow{new} IRun \qquad [aux := new]$$

$$IRun \xrightarrow{run} IAck \qquad [run\ Cell\ with\ idc := id; prev := last; next := aux]$$

$$IAck \xrightarrow{IAck!\langle id\rangle} Idle \qquad [last := aux]$$

Fig. 2. The process defining the last cell of the linked list associated to a counter

variable *last* with the pointer *u* sent by the thread *Cell*, namely the *prev* pointer of the cell pointed to by the current value of *last*, and then sends its identifier on channel *DAck* to acknowledge the first message (transition from *DAck* to *Idle*).

As shown in Fig. 3, a thread *Cell* with local variables *idc*, *prev*, and *next* that receives the message *decC* and such that $next = last$ sends as an acknowledgment on channel *dack* the value *prev*.

Increment To simulate the increment operation, *Last* does not have to interact with existing *Cell* threads. Indeed, it only has to link a new *Cell* thread to the head of the list (this is way the *Cell* thread has no operations to handle the increment operation). As shown in Fig. 2 this can be done by creating a new name stored in the

Thread $Cell(local\ idc, prev, next)$;

(Zero test)

$$idle \xrightarrow{tstC?\langle x,u\rangle} ackZ \quad [x = idc, u = next, prev = next]$$

$$idle \xrightarrow{tstC?\langle x,u\rangle} ackNZ \quad [x = idc, u = next, prev \neq next]$$

$$ackZ \xrightarrow{z!\langle idc\rangle} idle$$

$$ackNZ \xrightarrow{nz!\langle idc\rangle} idle$$

(Decrement)

$$idle \xrightarrow{dec?\langle x,u\rangle} dec \quad [x = idc, u = next, prev \neq next]$$

$$dec \xrightarrow{dack!\langle idc,prev\rangle} idle$$

Fig. 3. The process defining a cell of the linked list associated to a counter

Thread $CM(local\ id_1, id_2)$;

$\vdots$

(Instruction : ℓ_1 : $\mathbf{c_i := c_i + 1;\ goto\ \ell_2;}$**)**

$$\ell_1 \xrightarrow{Inc!\langle id_i\rangle} wait_{\ell_1}$$

$$wait_{\ell_1} \xrightarrow{IAck!\langle x\rangle} \ell_2 \quad [x = id_i]$$

$\vdots$

(Instruction : ℓ_1 : $\mathbf{c_i > 0\ then\ c_i := c_i - 1;\ goto\ \ell_2; else\ goto\ \ell_3;}$**)**

$$\ell_1 \xrightarrow{Zero!\langle id_i\rangle} wait_{\ell_1}$$

$$wait_{\ell_1} \xrightarrow{NZAck?\langle x\rangle} dec_{\ell_1} \quad [x = id_i]$$

$$dec_{\ell_1} \xrightarrow{Dec!\langle id_i\rangle} wdec_{\ell_1}$$

$$wdec_{\ell_1} \xrightarrow{DAck?\langle y\rangle} \ell_2 \quad [y = id_i]$$

$$wait_{\ell_1} \xrightarrow{ZAck?\langle x\rangle} \ell_3 \quad [x = id_i]$$

$\vdots$

Fig. 4. The thread associated to a 2CM.

local variable *aux* (transition from *INew* to *IRun*) and spawning a new *Cell* thread (transition from *IRun* to *IAck*) with *prev* pointer equal to *last*, and *next* pointer equal to *aux*. Finally, it acknowledges the increment request by sending its identifier on channel *IAck* and updates variable *last* with the current value of *aux*.

Two Counter Machine Instructions We are now ready to use the operations provided by the thread *Last* to simulate the instructions of a Two Counter Machine. As shown in Figure 4, we use a thread *CM* with two local variables id_1, id_2 to represent the list

Thread $Init(local\ nid_1, p_1, nid_2, p_2)$;

$$init \xrightarrow{freshId} init_1 \quad [nid_1 := new]$$

$$init_1 \xrightarrow{freshP} init_2 \quad [p_1 := new]$$

$$init_2 \xrightarrow{runC} init_3 \quad [run\ Cell\ with\ idc := nid_1; prev := p_1; next := p_1]$$

$$init_3 \xrightarrow{runL} init_4 \quad [run\ Last\ with\ idc := nid_1; last := p_1; aux := \perp]$$

$$init_4 \xrightarrow{freshId} init_5 \quad [nid_2 := new]$$

$$init_5 \xrightarrow{freshP} init_6 \quad [p_2 := new]$$

$$init_6 \xrightarrow{runC} init_7 \quad [run\ Cell\ with\ idc := nid_2; prev := p_2; next := p_2]$$

$$init_7 \xrightarrow{runL} init_8 \quad [run\ Last\ with\ idc := nid_2; last := p_2; aux := \perp]$$

$$init_8 \xrightarrow{runCM} init_9 \quad [run\ 2CM\ with\ id_1 := nid_1; id_2 := nid_2]$$

Fig. 5. The initialization thread.

of instructions of a 2CM with counters c_1, c_2. Control locations of the Two Counter Machines are used as local states of the thread CM. The initial local state of the CM thread is the initial control location. The increment instruction on counter c_i at control location ℓ_1 is simulated by an handshaking with the $Last$ thread with identifier id_i: we first send the message $Inc!\langle id_i \rangle$, wait for the acknowledgment on channel $IAck$ and then move to state ℓ_2. Similarly, for the decrement instruction on counter c_i at control location ℓ_1 we first send the message $Zero!\langle id_i \rangle$. If we receive an acknowledgment on channel $NZAck$ we send a Dec request, wait for completion and then move to ℓ_2. If we receive an acknowledgment on channel $ZAck$ we directly move to ℓ_3.

Initialization The last step of the encoding is the definition of the initial state of the system. For this purpose, we use the thread $Init$ of Figure 5. The first four rules of $Init$ initialize the first counter: they create two new names nid_1 (an identifier for counter c_1) and p_1, and then spawn the new threads $Cell(nid_1, p_1, p_1), Last(nid_1, p_1, \perp)$. The following four rules spawns the new threads $Cell(nid_2, p_2, p_2), Last(nid_2, p_2, \perp)$. After this stage, we create a thread of type $2CM$ to start the simulation of the instructions of the Two Counter Machines. The initial configuration of the whole system is $G_0 = \langle init, \perp, \perp \rangle$. By construction we have that an execution step from $\langle \ell_1, c_1 = n_1, c_2 = n_2 \rangle$ to $\langle \ell_2, c_1 = m_1, c_2 = m_2 \rangle$ is simulated by an execution run going from a global configuration in which the local state of thread CM is $\langle \ell_1, id_1, id_2 \rangle$ and in which we have n_i occurrences of thread $Cell$ with the same identifier id_i for $i : 1, 2$, to a global configuration in which the local state of thread CM is $\langle \ell_2, id_1, id_2 \rangle$ and in which we have m_i occurrences of thread $Cell$ with the same identifier id_i for $i : 1, 2$. Thus, every executions of a 2CM M corresponds to an execution of the corresponding TDL program that starts from the initial configuration $G_0 = \langle init, \perp, \perp \rangle$. □

As a consequence of the previous theorem, we have the following corollary.

Corollary 1
Given a TDL program, a global configurations G, and a control location ℓ, deciding if there exists a run going from G_0 to a global configuration that contains ℓ (control state reachability) is an undecidable problem.

3 From TDL to MSR$_{NC}$

As mentioned in the introduction, our verification methodology is based on a translation of TDL programs into low level specifications given in MSR$_{NC}$. Our goal is to extend the connection between CCS and Petri Nets (German and Sistla 1992) to TDL and MSR so as to be able to apply the verification methods defined in (Delzanno 2005) to multithreaded programs. In the next section we will summarize the main features of the language MSR$_{NC}$ introduced in (Delzanno 2001).

3.1 Preliminaries on MSR$_{NC}$

NC-constraints are linear arithmetic constraints in which conjuncts have one of the following form: *true*, $x = y$, $x > y$, $x = c$, or $x > c$, x and y being two variables from a denumerable set $\mathscr{V}$ that range over the rationals, and c being an integer. The *solutions* Sol of a constraint φ are defined as all evaluations (from $\mathscr{V}$ to $\mathbb{Q}$) that satisfy φ. A *constraint* φ is *satisfiable* whenever $Sol(\varphi) \neq \emptyset$. Furthermore, ψ *entails* φ whenever $Sol(\psi) \subseteq Sol(\varphi)$. NC-constraints are closed under elimination of existentially quantified variables.

Let $\mathscr{P}$ be a set of predicate symbols. An *atomic formula* $p(x_1, \ldots, x_n)$ is such that $p \in \mathscr{P}$, and $x_1, \ldots, x_n$ are *distinct* variables in $\mathscr{V}$. A *multiset* of atomic formulas is indicated as $A_1 \mid \ldots \mid A_k$, where A_i and A_j have distinct variables (we use variable renaming if necessary), and $\mid$ is the multiset constructor.

In the rest of the paper we will use $\mathscr{M}$, $\mathscr{N}$, ... to denote *multisets* of atomic formulas, ϵ to denote the *empty multiset*, $\oplus$ to denote *multiset union* and $\ominus$ to denote *multiset difference*. An MSR$_{NC}$ *configuration* is a multiset of *ground atomic formulas*, i.e., atomic formulas like $p(d_1, \ldots, d_n)$ where d_i is a rational for $i : 1, \ldots, n$.

An MSR$_{NC}$ *rule* has the form $\mathscr{M} \longrightarrow \mathscr{M}' : \varphi$, where $\mathscr{M}$ and $\mathscr{M}'$ are two (possibly empty) multisets of atomic formulas with *distinct* variables built on predicates in $\mathscr{P}$, and φ is an NC-constraint. The ground instances of an MSR$_{NC}$ rule are defined as

$$Inst(\mathscr{M} \longrightarrow \mathscr{M}' : \varphi) = \{\sigma(\mathscr{M}) \longrightarrow \sigma(\mathscr{M}') \mid \sigma \in Sol(\varphi)\}$$

where σ is extended in the natural way to multisets, i.e. $\sigma(\mathscr{M})$ and $\sigma(\mathscr{M}')$ are MSR$_{NC}$ configurations.

An MSR$_{NC}$ *specification* $\mathscr{S}$ is a tuple $\langle \mathscr{P}, \mathscr{I}, \mathscr{R} \rangle$, where $\mathscr{P}$ is a finite set of predicate symbols, $\mathscr{I}$ is finite a set of (*initial*) MSR$_{NC}$ configurations, and $\mathscr{R}$ is a finite set of MSR$_{NC}$ rules over $\mathscr{P}$.

The operational semantics describes the update from a configuration $\mathscr{M}$ to one of its possible successor configurations $\mathscr{M}'$. $\mathscr{M}'$ is obtained from $\mathscr{M}$ by rewriting (modulo associativity and commutativity) the left-hand side of an instance of a rule into the corresponding right-hand side. In order to be fireable, the left-hand side must be included in $\mathscr{M}$. Since instances and rules are selected in a non deterministic way, in general a configuration can have a (possibly infinite) set of (one-step) successors.

Formally, a rule $\mathscr{H} \longrightarrow \mathscr{B} \; : \; \varphi$ from $\mathscr{R}$ is enabled at $\mathscr{M}$ *via* the ground substitution $\sigma \in Sol(\varphi)$ if and only if $\sigma(\mathscr{H}) \preccurlyeq \mathscr{M}$. Firing rule R enabled at $\mathscr{M}$ via σ yields the new configuration

$$\mathscr{M}' \;=\; \sigma(\mathscr{B}) \oplus (\mathscr{M} \ominus \sigma(\mathscr{H}))$$

We use $\mathscr{M} \Rightarrow_{MSR} \mathscr{M}'$ to denote the firing of a rule at $\mathscr{M}$ yielding $\mathscr{M}'$.

A run is a sequence of configurations $\mathscr{M}_0\mathscr{M}_1 \ldots \mathscr{M}_k$ with $\mathscr{M}_0 \in \mathscr{I}$ such that $\mathscr{M}_i \Rightarrow_{MSR} \mathscr{M}_{i+1}$ for $i \geqslant 0$. A configuration $\mathscr{M}$ is reachable if there exists $\mathscr{M}_0 \in \mathscr{I}$ such that $\mathscr{M}_0 \overset{*}{\Rightarrow}_{MSR} \mathscr{M}$, where $\overset{*}{\Rightarrow}_{MSR}$ is the transitive closure of $\Rightarrow_{MSR}$. Finally, the successor and predecessor operators $Post$ and Pre are defined on a set of configurations S as $Post(S) = \{\mathscr{M}' | \mathscr{M} \Rightarrow_{MSR} \mathscr{M}', \mathscr{M} \in S\}$ and $Pre(S) = \{\mathscr{M} | \mathscr{M} \Rightarrow_{MSR} \mathscr{M}', \mathscr{M}' \in S\}$, respectively. Pre^* and $Post^*$ denote their transitive closure.

As shown in (Delzanno 2001; Bozzano and Delzanno 2002), Petri Nets represent a natural abstractions of MSR$_{NC}$ (and more in general of MSR rule with constraints) specifications. They can be encoded, in fact, in *propositional* MSR specifications (e.g. abstracting away arguments from atomic formulas).

3.2 Translation from TDL to MSR$_{NC}$

The first thing to do is to find an adequate representation of names. Since all we need is a way to distinguish old and new names, we just need an infinite domain in which the $=$ and $\neq$ relation are supported. Thus, we can interpret names in $\mathscr{N}$ either as integer of as rational numbers. Since operations like variable elimination are computationally less expensive than over integers, we choose to view names as non-negative rationals. Thus, a local (TDL) configuration $p = \langle s, n_1, \ldots, n_k \rangle$ is encoded as the atomic formula $p^\bullet = s(n_1, \ldots, n_k)$, where n_i is a non-negative rational. Furthermore, a global (TDL) configuration $G = \langle N, p_1, \ldots, p_m \rangle$ is encoded as an MSR$_{NC}$ configuration $G^\bullet$

$$p_1^\bullet \mid \ldots \mid p_m^\bullet \mid fresh(n)$$

where the value n in the auxiliary atomic formula $fresh(n)$ is a rational number strictly greater than all values occurring in $p_1^\bullet, \ldots, p_m^\bullet$. The predicate $fresh$ allows us to generate unused names every time needed.

The translation of constants $\mathscr{C} = \{c_1, \ldots, c_m\}$, and variables is defined as follows: $x^\bullet = x$ for $x \in \mathscr{V}$, $\bot^\bullet = 0$, $c_i^\bullet = i$ for $i : 1, \ldots, m$. We extend the translation in the natural way on a guard γ, by decomposing every formula $x \neq e$ into $x < e^\bullet$ and $x > e^\bullet$. We will call $\gamma^\bullet$ the resulting *set* of NC-constraints.[4]

Given $V = \{x_1, \ldots, x_k\}$, we define V' as the set of new variables $\{x_1', \ldots, x_k'\}$. Now, let us consider the assignment α defined as $x_1 := e_1, \ldots, x_k := e_k$ (we add

[4] As an example, if γ is the constraint $x = 1, x \neq z$ then $\gamma^\bullet$ consists of the two constraints $x = 1, x > z$ and $x = 1, z > x$.

assignments like $x_i := x_i$ if some variable does not occur as target of α). Then, $\alpha^\bullet$ is the NC-constraint $x'_1 = e^\bullet_1, \ldots, x'_k = e^\bullet_k$.

The translation of thread definitions is defined below (where we will often refer to Example 1).

Initial Global Configuration Given an initial global configuration consisting of the local configurations $\langle s_i, n_{i1}, \ldots, n_{ik_i}\rangle$ with $n_{ij} = \bot$ for $i : 1, \ldots, u$, we define the following MSR$_{NC}$ rule

$$init \rightarrow s_1(x_{11}, \ldots, x_{1k1}) \mid \ldots \mid s_u(x_{u1}, \ldots, x_{uk_u}) \mid fresh(x) \ : \\ x > C, x_{11} = 0, \ldots, x_{uk_u} = 0$$

here C is the largest rational used to interpret the constants in $\mathscr{C}$.

For each thread definition $P = \langle Q, s_0, V, R\rangle$ in $\mathscr{T}$ with $V = \{x_1, \ldots, x_k\}$ we translate the rules in R as described below.

Internal Moves For every *internal move* $s \xrightarrow{a} s'[\gamma, \alpha]$, and every $v \in \gamma^\bullet$ we define

$$s(x_1, \ldots, x_k) \rightarrow s'(x'_1, \ldots, x'_k) : v, \alpha^\bullet$$

Name Generation For every *name generation* $s \xrightarrow{a} s'[x_i := new]$, we define

$$s(x_1, \ldots, x_k) \mid fresh(x) \rightarrow s'(x'_1, \ldots, x'_k) \mid fresh(y) : y > x'_i, x'_i > x, \bigwedge_{j \neq i} x'_j = x_j$$

For instance, the name generation $init_A \xrightarrow{fresh} gen_A[n := new]$ is mapped into the MSR$_{NC}$ rule $init_A(id, x, y) | \ fresh(u) \longrightarrow gen_A(id', x', y') \mid fresh(u') : \varphi$ where φ is the constraint $u' > x', x' > u, y' = y, id' = id$. The constraint $x' > u$ represents the fact that the new name associated to the local variable n (the second argument of the atoms representing the thread) is fresh, whereas $u' > x'$ updates the current value of $fresh$ to ensure that the next generated names will be picked up from unused values.

Thread Creation Let $P = \langle Q', t_0, V', R'\rangle$ and $V' = \{y_1, \ldots, y_u\}$. Then, for every *thread creation* $s \xrightarrow{a} s'[run\ P\ with\ \alpha]$, we define

$$s(x_1, \ldots, x_k) \rightarrow s'(x'_1, \ldots, x'_k) \mid t(y'_1, \ldots, y'_u) \ : \ x'_1 = x_1, \ldots, x'_k = x_k, \alpha^\bullet.$$

E.g., consider the rule $create \xrightarrow{new_A} init_M[run\ Init\ with\ id := x, \ldots]$ of Example 1. Its encoding yields the MSR$_{NC}$ rule $create(x) \longrightarrow init_M(x') \mid init_A(id', n', m') : \psi$, where ψ represents the initialization of the local variables of the new thread $x' = x, id' = x, n' = 0, m' = 0$.

Rendez-vous The encoding of rendez-vous communication is based on the use of constraint operations like variable elimination. Let P and P' be a pair of thread definitions, with local variables $V = \{x_1, \ldots, x_k\}$ and $V' = \{y_1, \ldots, y_l\}$ with $V \cap V' = \emptyset$. We first select all rules $s \xrightarrow{e!m} s'[\gamma, \alpha]$ in R and $t \xrightarrow{e'?m'} t'[\gamma', \alpha']$ in R', such that $m = \langle w_1, \ldots, w_u\rangle$, $m' = \langle w'_1, \ldots, w'_v\rangle$ and $u = v$. Then, we define the new MSR$_{NC}$ rule

$$s(x_1, \ldots, x_k) \mid t(y_1, \ldots, y_l) \rightarrow s'(x'_1, \ldots, x'_k) \mid t'(y'_1, \ldots, y'_l) : \varphi$$

for every $v \in \gamma^\bullet$ and $v' \in \gamma'^\bullet$ such that the NC-constraint φ obtained by eliminating $w'_1, \ldots, w'_v$ from the constraint $v \wedge v' \wedge \alpha^\bullet \wedge \alpha'^\bullet \wedge w_1 = w'_1 \wedge \ldots \wedge w_v = w'_v$

$$
\begin{aligned}
& init \longrightarrow fresh(x) \mid init_M(y) \ : \ x > 0, y = 0. \\
& fresh(x) \mid init_M(y) \longrightarrow fresh(x') \mid create(y') \ : \ x' > y', y' > x. \\
& create(x) \longrightarrow init_M(x') \mid init_A(id', n', m') \ : \ x' = x, id' = x, n' = 0, m' = 0. \\
& create(x) \longrightarrow init_M(x') \mid init_B(id', n', m') \ : \ x' = x, id' = x, n' = 0, m' = 0. \\
& init_A(id, n, m) \mid fresh(u) \longrightarrow gen_A(id, n', m) \mid fresh(u') \ : \ u' > n', n' > u. \\
& gen_A(id_1, n, m) \mid init_B(id_2, u, v) \longrightarrow wait_A(id_1, n, m) \mid gen_B(id_2', u', v') \ : \ u' = n, v' = v \\
& gen_B(id, n, m) \mid fresh(u) \longrightarrow ready_B(id, n, m') \mid fresh(u') \ : \ u' > m', m' > u. \\
& wait_A(id_1, n, m) \mid ready_B(id_2, u, v) \longrightarrow stop_A(id_1, n, m') \mid stop_B(id_2, u, v) \ : \ n = u, m' = v. \\
& stop_A(id, n, m) \longrightarrow init_A(id', n', m') \ : \ n' = 0, m' = 0, id' = id. \\
& stop_B(id, n, m) \longrightarrow init_B(id', n', m') \ : \ n' = 0, m' = 0, id' = id.
\end{aligned}
$$

Fig. 6. Encoding of Example 1: for simplicity we embed constraints like $x = x'$ into the MSR formulas.

$$
\begin{aligned}
& init \Rightarrow \ldots \Rightarrow fresh(4) \mid init_M(0) \mid init_A(2,0,0) \mid init_B(3,0,0) \\
& \Rightarrow fresh(8) \mid init_M(0) \mid gen_A(2,6,0) \mid init_B(3,0,0) \\
& \Rightarrow fresh(8) \mid init_M(0) \mid wait_A(2,6,0) \mid gen_B(3,6,0) \\
& \Rightarrow \ldots \Rightarrow fresh(16) \mid init_M(0) \mid wait_A(2,6,0) \mid gen_B(3,6,0) \mid init_A(11,0,0)
\end{aligned}
$$

Fig. 7. A run in the encoded program.

is *satisfiable*. For instance, consider the rules $wait_A \xrightarrow{n_A?\langle y\rangle} stop_A[m_A := y]$ and $ready_B \xrightarrow{n_B!\langle m_B\rangle} stop_B[true]$. We first build up a new constraint by conjoining the NC-constraints $y = m_B$ (matching of message templates), and $n_A = n_B, m_A' = y, n_A' = n_A, m_B' = m_B, n_B' = n_B, id_1' = id_1, id_2' = id_2$ (guards and actions of sender and receiver). After eliminating y we obtain the constraint φ defined as $n_B = n_A, m_A' = m_B, n_A' = n_A, m_B' = m_B, n_B' = n_B, id_1' = id_1, id_2' = id_2$ defined over the variables of the two considered threads. This step allows us to *symbolically* represent the passing of names. After this step, we can represent the synchronization of the two threads by using a rule that simultaneously rewrite all instances that satisfy the constraints on the local data expressed by φ, i.e., we obtain the rule

$$
\begin{aligned}
& wait_A(id_1, n_A, m_A) \mid ready_B(id_2, n_B, m_B) \longrightarrow \\
& \qquad stop_A(id_1', n_A', m_A') \mid stop_B(id_2', n_B', m_B') : \varphi
\end{aligned}
$$

The complete translation of Example 1 is shown in Figure 6 (for simplicity we have applied a renaming of variables in the resulting rules). An example of run in the resulting MSR_{NC} specification is shown in Figure 7. Note that, a fresh name is selected between all values strictly greater than the current value of *fresh* (e.g. in the second step $6 > 4$), and then *fresh* is updated to a value strictly greater than all newly generated names (e.g. $8 > 6 > 4$).

Let $\mathscr{T} = \langle P_1, \ldots, P_t\rangle$ be a collection of thread definitions and G_0 be an initial global state. Let $\mathscr{S}$ be the MSR_{NC} specification that results from the translation described in the previous section.

Let $G = \langle N, p_1, \ldots, p_n\rangle$ be a global configuration with $p_i = \langle s_i, v_{i1}, \ldots, v_{ik_i}\rangle$, and let $h : N \rightsquigarrow \mathbb{Q}_+$ be an injective mapping. Then, we define $G^\bullet(h)$ as the MSR_{NC}

configuration

$$s_1(h(v_{11}),\ldots,h(v_{1k_1})) \mid \ldots \mid s_n(h(v_{n1}),\ldots,h(v_{nk_n})) \mid fresh(v)$$

where v is a the first value strictly greater than all values in the range of h. Given an MSR$_{NC}$ configuration $\mathscr{M}$ defined as $s_1(v_{11},\ldots,v_{1k_1}) \mid \ldots \mid s_n(v_{n1},\ldots,v_{nk_n})$ with $s_{ij} \in \mathbb{Q}_+$, let $V(\mathscr{M}) \subseteq \mathbb{Q}_+$ be the set of values occurring in $\mathscr{M}$. Then, given a bijective mapping $f : V(\mathscr{M}) \rightsquigarrow N \subseteq \mathscr{N}$, we define $\mathscr{M}^\bullet(f)$ as the global configuration $\langle N, p_1,\ldots,p_n\rangle$ where $p_i = \langle s_i, f(v_{i1}),\ldots,f(v_{ik_i})\rangle$.

Based on the previous definitions, the following property then holds.

Theorem 2

For every run $G_0G_1\ldots$ in $\mathscr{T}$ with corresponding set of names $N_0N_1\ldots$, there exist sets $D_0D_1\ldots$ and bijective mappings $h_0h_1\ldots$ with $h_i : N_i \rightsquigarrow D_i \subseteq \mathbb{Q}_+$ for $i \geqslant 0$, such that *init* $G_0^\bullet(h_0)G_1^\bullet(h_1)\ldots$ is a run of $\mathscr{S}$. Vice versa, if *init* $\mathscr{M}_0\mathscr{M}_1\ldots$ is a run of $\mathscr{S}$, then there exist sets $N_0N_1\ldots$ in $\mathscr{N}$ and bijective mappings $f_0f_1\ldots$ with $f_i : V(\mathscr{M}_i) \rightsquigarrow N_i$ for $i \geqslant 0$, such that $\mathscr{M}_0^\bullet(f_0)\mathscr{M}_1^\bullet(f_1)\ldots$ is a run in $\mathscr{T}$.

Proof

We first prove that every run in $\mathscr{T}$ is simulated by a run in $\mathscr{S}$.

Let $G_0\ldots G_l$ be a run in $\mathscr{T}$, i.e., a sequence of global states (with associated set of names $N_0\ldots N_l$) such that $G_i \Rightarrow G_{i+1}$ and $N_i \subseteq N_{i+1}$ for $i \geqslant 0$.

We prove that it can be simulated in $\mathscr{S}$ by induction on its length l.

Specifically, suppose that there exist sets of non negative rationals $D_0\ldots D_l$ and bijective mappings $h_0\ldots h_l$ with $h_i : N_i \rightsquigarrow D_i$ for $0 \leqslant i \leqslant l$, such that

$$init\ \widehat{G_0}(h_0)\ldots\widehat{G}_l(h_l)$$

is a run of $\mathscr{S}$. Furthermore, suppose $G_l \Rightarrow G_{l+1}$.

We prove the thesis by a case-analysis on the type of rule applied in the last step of the run.

Let $G_l = \langle N_l, p_1,\ldots,p_r\rangle$ and $p_j = \langle s, n_1,\ldots,n_k\rangle$ be a local configuration for the thread definition $P = \langle Q, s, V, R\rangle$ with $V = \{x_1,\ldots,x_k\}$ and $n_i \in N_l$ for $i : 1,\ldots,k$.

Assignment Suppose there exists a rule $s \xrightarrow{a} s'[\gamma,\alpha]$ in R such that ρ_{p_j} satisfies γ, $G_l = \langle N_l,\ldots,p_j,\ldots\rangle \Rightarrow \langle N_{l+1},\ldots,p'_j,\ldots\rangle = G_{l+1}$ $N_l = N_{l+1}$, $p'_j = \langle s', n'_1,\ldots,n'_k\rangle$, and if $x_i := y_i$ occurs in α, then $n'_i = \rho_{p_j}(y_i)$, otherwise $n'_i = n_i$ for $i : 1,\ldots,k$.

The encoding of the rule returns one MSR_{NC} rule having the form

$$s(x_1,\ldots,x_k) \rightarrow s'(x'_1,\ldots,x'_k) : \gamma', \widehat{\alpha}$$

for every $\gamma' \in \widehat{\gamma}$.

By inductive hypothesis, $\widehat{G}_l(h_l)$ is a multiset of atomic formulas that contains the formula $s(h_l(n_1),\ldots,h_l(n_k))$.

Now let us define h_{l+1} as the mapping from N_l to D_l such that $h_{l+1}(n'_i) = h_l(n_j)$ if $x_i := x_j$ is in α and $h_{l+1}(n'_i) = 0$ if $x_i := \bot$ is in α. Furthermore, let us the define the evaluation

$$\sigma = \langle x_1 \mapsto h_l(n_1),\ldots,x_k \mapsto h_l(n_k), x'_1 \mapsto h_{l+1}(n'_1),\ldots,x'_k \mapsto h_{l+1}(n'_k)\rangle$$

Then, by construction of the set of constraints $\widehat{\gamma}$ and of the constraint $\widehat{\alpha}$, it follows that σ is a solution for $\gamma', \widehat{\alpha}$ for some $\gamma' \in \widehat{\gamma}$. As a consequence, we have that

$$s(n_1, \ldots, n_k) \rightarrow s'(n'_1, \ldots, n'_k)$$

is a ground instance of one of the considered MSR_{NC} rules.

Thus, starting from the MSR_{NC} configuration $\widehat{G_l}(h_l)$, if we apply a rewriting step we obtain a new configuration in which $s(n_1, \ldots, n_k)$ is replaced by $s'(n'_1, \ldots, n'_k)$, and all the other atomic formulas in $\widehat{G_{l+1}}(h_{l+1})$ are the same as in $\widehat{G_l}(h_l)$. The resulting MSR_{NC} configuration coincides then with the definition of $\widehat{G_{l+1}}(h_{l+1})$.

Creation of new names Let us now consider the case of fresh name generation. Suppose there exists a rule $s \xrightarrow{a} s'[x_i := new]$ in R, and let $n \notin N_l$, and suppose $\langle N_l, \ldots, p_j, \ldots \rangle \Rightarrow \langle N_{l+1}, \ldots, p'_j, \ldots \rangle$ where $N_{l+1} = N_l \cup \{v\}$, $p'_j = \langle s', n'_1, \ldots, n'_k \rangle$ where $n'_i = n$, and $n'_j = n_j$ for $j \neq i$.

We note than that the encoding of the previous rule returns the MSR_{NC} rule

$$s(x_1, \ldots, x_k) \mid fresh(x) \rightarrow s'(x'_1, \ldots, x'_k) \mid fresh(x') : \varphi$$

where φ consists of the constraints $y > x'_i, x'_i > x$ and $x'_j = x_j$ for $j \neq i$. By inductive hypothesis, $\widehat{G_l}(h_l)$ is a multiset of atomic formulas that contains the formulas $s(h_l(n_1), \ldots, h_l(n_k))$ and $fresh(v)$ where h_l is a mapping into D_l, and v is the first non-negative rational strictly greater than all values occurring in the formulas denoting processes.

Let v be a non negative rational strictly greater than all values in D_l. Furthermore, let us define $v' = v + 1$ and $D_{l+1} = D_l \cup \{v, v'\}$.

Furthermore, we define h_{l+1} as follows $h_{l+1}(n) = h_l(n)$ for $n \in N_l$, and $h_{l+1}(n'_i) = h_{l+1}(n) = v'$. Furthermore, we define the following evaluation

$$\begin{aligned} \sigma \quad &= \langle x \mapsto v, x_1 \mapsto h_l(n_1), \ldots, x_k \mapsto h_l(n_k), \\ &\quad x' \mapsto v', x'_1 \mapsto h_{l+1}(n'_1), \ldots, x'_k \mapsto h_{l+1}(n'_k) \rangle \end{aligned}$$

Then, by construction of σ and $\widehat{\alpha}$, it follows that σ is a solution for $\widehat{\alpha}$. Thus,

$$s(n_1, \ldots, n_k) \mid fresh(v) \rightarrow s'(n'_1, \ldots, n'_k) \mid fresh(v')$$

is a ground instance of the considered MSR_{NC} rule.

Starting from the MSR_{NC} configuration $\widehat{G_l}(h_l)$, if we apply a rewriting step we obtain a new configuration in which $s(n_1, \ldots, n_k)$ and $fresh(v)$ are substituted by $s'(n'_1, \ldots, n'_k)$ and $fresh(v')$, and all the other atomic formulas in $\widehat{G_{l+1}}(h_{l+1})$ are the same as in $\widehat{G_l}(h_l)$. We conclude by noting that this formula coincides with the definition of $\widehat{G_{l+1}}(h_{l+1})$.

For the sake of brevity we omit the case of *thread creation* whose only difference from the previous cases is the creation of several new atoms instead (with values obtained by evaluating the action) of only one.

Rendez-vous Let $p_i = \langle s, n_1, \ldots, n_k \rangle$ and $p_j = \langle t, m_1, \ldots, m_u \rangle$ two local configurations for threads $P \neq P'$, $n_i \in N_l$ for $i : 1, \ldots, k$ and $m_i \in N_l$ for $i : 1, \ldots, u$.

Suppose $s \xrightarrow{c!m} s'[\gamma, \alpha]$ and $t \xrightarrow{c?m'} t'[\gamma', \alpha']$, where $m = \langle x_{i_1}, \ldots, x_{i_v} \rangle$, and $m' = \langle y_1, \ldots, y_v \rangle$ (all defined over distinct variables) are rules in R.

Furthermore, suppose that ρ_{p_i} satisfies γ, and that ρ' (see definition of the operational semantics) satisfies γ', and suppose that $G_l = \langle N_l, \ldots, p_i, \ldots, p_j, \ldots\rangle \Rightarrow \langle N_{l+1}, \ldots, p'_i, \ldots, p'_j, \ldots\rangle = G_{l+1}$, where $N_{l+1} = N_l$, $p'_i = \langle s', n'_1, \ldots, n'_k\rangle$, $p'_j = \langle t', m'_1, \ldots, m'_u\rangle$, and if $x_i := e$ occurs in α, then $n'_i = \rho_{p_i}(e)$, otherwise $n'_i = n_i$ for $i : 1, \ldots, k$; if $u_i := e$ occurs in α', then $m'_i = \rho'(e)$, otherwise $m'_i = m_i$ for $i : 1, \ldots, u$.

By inductive hypothesis, $\widehat{G_l}(h_l)$ is a multiset of atomic formulas that contains the formulas $s(h_l(n_1), \ldots, h_l(n_k))$ and $t(h_l(m_1), \ldots, h_l(m_u))$.

Now, let us define h_{l+1} as the mapping from N_l to D_l such that $h_{l+1}(n'_i) = h_l(n_j)$ if $x_i := x_j$ is in α, $h_{l+1}(m'_i) = h_l(m_j)$ if $u_i := u_j$ is in α', $h_{l+1}(n'_i) = 0$ if $x_i := \bot$ is in α, $h_{l+1}(m'_i) = 0$ if $u_i := \bot$ is in α'.

Now, let us define σ as the evaluation from N_l to D_l such that

$$\begin{aligned}
&\sigma = \sigma_1 \cup \sigma_2\\
&\sigma_1 = \langle x_1 \mapsto h_l(n_1), \ldots, x_k \mapsto h_l(n_k), u_1 \mapsto h_l(m_1), \ldots, u_u \mapsto h_l(m_u)\rangle\\
&\sigma_2 = \langle x'_1 \mapsto h_{l+1}(n'_1), \ldots, x'_k \mapsto h_{l+1}(n'_k), u'_1 \mapsto h_{l+1}(m'_1), \ldots, u'_u \mapsto h_{l+1}(m'_u)\rangle.
\end{aligned}$$

Then, by construction of the sets of constraints $\widehat{\gamma}, \widehat{\gamma'}, \widehat{\alpha}$ and $\widehat{\alpha'}$ it follows that σ is a solution for the constraint $\exists w'_1 \ldots . \exists w'_p . \theta \wedge \theta' \wedge \widehat{\alpha} \wedge \widehat{\alpha'} \wedge w_1 = w'_1 \wedge \ldots \wedge w_p = w'_p$ for some $\theta \in \widehat{\gamma}$ and $\theta' \in \widehat{\gamma'}$. Note in fact that the equalities $w_i = w'_i$ express the passing of values defined via the evaluation ρ' in the operational semantics.

As a consequence,

$$s(n_1, \ldots, n_k) \mid t(m_1, \ldots, m_u) \rightarrow s'(n'_1, \ldots, n'_k) \mid t'(m'_1, \ldots, m'_u)$$

is a ground instance of one of the considered MSR_{NC} rules.

Thus, starting from the MSR_{NC} configuration $\widehat{G_l}(h_l)$, if we apply a rewriting step we obtain a new configuration in which $s(n_1, \ldots, n_k)$ has been replaced by $s'(n'_1, \ldots, n'_k)$, and $t'(m'_1, \ldots, m'_k)$ has been replaced by $t(m'_1, \ldots, m'_u)$, and all the other atomic formulas are as in $\widehat{G_l}(h_l)$. This formula coincides with the definition of $\widehat{G_{l+1}}(h_{l+1})$.

The proof of completeness is by induction on the length of an MSR run, and by case-analysis on the application of the rules. The structure of the case analysis is similar to the previous one and it is omitted for brevity. $\square$

4 Verification of TDL Programs

Safety and invariant properties are probably the most important class of correctness specifications for the validation of a concurrent system. For instance, in Example 1 we could be interested in proving that every time a session terminates, two instances of thread *Init* and *Resp* have exchanged the two names generated during the session. To prove the protocol correct independently from the number of names and threads generated during an execution, we have to show that from the initial configuration G_0 it is not possible to reach a configuration that violates the aforementioned property. The configurations that violate the property are those in which two instances of *Init* and *Resp* conclude the execution of the protocol exchanging only the first nonce. These configurations can be represented by looking at only two threads and at the relationship among their local data. Thus, we can reduce the verification problem of this safety property to the following problem: Given an initial configuration G_0 we would like to decide if a global configuration that *contains at least* two local configurations having the form $\langle stop_A, i, n, m\rangle$ and $\langle stop_B, i', n', m'\rangle$ with $n' = n$ and $m \neq m'$ for some i, i', n, n', m, m' is reachable. This problem can be viewed as an extension of the *control state reachability problem* defined in (Abdulla and Nylén 2000) in which we consider both control locations and local variables. Although control state reachability is undecidable (see Corollary 1), the encoding of TDL into MSR$_{NC}$ can be used to define a sound and automatic verification methods for TDL programs. For this purpose, we will exploit a verification method introduced for MSR($\mathscr{C}$) (Delzanno 2001, 2005). In the rest of this section we will briefly summarize how to adapt the main results (Delzanno 2001, 2005) to the specific case of MSR$_{NC}$.

Let us first reformulate the control state reachability problem of Example 1 for the aforementioned safety property on the low level encoding into MSR$_{NC}$. Given the MSR$_{NC}$ initial configuration *init* we would like to check that no configuration in $Post^*(\{init\})$ has the following form

$$\{stop_A(a_1, v_1, w_1), stop_B(a_2, v_2, w_2)\} \oplus \mathscr{M}$$

for $a_i, v_i, w_i \in \mathbb{Q}$ $i : 1, 2$ and an arbitrary multiset of ground atoms $\mathscr{M}$. Let us call U the set of *bad MSR$_{NC}$ configurations* having the aforementioned shape. Notice that U is upward closed with respect to multiset inclusion, i.e., if $\mathscr{M} \in U$ and $\mathscr{M} \preccurlyeq \mathscr{M}'$, then $\mathscr{M}' \in U$. Furthermore, for if U is upward closed, so is $Pre(U)$. On the basis of this property, we can try to apply the methodology proposed in (Abdulla and Nylén 2000) to develop a procedure to compute a finite representation R of Pre^*U). For this purpose, we need the following ingredients:

1. a symbolic representation of upward closed sets of configurations (e.g. a set of assertions S whose denotation $[\![S]\!]$ is U);
2. a computable symbolic predecessor operator $SPre$ working on sets of formulas such that $[\![SPre(S)]\!] = Pre([\![S]\!])$;
3. a (decidable) entailment relation Ent to compare the denotations of symbolic representations, i.e., such that $Ent(N, M)$ implies $[\![N]\!] \subseteq [\![M]\!]$. If such a relation Ent exists, then it can be naturally extended to sets of formulas as follows: $Ent^S(S, S')$ if and only if for all $N \in S$ there exists $M \in S'$ such that $Ent(N, M)$ holds (clearly, if Ent is an entailment, then $Ent^S(S, S')$ implies $[\![S]\!] \subseteq [\![S']\!]$).

The combination of these three ingredients can be used to define a verification methods based on backward reasoning as explained next.

Symbolic Backward Reachability Suppose that $M_1, \ldots, M_n$ are the formulas of our assertional language representing the infinite set U consisting of all bad configurations. The symbolic backward reachability procedure (SBR) procedure computes a chain $\{I_i\}_{i \geqslant 0}$ of sets of assertions such that

$$\begin{aligned} I_0 &= \{M_1, \ldots, M_n\} \\ I_{i+1} &= I_i \cup SPre(I_i) \quad \text{for } i \geqslant 0 \end{aligned}$$

The procedure SBR stops when $SPre$ produces only redundant information, i.e., $Ent^S(I_{i+1}, I_i)$. Notice that $Ent^S(I_i, I_{i+1})$ always holds since $I_i \subseteq I_{i+1}$.

Symbolic Representation In order to find an adequate represention of infinite sets of MSR_{NC} configurations we can resort to the notion of *constrained configuration* introduced (Delzanno 2001) for the language scheme $\mathrm{MSR}(\mathscr{C})$ defined for a generic constraint system $\mathscr{C}$. We can instantiate this notion with NC constraints as follows. A constrained configuration over $\mathscr{P}$ is a formula

$$p_1(x_{11}, \ldots, x_{1k_1}) \mid \ldots \mid p_n(x_{n1}, \ldots, x_{nk_n}) : \varphi$$

where $p_1, \ldots, p_n \in \mathscr{P}$, $x_{i1}, \ldots, x_{ik_i} \in \mathscr{V}$ for any $i : 1, \ldots n$ and φ is an NC-constraint. The denotation a constrained configuration $M \doteq (\mathscr{M} : \varphi)$ is defined by taking the upward closure with respect to multiset inclusion of the set of ground instances, namely

$$[\![M]\!] = \{\mathscr{M}' \mid \sigma(\mathscr{M}) \preccurlyeq \mathscr{M}',\ \sigma \in Sol(\varphi)\}$$

This definition can be extended to sets of MSR_{NC} constrained configurations with *disjoint variables* (we use variable renaming to avoid variable name clashing) in the natural way.

In our example the following set S_U of MSR_{NC} constrained configurations (with distinct variables) can be used to finitely represent all possible violations U to the considered safety property

$$\begin{array}{rll} S_U = \{ & stop_A(i_1, n_1, m_1) \mid stop_B(i_2, n_2, m_2) & : \ n_1 = n_2, m_1 > m_2 \\ & stop_A(i_1, n_1, m_1) \mid stop_B(i_2, n_2, m_2) & : \ n_1 = n_2, m_2 > m_1\} \end{array}$$

Notice that we need two formulas to represent $m_1 \neq m_2$ using a disjunction of $>$ constraints. The MSR_{NC} configurations $stop_B(1, 2, 6) \mid stop_A(4, 2, 5)$, and $stop_B(1, 2, 6) \mid stop_A(4, 2, 5) \mid wait_A(2, 7, 3)$ are both contained in the denotation of S_U. Actually, we have that $[\![S_U]\!] = U$. This symbolic representation allows us to reason on infinite sets of MSR_{NC} configurations, and thus on global configurations of a TDL program, forgetting the actual number or threads of a given run.

To manipulate constrained configurations, we can instantiate to NC-constraints the symbolic predecessor operator $SPre$ defined for a generic constraint system in (Delzanno 2005). Its definition is also given in Section Appendix A in Appendix. From the general properties proved in (Delzanno 2005), we have that when applied to a finite set of MSR_{NC} constrained configurations S, $SPre_{NC}$ returns a finite set of constrained configuration such that $[\![SPre_{NC}(S)]\!] = Pre([\![S]\!])$, i.e., $SPre_{NC}(S)$ is a symbolic representation of the immediate predecessors of the configurations in the denotation (an upward closed set) of S. Similarly we can instantiate the generic entailment operator defined in (Delzanno 2005) to MSR_{NC} constrained configurations so as to obtain an a relation Ent such that $Ent_{NC}(N, M)$ implies $[\![N]\!] \subseteq [\![M]\!]$. Based on these properties, we have the following result.

Proposition 1
Let $\mathscr{T}$ be a TDL program with initial global configuration G_0, Furthermore, let $\mathscr{S}$ be the corresponding MSR_{NC} encoding. and S_U be the set of MSR_{NC} constrained configurations denoting a given set of bad TDL configurations. Then,

$init \notin SPre^*_{NC}(S_U)$ if and only if there is no finite run $G_0 \ldots G_n$ and mappings $h_0, \ldots, h_n$ from the names occurring in G to non-negative rationals such that $init^\bullet\ G^\bullet_0(h_0) \ldots G^\bullet_n(h_n)$ is a run in $\mathscr{S}$ and $G^\bullet_n(h_n) \in [\![U]\!]$.

Proof
Suppose $init \notin SPre^*_{NC}(U)$. Since $[\![SPre_{NC}(S)]\!] = pre([\![S]\!])$ for any S, it follows that there cannot exist runs $init\mathscr{M}_0 \ldots \mathscr{M}_n$ in $\mathscr{S}$ such that $\mathscr{M}_n \in [\![U]\!]$. The thesis then follows from the Theorem 2. □

As discussed elsewhere (Bozzano and Delzanno 2002), we have implemented our verification procedure based on *MSR* and *linear constraints* using a CLP system with linear arithmetics. By the translation presented in this paper, we can now reduce the verification of safety properties of multithreaded programs to a fixpoint computation built on *constraint operations*. As example, we have applied our CLP-prototype to automatically verify the specification of Figure 6. The unsafe states are those described in Section 4. Symbolic backward reachability terminates after 18 iterations and returns a symbolic representation of the fixpoint with 2590 constrained configurations. The initial state *init* is not part of the resulting set. This proves our original thread definitions correct with respect to the considered safety property.

4.1 An Interesting Class of TDL Programs

The proof of Theorem 1 shows that verification of safety properties is undecidable for TDL specifications in which threads have several local variables (they are used to create linked lists). As mentioned in the introduction, we can apply the sufficient conditions for the termination of the procedure SBR given in (Bozzano and Delzanno 2002; Delzanno 2005) to identify the following interesting subclass of TDL programs.

Definition 4
A monadic TDL thread definition $P = \langle Q, s, V, R \rangle$ is such that V is at most a singleton, and every message template in R has at most one variable.

A monadic thread definition can be encoded into the monadic fragment of MSR$_{NC}$ studied in (Delzanno 2005). Monadic MSR$_{NC}$ specifications are defined over atomic formulas of the form p or $p(x)$ with p is a predicate symbol and x is a variable, and on atomic constraints of the form $x = y$, and $x > y$. To encode a monadic TDL thread definitions into a Monadic MSR$_{NC}$ specification, we first need the following observation. Since in our encoding we only use the constant 0, we first notice that we can restrict our attention to MSR$_{NC}$ specifications in which constraints have no constants at all. Specifically, to encode the generation of fresh names we only have to add an auxiliary atomic formula $zero(z)$, and refer to it every time we need to express the constant 0. As an example, we could write rules like

$$init \longrightarrow fresh(x) \mid init_M(y) \mid zero(z) \ \ : \ x > z, y = z$$

for initialization, and

$$create(x) \mid zero(z) \longrightarrow init_M(x') \mid init_A(id', n', m') \mid zero(z) \ \ : \\ x' = x, id' = x, n' = z, m' = z, z' = z$$

for all assignments involving the constant 0. By using this trick an by following the encoding of Section 3, the translation of a collection of monadic thread definitions directly returns a *monadic* MSR$_{NC}$ specification. By exploiting this property, we obtain the following result.

Theorem 3
The verification of safety properties whose violations can be represented via an upward closed set U of global configurations is decidable for a collection $\mathscr{T}$ of monadic TDL definitions.

Proof
Let $\mathscr{S}$ be the MSR$_{NC}$ encoding of $\mathscr{T}$ and S_U be the set of constrained configuration such that $S_U = U$. The proof is based on the following properties. First of all, the MSR$_{NC}$ specification $\mathscr{S}$ is monadic. Furthermore, as shown in (Delzanno 2005), the class of monadic MSR$_{NC}$ constrained configurations is closed under application of the operator $SPre_{NC}$. Finally, as shown in (Delzanno 2005), there exists an entailment relation $CEnt$ for monadic constrained configurations that ensures the termination of the SBR procedure applied to a monadic MSR$_{NC}$ specification. Thus, for the monadic MSR$_{NC}$ specification $\mathscr{S}$, the chain defined as $I_0 = S_U$, $I_{i+1} = I_i \cup SPre(I_i)$ always reaches a point $k \geqslant 1$ in which $CEnt^S(I_{k+1}, I_k)$, i.e. $[\![I_k]\!]$ is a fixpoint for Pre. Finally, we note that we can always check for membership of *init* in the resulting set I_k. $\square$

As shown in (Schnoebelen 2002), the complexity of verification methods based on symbolic backward reachability relying on the general results in (Abdulla and Nylén 2000; Finkel and Schnoebelen 2001) is non primitive recursive.

5 Conclusions and Related Work

In this paper we have defined the theoretical grounds for the possible application of constraint-based symbolic model checking for the automated analysis of abstract models of multithreaded concurrent systems providing name generation, name mobility, and unbounded control. Our verification approach is based on an encoding into a low level formalism based on the combination of multiset rewriting and constraints that allows us to naturally implement name generation, value passing, and dynamic creation of threads. Our verification method makes use of symbolic representations of infinite set of system states and of symbolic backward reachability. For this reason, it can be viewed as a conservative extension of traditional finite-state model checking methods. The use of symbolic state analysis is strictly related to the analysis methods based on abstract interpretation. A deeper study of the connections with abstract interpretation is an interesting direction for future research.

Related Work The high level syntax we used to present the abstract models of multithreaded programs is an extension of the communicating finite state machines used in protocol verification (Bochmann 1978), and used for representing abstraction of multithreaded software programs (Ball et al. 2001). In our setting we enrich the formalism with local variables, name generation and mobility, and unbounded control.

Our verification approach is inspired by the recent work of Abdulla and Jonsson. In (Abdulla and Jonsson 2003), Abdulla and Jonsson proposed an assertional language for Timed Networks in which they use dedicated data structures to symbolically represent configurations parametric in the number of tokens and in the *age* (a real number) associated to tokens. In (Abdulla and Nylén 2000), Abdulla and Nylén formulate a symbolic algorithm using *existential zones* to represent the state-space of Timed Petri Nets. Our approach generalizes the ideas of (Abdulla and Jonsson 2003; Abdulla and Nylén 2000) to systems specified via multiset rewriting and with more general classes of constraints. In (Abdulla and Jonsson 2001), the authors apply similar ideas to (unbounded) channel systems in which messages can vary over an infinite *name* domain and can be stored in a finite (and fixed a priori) number of data variables. However, they do not relate these results to multithreaded programs. Multiset rewriting over first order atomic formulas has been proposed for specifying security protocols by Cervesato et al. in (Cervesato et al. 1999). The relationships between this framework and concurrent languages based on process algebra have been recently studied in (Bistarelli et al. 2005).

Apart from approaches based on Petri Net-like models (as in (German and Sistla 1992; Ball et al. 2001)), networks of *finite-state* processes can also be verified by means of automata theoretic techniques as in (Bouajjani et al. 2000). In this setting the set of possible *local states* of individual processes are abstracted into a *finite alphabet*. Sets of global states are represented then as *regular languages*, and transitions as relations on languages. Differently from the automata theoretic approach, in our setting we handle parameterized systems in which individual components have local variables that range over *unbounded* values. The use of constraints for the verification of concurrent systems is related to previous works connecting Constraint Logic Programming and verification, see e.g. (Delzanno and Podelski 1999). In this setting transition systems are encoded via CLP programs used to encode the *global* state of a system and its updates. In the approach proposed in (Delzanno 2001; Bozzano and Delzanno 2002), we refine this idea by using multiset rewriting and constraints to *locally* specify updates to the *global* state. In (Delzanno 2001), we defined the general framework of multiset rewriting with constraints and the corresponding symbolic analysis technique. The language proposed in (Delzanno 2001) is given for a generic constraint system $\mathscr{C}$ (taking inspiration from *CLP* the language is called $MSR(\mathscr{C})$). In (Bozzano and Delzanno 2002), we applied this formalism to verify properties of mutual exclusion protocols (variations of the *ticket algorithm*) for systems with an arbitrary number of processes. In the same paper we also formulated sufficient conditions for the termination of the backward analysis. The present paper is the first attempt of relating the low level language proposed in (Delzanno 2001) to a high level language with explicit management of names and threads.

Acknowledgments

The author would like to thank Ahmed Bouajjani, Andrew Gordon, Fabio Martinelli, Catuscia Palamidessi, Luca Paolini, and Sriram Rajamani and the anonymous reviewers for several fruitful comments and suggestions.

Appendix A Symbolic Predecessor Operator

Given a set of MSR$_{NC}$ configurations S, consider the MSR$_{NC}$ *predecessor* operator $Pre(S) = \{\mathcal{M} | \mathcal{M} \Rightarrow_{MSR} \mathcal{M}', \mathcal{M}' \in S\}$. In our assertional language, we can define a symbolic version $SPre_{NC}$ of Pre defined on a set S containing MSR$_{NC}$ constrained multisets (with *disjoint* variables) as follows:

$$\begin{array}{lll} SPre_{NC}(S) = \{\ (\mathcal{A} \oplus \mathcal{N}\ :\ \xi)\ | & & (\mathcal{A} \longrightarrow \mathcal{B} : \psi) \in \mathcal{R},\ \ (\mathcal{M} : \varphi) \in S, \\ & & \mathcal{M}' \preccurlyeq \mathcal{M},\ \ \mathcal{B}' \preccurlyeq \mathcal{B}, \\ & & (\mathcal{M}' : \varphi)\ =_\theta\ (\mathcal{B}' : \psi),\ \ \mathcal{N} = \mathcal{M} \ominus \mathcal{M}', \\ & & \xi \equiv (\exists x_1 \ldots x_k.\theta) \\ & & \text{and } x_1, \ldots, x_k \text{ are all variables not in } \mathcal{A} \oplus \mathcal{N}\}. \end{array}$$

where $=_\theta$ is a matching relation between constrained configurations that also takes in consideration the constraint satisfaction, namely

$$(A_1 \mid \ldots \mid A_n : \varphi)\ =_\theta\ (B_1 \mid \ldots \mid B_m : \psi)$$

provided $m = n$ and there exists a permutation $j_1, \ldots, j_n$ of $1, \ldots, n$ such that the constraint $\theta = \varphi \wedge \psi \wedge \bigwedge_{i=1}^n A_i = B_{j_i}$ is *satisfiable*; here $p(x_1, \ldots, x_r) = q(y_1, \ldots, y_s)$ is an abbreviation for the constraints $x_1 = y_1 \wedge \ldots \wedge x_r = y_s$ if $p = q$ and $s = r$, $false$ otherwise.

As proved in (Delzanno 2005), the symbolic operator $SPre_{NC}$ returns a set of MSR$_{NC}$ constrained configurations and it is correct and complete with respect to Pre, i.e., $[\![SPre_{NC}(S)]\!] = Pre([\![S]\!])$ for any S. It is important to note the difference between $SPre_{NC}$ and a simple backward rewriting step.

For instance, given the constrained configurations M defined as $p(x,z) \mid f(y)\ :\ z > y$ and the rule $s(u,m) \mid r(t,v) \rightarrow p(u',m') \mid r(t',v')\ \ :\ u = t, m' = v, v' = v, u' = u, t' = t$ (that simulates a rendez-vous (u, t are channels) and value passing ($m' = v$)), the application of $SPre$ returns $s(u,m) \mid r(t,v) \mid f(y)\ :\ \ u = t, v > y$ as well as $s(u,m) \mid r(t,v) \mid p(x,z) \mid f(y)\ :\ u = t, x > y$ (the common multiset here is ϵ).

References

Abdulla, P. A., Cerāns, K., Jonsson, B., and Tsay, Y.-K. 1996. General Decidability Theorems for Infinite-State Systems. In *Proceedings 11th Annual International Symposium on Logic in Computer Science (LICS'96)*. IEEE Computer Society Press, New Brunswick, New Jersey, 313–321.

Abdulla, P. A. and Jonsson, B. 2001. Ensuring Completeness of Symbolic Verification Methods for Infinite-State Systems. *Theoretical Computer Science 256,* 1-2, 145–167.

Abdulla, P. A. and Jonsson, B. 2003. Model checking of systems with many identical timed processes. *Theoretical Computer Science 290,* 1, 241–264.

Abdulla, P. A. and Nylén, A. 2000. Better is Better than Well: On Efficient Verification of Infinite-State Systems. In *Proceedings 15th Annual International Symposium on Logic in Computer Science (LICS'00)*. IEEE Computer Society Press, Santa Barbara, California, 132–140.

Ball, T., Chaki, S., and Rajamani, S. K. 2001. Parameterized Verification of Multithreaded Software Libraries. In *7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001), Genova, Italy, April 2-6,*. LNCS, vol. 2031. Springer-Verlag, 158–173.

Bistarelli, S., Cervesato, I., Lenzini, G., and Martinelli, F. 2005. Relating multiset rewriting and process algebras for security protocol analysis. *Journal of Computer Security 13,* 1, 3–47.

Bochmann, G. V. 1978. Finite state descriptions of communicating protocols. *Computer Networks 2*, 46–57.

Bouajjani, A., Jonsson, B., Nilsson, M., and Touili, T. 2000. Regular Model Checking. In *Proceedings 12th International Conference on Computer Aided Verification (CAV'00)*, E. A. Emerson and A. P. Sistla, Eds. LNCS, vol. 1855. Springer-Verlag, Chicago, Illinois, 403–418.

Bozzano, M. and Delzanno, G. 2002. Algorithmic verification of invalidation-based protocols. In *14th International Conference on Computer Aided Verification, CAV '02*. Lecture Notes in Computer Science, vol. 2404. Springer.

Cervesato, I., Durgin, N., Lincoln, P., Mitchell, J., and Scedrov, A. 1999. A Meta-notation for Protocol Analysis. In *12th Computer Security Foundations Workshop (CSFW'99)*. IEEE Computer Society Press, Mordano, Italy, 55–69.

Delzanno, G. 2001. An Assertional Language for Systems Parametric in Several Dimensions. In *Verification of Parameterized Systems – VEPAS 2001*. ENTCS, vol. 50.

Delzanno, G. 2005. Constraint Multiset Rewriting. Tech. Rep. TR-05-08, Dipartimento Informatica e Scienze dell'Informazione, Università di Genova, Italia.

Delzanno, G. and Podelski, A. 1999. Model checking in CLP. In *Proceedings 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*. Lecture Notes in Computer Science, vol. 1579. Springer-Verlag, Amsterdam, The Netherlands, 223–239.

Finkel, A. and Schnoebelen, P. 2001. Well-Structured Transition Systems Everywhere! *Theoretical Computer Science 256,* 1-2, 63–92.

German, S. M. and Sistla, A. P. 1992. Reasoning about Systems with Many Processes. *Journal of the ACM 39,* 3, 675–735.

Gordon, A. D. 2001. Notes on nominal calculi for security and mobility. In *Foundations of Security Analysis and Design, Tutorial Lectures*. Lecture Notes in Computer Science, vol. 2171. Springer, 262–330.

Kesten, Y., Maler, O., Marcus, M., Pnueli, A., and Shahar, E. 2001. Symbolic model checking with rich assertional languages. *Theoretical Computer Science 256,* 1, 93–112.

Schnoebelen, P. 2002. Verifying Lossy Channel Systems has Nonprimitive Recursive Complexity. *Information Processing Letters 83,* 5, 251–261.