Electronic Notes in Theoretical Computer Science 50 No. 4 (2001) – Proc. VEPAS 2001
URL: http://www.elsevier.nl/locate/entcs/volume50.html 15 pages

An Assertional Language for the Verification of Systems Parametric in Several Dimensions (Preliminary Results)

G. Delzanno [1]

Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova, Via Dodecaneso 35, 16146 Genova, Italy

Abstract

We propose a *rich assertional language* to be used for symbolic verification of systems with *several parametric dimensions.* Our approach combines notions coming from different fields. We use Colored Petri Nets [16] to describe nets of processes carrying structured data. We combine concepts coming from *constraint* programming [23] and *multiset rewriting* [19] to finitely and concisely represent transitions and *infinite* collection of states of Colored Petri Nets. Finally, we incorporate these concepts in the verification technique based on *backward reachability* and *upward-closed sets* of [1,12]. We obtain a procedure that can be used as an automatic support for attacking *parameterized* verification problems. We apply these ideas to verify safety properties of a parameterized mutual exclusion algorithm. A number of open questions arise from our preliminary experiments, finding an adequate counterpart of our framework in the world of automated deduction being among the more interesting ones.

1 Introduction

Given a specification of a concurrent system, the challenge of *parameterized verification* is to verify a *property* for any of its possible *initial* configurations see (e.g. [2,6,10,8,14]). To illustrate the problem on a practical example, let us consider Petri Nets as abstract model of concurrent systems. Via the *abstraction* that maps processes to tokens, we can describe the behaviour of our system via *places* and *transitions*, and a possible initial configuration as an instance of a *parametric initial marking* (e.g., where $K \geq 1$ tokens are in a given place). The goal here is to verify a given property for *any* value of the parameters in the initial marking (e.g. for any value of K). For *safety* properties, the previous problem can be

[1] Email: giorgio@disi.unige.it

solved using techniques like Karp-Miller's *coverability tree* [17]. Parameterized verification problems for a wider range of concurrent systems like Lossy FIFO Systems [1,12], Timed Petri Nets [2], Lossy Vector Addition Systems [4], Broadcast Protocols [11], and Coherence Protocols [8] can be solved using the theory of well-structured systems [1,12]. This approach fits well in the general framework of symbolic model checking with *rich assertional languages* proposed in [18]. An assertional language is a formalism used to symbolically represent sets of states (e.g. BDDs for finite-state verification) and equipped with *algorithmic procedures* for the operations needed to automatically reason on the specification (e.g. to compute *preconditions*). The method of [1] combines in fact *backward reachability* analysis and symbolic representations of *infinite* collection of states. One of the more interesting feature of the 'backward' approach comes from the following observation: *violations* of safety properties can often be represented via *upward-closed* sets of states. This is a desirable property since in several cases it allows to give *finite representations* of infinite set of states. For instance, an *upward-closed* set of Petri Net markings can be represented via the finite collection of its minimal points. Despite of all these nice properties, models based on Petri Nets are often too abstract: processes here are represented as *black* tokens. However, in many practical cases like mutual exclusion and security protocols processes carry along information like *identifiers* or *time-stamps* that are essential for establishing the *correctness* of the whole system. When data cannot be abstracted away, we can say that a specification becomes parametric in *several dimensions* (e.g. number of processes and values of the time-stamps on each process as in [2]). Though in the general case fully automatic verification of parameterized problems becomes impossible, porting the backward approach to systems with several parametric dimensions is a possible way to enrich the set of tools (e.g. automated deduction, bounded model checking, simulation, and testing) used to reason over these complex systems.

Following this line of thoughts and taking inspiration from [2,9], in this paper we propose an *assertional language* to symbolically represent infinite collections of states for systems parametric in *several dimensions*. For this purpose, we found technically convenient to combine notions coming from different fields like *high level Petri Nets* [16], *constraints* [23], multiset rewriting [19], and Constraint Logic Programming [15]. In particular, the notion of *constraint* is central to our construction. Following the terminology of [15], constraints can be viewed as formulas interpreted over a fixed domain, and for which one uses specialized decision procedures, e.g., to test *satisfiability*, and *entailment*. More in detail, our approach works as follows. To smoothly extend the ideas used in [1,2,8,14], we adopt Colored Petri Nets (CPNs) [16] as our general model of parameterized concurrent systems. Basically, CPNs are Petri Nets in which tokens can be *colored* with structured data. Differently from [16] however, our presentation of CPNs is based on a combination of constraints with a restricted form of multiset rewriting. As shown in [19], multiset rewriting allows one to *locally* specify the behaviour of processes in a natural way. Annotating multiset rewriting rules with constraints allows us to *finitely* and *concisely* (e.g. without the need of axioms for arithmetic operations) represent transition relations of CPNs.

Building upon these ideas, we will introduce the new concept of *contrained configuration* as a symbolic representation of *upward-closed* sets of configurations of CPNs: here constraints are used to finitely represent *data* attached to processes, whose *minimal* distribution on the net is described via a multiset of atomic formulas. Constrained configurations play the role of the minimal points used to represent upward-closed sets of Petri Nets markings. Following [18], in order to make an assertional language out of the previous notions, we need symbolic operations to manipulate 'properties', i.e., to compute the *pre-image* and the *union* of sets of CPNs states, and for checking *containment* bewteen sets of CPNs states. In this paper we will show how to build these operations upon the operations of the underlying constraint language. As for CLP systems [15], our assertional language is defined for *any* constraint language equipped with the operations of entailment, satisfiability and existential quantification. Our assertional language can be used as 'constraint system' in the verification framework of [1]. This way we obtain a new backward reachability procedure, that represents an alternative to 'forward' techniques based on the construction of the *occurrece graph* of CPNs [16]. Given the generality of the approach, the resulting procedure is not guaranteed to terminate. However in the same philosophy as *simulation*, *testing*, and *bounded model checking*, it can be used as an automatic support for the analysis of complex protocols. Furthermore, the procedure is robust w.r.t. the backward approach for Petri Nets: if we abstract away the data from all components of our assertional language, we obtain the backward reachability *algorithm* for Petri Nets, whose termination is guaranteed by Dicskon's lemma [1,12]. Following [9], we have implemented the library of symbolic operations and the backward reachability procedure using a CLP system [15] equipped with a *constraint solver* for linear arithmetic constraints over the reals. As a preliminary experiment, we proved safety properties for a *distributed test-and-lock* protocol parameterized on the number of processes, monitors, and resource identifiers.

In our view, multiset rewriting is a possible way to connect parameterized verification and automated deduction. In fact, at least in principle, it is possible to express verification problems as the ones considered in the paper using *AC Rewriting* and specialized theories to represent and handle colors. As we will discuss at the end of the paper, it turns out, however, that the strategies used in existing automated deduction tools are not well-suited to solve problems like the one we are interested in (e.g. computing Pre^* starting from and upward closed set of states). We believe that the application of automated theorem provers to solve parameterized verification problems, as well as finding decidable fragments and accelerations techniques are among the more interesting future direction of our research. In Section 2 and Section 3, we will briefly recall all concepts we need for our assertional language. The language is defined in Section 4. In Section 5 we will discuss how to incorporate the language in the backward approach, and discuss the analysis of our case-study. In Section 6 we discuss related works and discuss a number of open problems we plan to address in our future work. Finally, in Section 7 we will draw some conclusions.

2 Constraints, Multiset Rewriting, and Colored Nets

Our presentation of CPNs is technically but not substantially different from Jensen's definition in [16]. Let us first introduce the notion of *constraint*, and consider a restricted form of multiset (or AC) rewriting.

Constraints. Following [15], a *constraint* φ is a conjunction $c_1, \ldots, c_n$ of atomic formulas with free variables (implicitly existentially quantified) from a denumerable set $\mathcal{V}$. The interpretation domain $\mathcal{D}$ of the constraints is fixed a priori. As an example, linear arithmetic constraints are conjunctions of formulas $k_1 \cdot x_1 + \ldots + k_n \cdot x_n \ rel \ k$, where k, and k_i are an integer constants, $x_i \in \mathcal{V}$ fir $i : 1, \ldots, n$ and *rel* is a relational operator. An *evaluation* is an assignment σ which maps variables in $\mathcal{V}$ to values in $\mathcal{D}$. The application of an evaluation σ is extended to a formula φ containing variables in $\mathcal{V}$, written $\sigma(\varphi)$, in the natural way. A *solution* for a constraint φ is an evaluation σ (restricted to the variables in φ) such that $\sigma(\varphi) \equiv true$ (in the example $\langle x, y \rangle \rightsquigarrow \langle 2, 1 \rangle$ is a solution for φ). The set of solutions of a constraints is denoted as $Sol(\varphi)$. In this paper we will restrict our attention to constraint languages and domains equipped with *decision* procedures for testing entailment and satisfiability. These operations are defined as follows: φ is *satisfiable* if and only if $Sol(\varphi) \neq \emptyset$; φ *entails* ψ if and only if $Sol(\varphi) \subseteq Sol(\psi)$. Furthermore, we will assume to have an algorithm for variable elimination such that given φ and a variable x it returns a constraint $\exists x.\varphi$ whose solutions are obtained by projecting the solutions of φ over $Var(\varphi) \setminus \{x\}$. For instance, let φ be $x \geq y, x \geq 1, y \geq 1$, then φ is *satisfiable* and *entails* $x \geq 1, y \geq 1$; furthermore $\exists y.\varphi$ is equivalent to the constraint $x \geq 1$. For linear constraints, we can use the *Symplex* and *Fourier-Motzkin* as procedures for satisfiability, entailment, and variable elimination [23].

Multiset Rewriting. In this paper we consider the following restricted form of rewriting. Let $\mathcal{P}$ be a finite set of symbols, $\mathcal{V}$ be a denumerable set of variables, and $\mathcal{D}$ be a set of values. An *atomic formula* $p(\vec{x})$ is such that $p \in \mathcal{P}$, and $\vec{x} = \langle x_1, \ldots, x_n \rangle$ is a vector of variables in $\mathcal{V}$. A *ground* atomic formula is obtained by substituting *values* $\mathcal{D}$ to variables. We will use $\cdot|\cdot$ as multiset constructor (*associative* and *commutative*), ϵ being the empty multiset. Furthermore, $\oplus$ will denote the *multiset union*; and $\ominus$ will denote the *multiset difference* (e.g. $(p|p|q) \ominus (p) = (p|q)$). In the rest of the paper we will use $\mathcal{M}$, $\mathcal{N}$, ... to denote *multisets* of atomic formulas. A multiset rewriting rule annotated with constraints has the following form:

$$p_1(\vec{x_1}) \mid \ldots \mid p_n(\vec{x_n}) \longrightarrow q_1(\vec{y_1}) \mid \ldots \mid q_m(\vec{y_m}) \ : \ \varphi,$$

where $p_i, q_j \in \mathcal{P}$, and φ is a constraint with variables in $\vec{x_1}, \ldots, \vec{x_n}, \vec{y_1}, \ldots, \vec{y_m}$. A rule $\mathcal{M} \longrightarrow \mathcal{M}' : \varphi$ denotes a (possibly) infinite collection of *ground rules* formally defined as follows:

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

For instance, let R be the rule $p(x) \mid r(y) \longrightarrow t(x') : x \geq 1, y \geq 0, x' = x - 1$, then, e.g, $p(1) \mid r(0) \longrightarrow t(0) \in Inst(R)$. Finally, given two multisets of ground atoms $\mathcal{M}_1$ and $\mathcal{M}_2$, one step of rewriting is defined via the following relation:

$\mathcal{M}_1 \Rightarrow \mathcal{M}_2$ if and only if there exists a multiset of ground atomic formulas $\mathcal{Q}$ s.t. $\mathcal{M}_1 = \mathcal{N}_1 \oplus \mathcal{Q}$, $\mathcal{M}_2 = \mathcal{N}_2 \oplus \mathcal{Q}$, and $\mathcal{N}_1 \longrightarrow \mathcal{N}_2 \in Inst(\mathcal{R})$.

Colored Petri Nets (CPNs). Based on the previous definition, a Colored Petri Net can be viewed as a tuple $\langle \mathcal{P}, \mathcal{R} \rangle$, where $\mathcal{P}$ is a set of predicates representing *place names*, and $\mathcal{R}$ is a set of multiset rewriting rules representing *transitions*. In the rest of the paper we will call *configuration* a multiset of *ground atomic formulas*. Configurations are the 'generalization' of Petri Net markings, where a token in place p and data $\vec{d}$ is represented via the atomic formula $p(\vec{d})$. Multiset rewriting rules allow us to *locally* model *rendez-vous* and *internal* actions of processes (tokens), independently from the *global state* of the system. The constraint in a transition denotes the relation between the the data of different tokens. A transition can be fired only if the current configuration satisfies φ. By construction, a CPN implicitly defines a *family* of transition systems, each one obtained by fixing an *initial configuration.* An initialized CPN is a tuple $\langle \mathcal{P}, \mathcal{R}, \mathcal{I} \rangle$ where $\langle \mathcal{P}, \mathcal{R} \rangle$ is a CPN, and $\mathcal{I}$ is the *set* of initial configurations. The *occurrence sequences* (runs) of an initialized CPN are then defined via the rewriting relation $\Rightarrow$ defined in the previous section. Let S be a set of configurations, then the immediate *successor* operator is defined as

$$Post(S) \;=\; \{\; \mathcal{M}' \mid \mathcal{M} \Rightarrow \mathcal{M}',\; \mathcal{M} \in S \;\},$$

whereas the immediate *predecessor* operator *Pre* is defined as

$$Pre(S) \;=\; \{\; \mathcal{M} \mid \mathcal{M} \Rightarrow \mathcal{M}',\; \mathcal{M}' \in S \;\}.$$

Finally, the *reachability set* is defined using the transitive closure of the *Post* operator as follows $\mathcal{O} = Post^*(\mathcal{I})$.

2.1 A Case-study.

We consider here a *distributed test-and-lock* protocol for a net with multiple resources each one controlled by a monitor. Each resource is labelled with a nonnegative integer. A process can non-deterministically request any resource. We use the predicates *think*, $wait(x)$, and $use(x)$ (x=resource id) to denote the current state of processes, while a predicate $m(x, t)$ is used to specify a monitor for the resource x with value t for the semaphor. As constraint language, we use linear constraints interpreted over nonegative numbers. We give the specification in two steps. First, we specify all the initial configurations using the predicate *start* as follows.

$$\begin{array}{lcll} \mathit{start} & \longrightarrow & \mathit{start} \mid \mathit{think} & : \mathit{true}. \qquad (1)\\ \mathit{start} & \longrightarrow & \mathit{start} \mid m(x,t) & : x \geq 0, t = 0. \qquad (2)\\ \mathit{start} & \longrightarrow & \epsilon & : \mathit{true}. \qquad (3) \end{array}$$

A run from *start* generates a configuration with an arbitrary number of thinking processes, and *idle* monitors. As from the rule (3), *start* is not needed after the initialization phase. The core of the protocol is as follows:

$$\begin{array}{rcll} \mathit{think} & \longrightarrow & \mathit{wait}(x) & : x \geq 0. \qquad (4)\\ \mathit{wait}(x) & \longrightarrow & \mathit{think} & : x \geq 0. \qquad (5)\\ \mathit{wait}(x) \mid m(x,t) & \longrightarrow & \mathit{use}(x) \mid m(x,t') & : t = 0, t' = 1. \qquad (6)\\ \mathit{use}(x) \mid m(x,t) & \longrightarrow & \mathit{think} \mid m(x,t') & : t = 1, t' = 0. \qquad (7) \end{array}$$

The rules work as follows: (4) a *thinking* process asks for a resource with identifier x (a nonnegative integer, chosen non-deterministically), moving to the state $\mathit{wait}(x)$; (5) waiting processes can choose to go back thinking; (6) a resource x is assigned to a waiting process provided the semaphore of its monitor is not locked (i.e. the monitor is idle); (7) when a process releases the resource the monitor resets the lock. Note that in the previous specification we simply express the *local* interaction between one process and one monitor (the power of multiset rewriting). The initial configuration of the protocol consists of the configuration *start*, that in turn generates a configurations in which all processes are thinking, and all monitors are idle. For instance, a possible run from *start* is as follows (we will use $\Rightarrow_i$ to indicate the application of rule (i)):

$$\mathit{start} \Rightarrow_1 \mathit{start} \mid \mathit{think} \Rightarrow_2 \mathit{start} \mid \mathit{think} \mid m(1,0) \Rightarrow_1 \mathit{start} \mid \mathit{think} \mid \mathit{think} \mid m(1,0)$$
$$\Rightarrow_2 \mathit{start} \mid \mathit{think} \mid \mathit{think} \mid m(1,0) \mid m(4,0) \Rightarrow_3 \mathit{think} \mid \mathit{think} \mid m(1,0) \mid m(4,0).$$

Starting form the initial configuration of the previous example, a run involving the other rules is as follows.

$$\mathit{think} \mid \mathit{think} \mid m(1,0) \mid m(4,0) \ \Rightarrow_4 \ \mathit{wait}(1) \mid \mathit{think} \mid m(1,0) \mid m(4,0)$$
$$\Rightarrow_6 \ \mathit{use}(1) \mid \mathit{think} \mid m(1,1) \mid m(4,0) \ \Rightarrow_4 \ \mathit{use}(1) \mid \mathit{wait}(1) \mid m(1,1) \mid m(4,0).$$

3 Parameterized Verification

In this paper we will restrict ourselves to consider verification of *safety* properties. A safety property for a CPN with initial configurations $\mathcal{I}$ can be represented via two sets S_{good} and S_{bad}, that represent the configurations that respectively do and do not satisfy the property. In the forward approach to verification (in the style of Karp and Miller [17,10,16]) one tries to prove that $\mathit{Post}^*(\mathcal{I}) \subseteq S_{good}$, whereas in the backward approach of [1,12] one tries to prove that $\mathit{Pre}^*(S_{bad}) \cap$

$\mathcal{I} = \emptyset$ (where $Post^*/Pre^*$ is the transitive closure of $Post/Pre$). When applied to parameterized verification, the backward approach has two advantages: (1) computing Pre^* can be done *independently* from the initial configuration; (2) bad configurations often form *upward-closed* sets. Let us explain this intuition using our example. The safety property of the multiple test-and-lock protocol is that *only one process* per time can use a given resource; thus, S_{bad} is the set of configurations *containing* the *minimal* violations: $use(n) \mid use(n)$ for some resource-id n. To formalize the idea of 'minimality' of a set, let us introduce the following ordering between configurations:

$$\mathcal{M} \preccurlyeq \mathcal{N} \text{ if and only if } Occ_A(\mathcal{M}) \leq Occ_A(\mathcal{N}) \text{ for any atom } A,$$

where $Occ_A(\mathcal{M})$ is the number of occurrences of A in $\mathcal{M}$ (e.g. $Occ_{use(1)}(\mathcal{M}) = 2$ for $\mathcal{M} = use(1) \mid use(1) \mid think(2)$). A set of configurations S *generates* its *upward* closure $Up(S)$ defined as follows:

$$Up(S) = \{\mathcal{N} \mid \mathcal{M} \preccurlyeq \mathcal{N},\ \mathcal{M} \in S\}.$$

A set S is *upward-closed* whenever $Up(S) = S$ (e.g. S_{bad} in our example). Upward-closed sets of configurations have interesting properties w.r.t. the *predecessor* operator Pre of a CPN.

Proposition 3.1 *$Up(Pre(S)) \subseteq Pre(Up(S))$ for any set S of configurations.*

In general the reverse implication does not hold. As a counter-example, simply take the rule $p \longrightarrow q_1 \mid q_2$ and the singleton set S consisting of the multiset q_1. Then, $Pre(S) = \emptyset$, whereas the multiset p belongs to $Pre(Up(S))$. However, the following property holds.

Corollary 3.2 *If S is upward-closed, then $Up(Pre(S)) = Pre(Up(S))$.*

In other words, the class of upward-closed sets of configurations is closed under the computation of the pre-image. In the following section we will try to exploit these properties to define *assertions* for verification of CPNs.

4 The Assertional Language

Following [18], a rich assertional language (in the context of automatic, or semi-automatic verification) should allow one to symbolically represent and manipulate *properties* expressed via (possibly infinite) sets of states. Boolean formulas and BDDs are classical examples from finite-state verification. In our setting we will use *constraints*, this time to finitely represent minimal configurations of upward-closed sets. For this purpose, we introduce the notion of *constrained configuration* defined as

$$p_1(\vec{x_1}) \mid \ldots \mid p_n(\vec{x_n}) \ :\ \varphi$$

where $p_1, \ldots, p_n \in \mathcal{P}$, and φ is a constraint over the *variables* $\vec{x_1}, \ldots, \vec{x_n}$ (free variables are implicitly existentially quantified). Given a constrained configuration $\mathcal{M} : \varphi$ the *set* of its *ground instances* is defined as

$$Inst(\mathcal{M} : \varphi) = \{\sigma(\mathcal{M}) \mid \sigma \in Sol(\varphi)\}.$$

This definition can be extended to sets of constrained configurations with *disjoint variables* (indicated as $\mathbf{S}, \mathbf{S}', \ldots$) in the natural way. For instance, if $\mathbf{S}_{bad}$ is the singleton containing $use(x)|use(y) : x = y$, then $Inst(\mathbf{S}_{bad})$ is the set of configurations having the form $use(n)|use(n)$ for any n. Note that $Inst(\mathbf{S}_{bad})$ does not model our intuition that constrained configurations should *generate* all violations. Thus, instead of taking the set of instances as 'flat' denotation of a set of constrained configuration $\mathbf{S}$, we choose the rich denotation:

$$[\![\mathbf{S}]\!] = Up(Inst(\mathbf{S})).$$

For instance, in our first example we have that $S_{bad} = [\![\mathbf{S}_{bad}]\!]$ as desired. In the rest of this section we will show how to formulate operations on sets of configurations at the symbolic level of constrained configurations w.r.t. the 'rich' denotation $[\![\cdot]\!]$. We anticipate here that all operations will be parametric on the constraint language used in the specification.

Pre-image computation. In order to handle our symbolic representation, we need a new operator **Pre** such that $[\![\mathbf{Pre}(\mathbf{S})]\!] = Pre([\![S]\!])$. We first introduce a new operator working on sets of configurations as follows:

$$\overline{Pre}(S) = \{\ \mathcal{A} \oplus \mathcal{M}' \mid \mathcal{A} \longrightarrow \mathcal{B} \in Inst(\mathcal{R}),\ \mathcal{M} \in S,\ \mathcal{M}' = \mathcal{M} \ominus \mathcal{B}\ \}.$$

Intuitively, $\overline{Pre}$ treats a configurations as 'representation' of its upward-closure. For instance, consider $\mathcal{D} = \{0, 1\}$, and the rule $p(x) \longrightarrow q(x) : true$. Then, $\overline{Pre}(\{q(1)\})$ returns $p(1)$ as well as $p(0) \mid q(1)$. In fact, $p(0)$ rewtites into $q(1)$, whereas $p(0) \mid q(1)$ rewrites into $q(0) \mid q(1)$ that belongs to the upward-closure of $q(1)$. The new operator satisfies the following property.

Proposition 4.1 *$Pre(Up(S)) \subseteq Up(\overline{Pre}(S))$ for any set S of configurations.*

To lift the definition of $\overline{Pre}$ to the symbolic level, we introduce the notion of *unification* between constrained configurations:

$$(p_1(\vec{x_1}) \mid \ldots \mid p_n(\vec{x_n}) \ :\ \varphi) \ \textbf{unifies with}\ (q_1(\vec{y_1}) \mid \ldots \mid q_m(\vec{y_m}) \ :\ \psi) \ \textbf{via}\ \theta$$

if and only if: *(i)* $m = n$; *(ii)* there exist two permutations $i_1, \ldots, i_n$ and $j_1, \ldots, j_n$ of $1, \ldots, n$ such that $p_{i_k} = q_{j_k}$, and $\vec{x_{i_k}}$ and $\vec{y_{j_k}}$ are tuples of the same size for $k : 1, \ldots, n$; *(iii)* the constraint θ defined as $\varphi \wedge \psi \wedge_{k=1}^{n} \ \vec{x_{i_k}} = \vec{y_{i_k}}$ is *satisfiable* w.r.t. the domain $\mathcal{D}$ taken into consideration.

The operator **Pre** is defined on a set **S** containing constrained multisets (with *disjoint* variables) as follows

$$\begin{aligned}\mathbf{Pre}(\mathbf{S}) = \{\ (\mathcal{A} \oplus \mathcal{N}\ :\ \exists \vec{x}.\theta)\ \mid\ & (\mathcal{A} \longrightarrow \mathcal{B} : \psi) \in \mathcal{R},\ \ (\mathcal{M} : \varphi) \in \mathbf{S},\\ & \mathcal{M}' \preccurlyeq \mathcal{M},\ \ \mathcal{B}' \preccurlyeq \mathcal{B},\\ & (\mathcal{M}' : \varphi)\ \textbf{unifies with}\ (\mathcal{B}' : \psi)\ \textbf{via}\ \theta,\\ & \mathcal{N} = \mathcal{M} \ominus \mathcal{M}',\\ & \text{and } \vec{x} \text{ are all variables not in } \mathcal{A} \oplus \mathcal{N}\}.\end{aligned}$$

For instance, consider $p(x, y) \longrightarrow q(x, y) : x \geq 0, y = 1$. Given the singleton **S** with $q(u, w) : u = 1, w \geq 0$, **Pre**(**S**) should contain $p(x, y) : x = 1, y = 1$ as well as $p(x, y) \mid q(u, w) : x \geq 0, y = 1, u = 1, w \geq 0$ (e.g., $p(4, 1) \mid q(1, 5)$ rewrites into $q(4, 1) \mid q(1, 5) \in [\![\mathbf{S}]\!]$.) The latter constrained multiset can be obtained by setting $\mathcal{M}' = \mathcal{B}' = \epsilon$ (the empty multiset) when applying **Pre** to **S**. The new operator enjoys the following properties

Proposition 4.2 *For any CPN and any set* **S** *of constrained configurations,* $\overline{Pre}(Inst(\mathbf{S})) \subseteq Inst(\mathbf{Pre}(\mathbf{S}))$ *and* $Inst(\mathbf{Pre}(\mathbf{S})) \subseteq Up(\overline{Pre}(Inst(\mathbf{S})))$.

In general $Inst(\mathbf{Pre}(\mathbf{S})) \subseteq \overline{Pre}(Inst(\mathbf{S}))$ does not hold. As a counter-example, take $\mathcal{D} = \{0, 1\}$, the rule $p(x) \longrightarrow q(x) : true$, and a set **S** containing $(q(y) : y = 0)$. Then, $p(x) \mid q(y) : y = 0$ (x free) is in **Pre**(**S**), while its instance $p(0) \mid q(0)$ is not in $\overline{Pre}(Inst(\mathbf{S}))$ (while it is in $Up(\overline{Pre}(Inst(\mathbf{S})))$). From the previous proposition, it follows however that

Theorem 4.3 $[\![\mathbf{Pre}(\mathbf{S})]\!] = Pre([\![\mathbf{S}]\!])$ *for any* **S**.

It is easy to verify that $[\![\mathbf{S}_1]\!] \cup [\![\mathbf{S}_2]\!] = [\![\mathbf{S}_1\ \cup\ \mathbf{S}_2]\!]$. As a consequence of the previous observation and of Theorem 4.3, it follows that we can use **Pre** in order to symbolically compute $Pre^*(S) = S\ \cup\ Pre(S)\ \cup\ \ldots$ (the transitive closure of Pre). Formally, if **S** is the symbolic representation of the upward-closed set S (i.e. $S = [\![\mathbf{S}]\!]$), then $[\![\mathbf{Pre}^*(\mathbf{S})]\!] = Pre^*([\![\mathbf{S}]\!])$.

An Effective Containment Test. To build a procedure for computing (or simply approximating) $\mathbf{Pre}^*(\mathbf{S})$, we need to define a procedure to check the containment of intermediate results. Specifically, given two sets **S** and $\mathbf{S}'$, we need to test whether $[\![\mathbf{S}]\!] \subseteq [\![\mathbf{S}']\!]$ holds. We first note that the previous condition cannot be tested pointwise as shown by the following counter-example. Let $\mathbf{S}_1$ be the singleton containing the constrained configuration $p(x) : x \geq 0$, and $\mathbf{S}_2$ be the set containing $p(x) : x = 0$ and $p(x) : x \geq 1$. Let $\mathcal{D}$ be the set of nonnegative integers. Clearly, $[\![\mathbf{S}_1]\!] \subseteq [\![\mathbf{S}_2]\!]$ holds, however there exists no $\mathcal{M} \in \mathbf{S}_2$ such that $[\![p(x) : x \geq 0]\!] \subseteq [\![\mathcal{M}]\!]$. Though the pointwise subsumption test is not complete, it gives us a sufficient condition to check containment between the denotations of sets of configurations. Furthermore, we can make it 'effective' by introducing

```
Proc $\mathbf{Pre}^*(\mathbf{U}$ : set of constrained configurations$)$
  $\mathbf{S} := \mathbf{U};\ \mathbf{R} := \emptyset;$
  while $\mathbf{S} \neq \emptyset$ do
    remove $(\mathcal{M} : \varphi)$ from $\mathbf{S}$;
    if there are no $(\mathcal{N} : \psi) \in \mathbf{R}$ s.t. $(\mathcal{M} : \varphi)$ entails $(\mathcal{N} : \psi)$ then
        add $(\mathcal{M} : \varphi)$ to $\mathbf{R}$ and set $\mathbf{S} := \mathbf{S} \cup \mathbf{Pre}(\{\mathcal{M} : \varphi\})$;
    endif;
  end_while.
```

Fig. 1. Symbolic backward reachability for CPNs.

the following operator working on constrained configurations:

> $(\mathcal{M} : \varphi)$ **entails** $(\mathcal{N} : \psi)$ if and only if there exists $\exists\mathcal{M}'$ s.t. $\mathcal{M}' \preccurlyeq \mathcal{M}$, $(\mathcal{M}' : \varphi)$ **unifies with** $(\mathcal{N} : \psi)$ **via** θ, and $(\exists\vec{x}.\ \theta)$ *entails* ψ, $\vec{x}$ being all variables not occurring in $\mathcal{M}'$ and $\mathcal{N}$.

For instance, $p(x) \mid q(z) : x \geq 1, z \geq 1$ **entails** $p(x') : x' \geq 0$, since the constraint $\exists z.x \geq 1$ *entails* $x \geq 0$ (modulo renaming, $x \geq 1$ is the unifier for $p(x) : x \geq 1$ and $p(x') : x' \geq 0$). Then, the following proposition holds.

Theorem 4.4 *Let $(\mathcal{M} : \varphi)$ and $(\mathcal{N} : \psi)$ be two constrained configurations. Suppose $(\mathcal{M} : \varphi)$* **entails** *$(\mathcal{N} : \psi)$ holds, then $[\![\mathcal{M} : \varphi]\!] \subseteq [\![N : \psi]\!]$.*

Note that, the reverse implication does not hold. As a counter-example, consider $\mathcal{D}$ as the set of terms built over a, b, and f, and the two constrained configurations $p(x, y) \mid p(z, w) : x = a, y = z, w = f(u)$, and $p(x, w) : x = a, w = f(u)$ (the denotations of the former are contained in those of the latter, while they are not in the **entail** relation).

5 An Automatic Support for Parameterized Verification

Given the generality of the notion of *color*, most of the verification problems for CPNs are undecidable. However as advocated in [16], automatic techniques can still be useful as a support for their validation. Automated deduction, bounded model checking, simulation and testing are all examples of techniques one could use in this sense. Our assertional language can incorporated in the procedure of [1], as shown by the program skeleton in Fig. 1. Fixing the depth of the backward search, the resulting procedure be used, e.g., as an automatic support for searching *bugs*. Interestingly, the procedure can be reduced to the usual *backward reachability algorithm* for Petri Nets by abstracting away *colors* in the data structures and operations. Following [9], we directly implement the operations on constrained multisets, and the procedure of Fig. 1 in the CLP system SICStus

m_1	$\mathit{start} : \mathit{true}.$
m_2	$\mathit{think} \mid \mathit{think} \mid m(x,0) \mid m(x,0) : x \geq 0.$
m_3	$\mathit{use}(x) \mid m(x,1) \mid \mathit{use}(x) : x \geq 0.$
m_4	$\mathit{think} \mid m(x,0) \mid \mathit{wait}(x) \mid m(x,0) : x \geq 0.$
m_5	$\mathit{wait}(x) \mid m(x,0) \mid \mathit{wait}(x) \mid m(x,0) : x \geq 0.$
m_6	$\mathit{think} \mid m(x,0) \mid \mathit{use}(x) : x \geq 0.$
m_7	$\mathit{wait}(y) \mid \mathit{use}(x) \mid \mathit{use}(x) : x \geq 0, y \geq 0.$
m_8	$\mathit{wait}(x) \mid m(x,0) \mid \mathit{use}(x) : x \geq 0.$
m_9	$\mathit{think} \mid \mathit{use}(x) \mid \mathit{use}(y) : x \geq 0.$
m_{10}	$\mathit{use}(x) \mid \mathit{use}(x) : x \geq 0.$

Fig. 2. $\mathbf{Pre}^*(\mathbf{S})$ with $\mathbf{S} = \{\ \mathit{use}(x) \mid \mathit{use}(x) : x \geq 0\ \}$.

Prolog, providing symbolic manipulations of terms via unification, and contraint-operations for different domains (e.g. linear constraints over reals). We used the implementation to study safety properties for our case-study.

Analysis of the case-study. The goal is to verify mutual exclusion for *any* number of processes, monitors, and resources for the distributed test-and-lock protocol of Section 2, i.e., $\mathit{start} \notin [\![\mathbf{Pre}^*(\mathbf{S})]\!]$ where $\mathbf{S}$ is the singleton containing $\mathit{use}(x) \mid \mathit{use}(y) : x = y, x \geq 0$. On such an input, the procedure of Fig. 1 terminates in 8 steps. Counting all redundancies, the fixpoint consists of 413 constrained multisets (making manual computations highly error-prone). In our implementation we automatically remove redundant elements. The result consists of ten elements, as shown in Fig. 2. Observing the history of the backward computation, we note that *start* (generated in the last step) can be derived (applying 4 times **Pre**) from m_2 (generated in the fourth step); however m_2 represents initial configurations only if we let two monitors control the same resource. If we assume that all monitors control *distinct* resources, then we can discharge m_2 and m_0 (automatically, if we apply the invariant to cut the search), and conclude that our protocol ensures mutual exclusion for *any* number of processes, and resources. Though in the example the precondition on the monitors could be derived by looking at the specification, the example shows that the approach could have interesting applications to synthetized invariants for parameterized systems. Finally, it is important to note that the previous property cannot be proved by abstracting away data. In fact, the 'abstract' unsafe configurations containing the marking $\mathit{use} \mid \mathit{use}$ (the abstraction of $\mathbf{S}_{\mathit{bad}}$) can always be reached from abstractions of admissible configurations (e.g. from the abstraction of $\mathit{wait}(1) \mid \mathit{wait}(2) \mid m(1,0) \mid m(2,0)$).

6 Related Work and Open Questions

In their seminal paper [2], Abdulla and Jonsson proposed an assertional language for Timed Petri Nets in which they use dedicated data structures to symbolically represent markings parametric in the number of tokens and in the *age* associated to tokens. In [3], Abdulla and Nylén formulate a symbolic algorithm using *difference constraints* to represent the state-space of Time Petri Nets. Our approach is an attempt to generalize the ideas of [2,3] to problems and constraint systems that do not depend strictly on *time*. In our opinion the combination of *multiset rewriting* and *constraints* is an elegant way to achieve the goal.

Relation with Constraint-based Verification. Our ideas are related to previous works connecting Constraint Logic Programming and verification [13]. In this setting transition systems are encoded via CLP programs of the form A:-B, φ where A and B are atomic formulas used to encode the *global* state of a system, and φ is a constraint modeling state updates, see e.g. [9]. Contrary, here we refine this idea by using multiset rewriting and constraints to *locally* specify the update of the *global* state, and the *synchronization* between different processes (i.e., we only keep the information in A and B strictly relevant to the rule by splitting an atomic formula into a multiset of formulas). The notion of *constrained multiset* extends naturally the notion of *constrained atom* of [9]. However, the *locality* in the representation of rules allows us to consider *rich* denotations (upward-closures) instead of *flat* ones (instances) like in [9]. This way, we can lift the approach to the parameterized case. Concerning extensions of logic programs with aspects of concurrency, the use of *contrained formulas* to define *non-ground* semantics for linear logic programs has been investigated in [5].

Relation with AC Rewriting. The use of *constraints* and the *independence* from the initial configuration are two distinguished features of our approach compared to applications of *AC rewriting* and rewrite logic to simulation and analysis (see e.g. [21]). Backward analysis share some similarities with *saturation*-based procedures for *first-order logic*, with associative and commutative (AC) operators: computing Pre^* here amounts to *saturate* the theory with the set of the logical consequences of a theory. More precisely, we first express a multiset rewriting rule $\mathcal{M} \rightarrow \mathcal{M}'$ (where $\mathcal{M} = a_1 \mid \ldots \mid a_m$, and $\mathcal{M}' = b_1 \mid \ldots \mid b_n$) as the rewriting rule:

$$\mathcal{M} \mid X \ \rightarrow \ \mathcal{M}' \mid X,$$

where $\mid$ is an AC symbol, and X is a *free variable* used to obtain an upward closed set of ground rule instances. A verification problem expressible in terms of upward-closed sets can be represented as the 'reachability' problem: the initial state, say *start*, reaches an instance of the term t_{unsafe} having the form

$$t_1 \mid \ldots \mid t_n \mid Y,$$

where $t_1 \mid \ldots \mid t_n$ denotes the minimal violations, and Y is a *free variable.* As an experiment, we have encoded our example, and tested on the saturation-based theorem prover daTac [22] specialized in AC theories. The prover returned a *don't know* answer to our problem.

Apart from the decidability results for the verification problems of Timed Petri Nets given in [2,3], and some results for *ground* AC-rewriting systems [20], we are not aware of theoretical results concerning decidability of reachability problems for multiset rewriting systems with 'colors'. Though the notion of *contraint* (to symbolically represent data) is not peculiar of systems like daTac, we plan to investigate more deeply the applicability of *automated deduction* problems like the one presented in this paper.

Need of Accelerations. The framework presented in the paper allows us to express examples of mutual exclusion algorithms, with structured colors. As an example, consider the CPN model of the well-know *ticket* mutual exclusion algorithm. Differently from our previous example, the ticket algorithm deals with *one common critical section.* Before entering the critical section, each process gets a ticket and waits until its turn comes. Turns are selected by incrementing a variable everytime a process leaves the critical section. In this example, tickets will be the *colors* of our CPNs. The protocol is parametric on the number of processes and on the values of tickets. We use the predicates $\{think, wait, use\}$ to denote the state of processes, *count* to emit new tickets, and *turn* to establish the current turn. Note that while we need multiset rewriting to handle processes, we should consider only one copy of the two counters (we will discuss later this point). As constraint language we will use *linear arithmetic constraints* interpreted over the *nonnegative* integers. The specification is given as follows.

$$\begin{array}{rcll} think(x) \mid count(t) & \longrightarrow & wait(x') \mid count(t') & : \ x' = t, t' = t+1. \\ wait(x) \mid turn(s) & \longrightarrow & use(x) \mid turn(s) & : \ x \leq s. \\ use(x) \mid turn(s) & \longrightarrow & think(x') \mid turn(s') & : \ x' = x, s' = s+1. \end{array}$$

With the first rule, a process in state *think* represented as gets the current ticket t (stored in $couint(t)$). A new ticket is prepared incrementing t. With the second rule a process requesting enters the critical section whenever its ticket t has a value less or equal than the current turn s (stored in $turn(s)$). With the third rule, a process releases the resource an the current turn is updated. In the initial state of the system all processes are thinking (the initial value of their ticket do not matter), while *turn* and *count* must store the *same initial value.* values of the identifiers of the resources. The set of violations can be represented through the constraint multiset $use(x) \mid use(y) \ : \ x \geq 0, y \geq 0$ denoting all configurations with *at least* two processes in the critical section. Our procedure (enriched with the use of invariants like *every reachable state has at most one occurrence of turn and count*) does not terminate on this example. Furthermore, acceleration operations built on the top of widenings used in for static analysis [7] did not return useful approximations. Finding stronger acceleration operators seems to be an interesting direction for future research.

7 Conclusions

In this paper we made a first attempt to merge notions coming from different research fields (constraints, rewriting, Petri Nets) in order to attack one of the more difficult problem of (manual as well as automated) verification: proving parameterized properties. Our conceptual contribution is the definition of an assertional language for expressing properties of Colored Petri Nets. The concept of *constraint* allows us to define the language on the basis of the operations (entailment, etc.) supported by any system for programming over constraints [23]. The language is designed for the backward approach with upward-closed sets proposed in [1]. This approach represents an alternative to previous validation techniques used for CPNs based on the construction of the *occurrence graph* (forward reachability), and on structural analysis [16].

Acknowledgments. The author would like to thank Sandro Etalle, Marco Bozzano and Gianna Reggio for useful discussions, and Alessandro Armando and Michael Rusinowitch for the experiments with daTac.

References

[1] Abdulla, P. A., K. Cerāns, B. Jonsson and Y.-K. Tsay, *General Decidability Theorems for Infinite-State Systems*, in: *Proc. LICS'96* (1996), pp. 313–321.

[2] Abdulla, P. A. and B. Jonsson, *Verifying networks of timed processes*, in: *Proc. TACAS '98*, LNCS **1384** (1998), pp. 298–312.

[3] Abdulla, P. A. and A. Nylén, *Better is Better than Well: On Efficient Verification of Infinite-State Systems* , in: *Proc. LICS 2000* (2000), pp. 132–140.

[4] Bouajjani, A. and R. Mayr, *Model checking lossy vector addition systems*, in: *Proc. STACS '99*, LNCS **1563** (1999), pp. 323–333.

[5] Bozzano, M., G. Delzanno and M. Martelli, *An effective fixpoint semantics for first-order linear logic programs*, in: *Proc. FLOPS 01'*, LNCS **2024** (2001), pp. 138–152.

[6] Clarke, E., O. Grumberg and S. Jha, *Verifying Parameterized Networks*, TOPLAS **19** (1997), pp. 726–750.

[7] Cousot, P. and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fix-Points*, in: *Proc. POPL'77* (1977), pp. 238–252.

[8] Delzanno, G., *Automatic Verification of Parameterized Cache Coherence Protocols*, in: *Proc. CAV 2000*, LNCS **1855** (2000), pp. 53–68.

[9] Delzanno, G. and A. Podelski, *Model Checking in CLP*, in: *Proc. TACAS '99*, LNCS **1579** (1999), pp. 223–239.

[10] Emerson, E. A. and K. S. Namjoshi, *On the verification of broadcast protocols*, in: *Proc. LICS '98* (1998), pp. 70–80.

[11] Esparza, J., A. Finkel and R. Mayr, *On Model Checking for Non-deterministic Infinite-state Systems*, in: *Proc. LICS '99* (1999), pp. 352–359.

[12] Finkel, A. and P. Schnoebelen, *Well-structured transition systems everywhere!*, Theoretical Computer Science **256** (2001), pp. 63–92.

[13] Fribourg, L., *Constraint Logic Programming Applied to Model Checking*, in: *Proc. LOPSTR '99*, LNCS **1817** (1999), pp. 30–41.

[14] German, S. M. and A. P. Sistla, *Reasoning about Systems with Many Processes*, JACM **39** (1992), pp. 675–735.

[15] Jaffar, J. and M. J. Maher, *Constraint Logic Programming: A Survey*, Journal of Logic Programming **19-20** (1994), pp. 503–582.

[16] Jensen, K., *An Introduction to the Theoretical Aspects of Coloured Petri Nets*, in: *A Decade of Concurrency*, LNCS **803** (1994), pp. 230–272.

[17] Karp, R. M. and R. E. Miller, *Parallel program schemata*, Journal of Computer and System Sciences **3** (1969), pp. 147–195.

[18] Kesten, Y., O. Maler, M. Marcus, A. Pnueli and E. Shahar, *Symbolic model checking with rich assertional languages*, in: *Proc. CAV '97*, LNCS **1254** (1997), pp. 424–435.

[19] Meseguer, J., *Conditioned Rewriting Logic as a Unified Model of Concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[20] Narendran, P. and M. Rusinowitch, *Any ground associative-commutative theory has a finite canonical system*, Journal of Automated Reasoning **17** (1996), pp. 131–143.

[21] Ölveczky, P. C. and J. Meseguer, *Real-Time Maude: A Tool for Simulating and Analyzing Real-Time and Hybrid Systems*, in: *Proc. Workshop on Rewriting Logic and its Applications '00*, ENTCS **36** (2000).

[22] Rusinowitch, M. and L. Vigneron, *Automated Deduction with Associative and Commutative Operators*, Applicable Algebra in Engineering, Communication and Computing **6** (1995), pp. 23–56.

[23] Saraswat, V. and P. Van Henteryck, "Principles and Practice of Constraint Programming," MIT Press, 1995.