An Overview of MSR(чистота): A CLP-based Framework for the Symbolic Verification of Parameterized Concurrent Systems

G. Delzanno

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

Abstract
In recent works we have defined a general framework for the validation of parameterized concurrent systems based on the combination of multiset rewriting and constraints. The class of systems we are interested in consists of concurrent systems parametric in the number of individual components.

Our framework provides the following features: (1) a specification language for the class of concurrent systems taken into consideration; (2) an assertional language to finitely represent infinite sets of configurations; and (3) a sound and fully automatic verification method based on symbolic state exploration.

The verification procedure has been implemented in a Constraint Logic Programming systems, namely Sicstus Prolog and the clp(Q,R) library. CLP provides in fact all necessary operations to manipulate multisets and constraints both as uninterpreted and interpreted objects. Operations over constraints are delegated in fact to the clp(Q,R) library, and encapsulated into Sicstus Prolog predicates. The method can be applied to solve validation problems for communication protocols, and (potentially) of security and authentication protocols and abstractions of concurrent programs.

In this paper we overview the main features of our framework and comment on some of the more interesting applications.

1 Introduction
In this paper we describe a general framework for the automatic verification of concurrent systems parametric in the number of individual components. Our framework extends multiset rewriting over first order atomic formulas (MSR), a paradigm

1 Email: giorgio@disi.unige.it

©2002 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.
proposed in [11], with the notion of constraints peculiar of Constraint Logic Programming [22]. While multiset rewriting allows to specify concurrent processes in a natural way, constraints can be used to declaratively represent the relationship among their internal data. The combination of multisets and constraints is also important to define symbolic data structures to represent infinite sets of system configurations.

The relationship between multiset rewriting and concurrency can be well illustrated by reasoning in terms of Petri Nets. Petri Nets can be viewed in fact as multiset rewriting systems defined over a finite alphabet $\Sigma$. The symbols in $\Sigma$ represent the set of places of a Petri Net. A Petri Net marking can be modeled as a multiset of symbols from $\Sigma$: $k$ occurrences of the symbol $p$ in the multiset correspond to $k$ tokens in the place $p$. As a consequence, Petri Net transitions can be modeled as multiset rewriting rules over symbols in $\Sigma$. Finally, a sequence of rewriting steps can be viewed as the sequence of markings generating by an execution of the corresponding Petri Net.

Multiset rewriting over first-order atomic formulas (MSR) [11] naturally extends this connection to Petri Nets with colored tokens, the colors being first-order terms attached to atomic formulas representing tokens. It is important to note that in this setting rewriting is restricted to multisets of atomic formulas only. Intuitively, a multiset of atomic formulas is interpreted as a pool of active and concurrently executing processes. Atomic formulas denote the current state of individual processes.

In [14,8], we proposed a generalization of the MSR formalism in which multiset rewriting rules (over first-order atomic formulas) are annotated with constraints. The resulting language scheme, called MSR(\$\mathcal{C}\$), is parametric on the constraint system $\mathcal{C}$. MSR(\$\mathcal{C}\$) provides a clear separation between process structure and declarations of data relations. One of the advantages of this conceptual separation is a major modularity of the resulting specifications.

The logic obtained by combining multisets and constraints can be naturally used to reason about system properties. Specifically, several correctness properties can be verified by searching for error traces following the so-called backward reachability search scheme. The idea here is to first isolate the set of configurations that represent possible violations and then to generate all possible preconditions for those configurations to occur. If the initial states do not satisfy any of those preconditions than the violations will never occur, and thus the system is proved to be correct.

A particularly interesting class of safety properties are those whose violations can be expressed via upward-closed sets of configurations, i.e., that can be represented via minimal violations. Our symbolic representation of upward-closed sets of configurations is based on the notion of constrained configuration introduced in [14] to deal with asynchronous systems, i.e. multiset of first-order atomic formulas, annotated with constraints.

Multisets represent minimal requirements about the distribution of tokens in the net, whereas constraints provide a natural symbolic representation for relations over data of different processes. Based on this rich assertional language, we provide
symbolic operations needed to implement backward reachability, namely we define a symbolic predecessor operator, which is sound and complete for any constraint system.

The interest of this method is that allows to study properties that do not depend on the number of processes generated during a computation. We will clarify this point in this survey by discussing the application to a mutual exclusion protocol designed for client-server systems in which the number of clients is not bounded a priori.

We have implemented our automated verification method and applied it to analyze several practical case-studies. In general the method works without guarantees of termination. However, several techniques like static analysis and abstract interpretation can be applied to enforce termination on specific case studies. Furthermore, soundness ensures that error-traces will eventually be found: the prototype we used to implement the method performs a breadth-first symbolic backward exploration of the state space. Termination can be shown only for special subclasses of MSR(ε)-specifications [15].

As an implementation platform we have used a Constraint Logic Programming system, namely the clp(Q,R) library of Sicstus Prolog. CLP provides built-in operations and data structures to implement multiset-based operations (unification, efficient list representations and operations). Furthermore, it allows to handle constraints both as interpreted and uninterpreted objects. When queried, the clp(Q,R) solver provides operations like satisfiability, entailment, and projections that (via the encapsulation predicates peculiar of Sictus Prolog) can be embedded into procedure used for symbolic backward search.

In this paper we will give an overview of the general framework, the corresponding verification methodology. Finally, we will discuss issues about the CLP implementation and some of practical applications of the framework.

2 Constraint Multiset Rewriting

The framework called MSR is based on multiset rewriting systems defined over first-order atomic formulas and it has been introduced by Cervesato et al. [11] for the formal specification of cryptographic protocols. In [14], the basic formalism (i.e., without existential quantification) has been extended to allow for the specification of relations over data variable using constraints, i.e. a logic language interpreted over a fixed domain. Multiset rewriting rules allow one to locally model rendez-vous and internal actions of processes, and constraints to symbolically represent the relation between the data of different processes, thus achieving a clear separation between process structure and data paths. This formalism can be viewed as a first-order extension of Petri Nets in which tokens carry along structured data. We will call the resulting formalism MSR(ε). First of all, we introduce the notion of constraint system.
2.1 Constraint Systems

A constraint system is a tuple \( C = \langle V, L, D, \mathcal{R}, \sqsubseteq^C \rangle \) where:

(i) \( V \) is a denumerable set of variables;

(ii) \( L \) is a first-order language (the assertional language) defining a set of formulas with free variables in \( V \) (the constraints), closed with respect to variable renaming, existential quantification and conjunction, and allowing equalities between variables;

(iii) \( D \) is a possibly infinite set (the interpretation domain);

(iv) \( \mathcal{R}(\varphi) \) is a set of mappings \( V \to D \) (the set of solutions of a constraint \( \varphi \in L \)) that preserves the usual semantics of equalities, \( \land \) and \( \exists \) (intersection and projection of the solutions);

(v) \( \sqsubseteq^C \) is a relation such that \( \varphi \sqsubseteq^C \psi \) implies \( \mathcal{R}(\psi) \subseteq \mathcal{R}(\varphi) \) (the entailment relation: we say that \( \psi \) entails \( \varphi \)).

We assume that \( L \) contains constraints, denoted ‘true’, and ‘false’, which are identically true and identically false in \( D \).

By analogy with constraint programming, further requirements on constraint systems, like solution compactness [22], can be imposed. We refer to [22] for a discussion.

In the rest of the paper, we often denote the conjunction between constraints with a simple comma. We refer to a generic mapping \( \sigma : V \to D \) as an evaluation for the variables in \( V \). We use the notation \( \langle x_1 \mapsto d_1, x_2 \mapsto d_2, \ldots \rangle \) to denote an evaluation \( \sigma \) mapping \( x_1 \) to \( d_1 \), \( x_2 \) to \( d_2 \), and so on, and the notation \( \sigma|_x \) to denote the restriction of the evaluation \( \sigma \) to the variables \( x \). We also say that a constraint \( \varphi \) is satisfiable if \( \mathcal{R}(\varphi) \neq \emptyset \).

**Example 2.1** The class of linear integer constraints consists of formulas generated by the following grammar

\[
\varphi ::= \varphi \land \varphi \mid a_1 x_1 + \ldots + a_n x_n = a_{n+1} \mid a_1 x_1 + \ldots + a_n x_n > a_{n+1} \mid true \mid false
\]

where \( a_i \in \mathbb{Z} \) for \( i : 1, \ldots, n + 1 \). L.C.-constraints are interpreted over \( \mathbb{Z} \); \( \sqsubseteq^C \) is the usual entailment relation for linear integer constraints.

Note that there exists several methods for checking satisfiability, entailment, and for variable elimination (see e.g. [10]) over linear arithmetic constraints.

**Example 2.2** Let \( \varphi \) be \( x > y \land x > z \), then \( \sigma = \langle x \mapsto 2, y \mapsto 1, z \mapsto 0, \ldots \rangle \in \mathcal{R}(\varphi) \). Furthermore, \( \varphi \) is satisfiable, \( (x > y) \sqsubseteq^C \varphi \), and \( \exists y. \varphi \equiv (x > z) \).

We are now ready to define the framework of multiset rewriting with constraints.
In the following we will use the notion of multisets. We use \( \oplus \) and \( \ominus \) to denote \textit{multiset union} and \textit{multiset difference}, respectively.

Let \( \mathcal{P} \) be a finite set of predicate symbols, and \( \mathcal{V} \) a denumerable set of variables. An atomic formula over \( \mathcal{P} \) and \( \mathcal{V} \) has the form \( p(x_1, \ldots, x_n) \) (with \( n \geq 0 \)), where \( p \in \mathcal{P} \), and \( x_1, \ldots, x_n \) are distinct variables in \( \mathcal{V} \).

Now, let \( \mathcal{P} \) be a finite set of predicate symbols, and \( \mathcal{V} \) a denumerable set of variables. A \textit{multiset} of atomic formulas \( A_1, \ldots, A_k \) (with \( k \geq 1 \)) over \( \mathcal{P} \) and \( \mathcal{V} \), is indicated as \( A_1 | \ldots | A_k \), where \( A_i \) and \( A_j \) must have distinct variables for \( i \neq j \), and \( | \) is an associative and commutative constructor. The empty multiset is denoted by \( \varepsilon \).

In the rest of the paper will use \( \mathcal{M}, \mathcal{N}, \ldots \) to denote \textit{multisets} of atomic formulas.

Let \( \mathcal{P} \) be a finite set of predicate symbols, and \( \mathcal{C} = \langle \mathcal{V}, \mathcal{L}, \mathcal{P}, \mathcal{I}, \subseteq^c \rangle \) a constraint system. An MSR(\( \mathcal{C} \)) rule over \( \mathcal{P} \) and \( \mathcal{C} \) has the form \( \mathcal{M} \rightarrow \mathcal{M}' : \varphi \), where \( \mathcal{M} \) and \( \mathcal{M}' \) are two multisets of atomic formulas over \( \mathcal{P} \) and \( \mathcal{V} \), with distinct variables, and \( \varphi \in \mathcal{L} \).

Note that \( \mathcal{M} \rightarrow \varepsilon : \varphi \) and \( \varepsilon \rightarrow \mathcal{M} : \varphi \) are possible MSR(\( \mathcal{C} \)) rules.

In order to make the intuitive semantics of the previous specification precise, we introduce the ingredients for the operational reading of MSR rules. We first define the notion of \textit{ground} formulas. Let \( \mathcal{P} \) be a finite set of predicate symbols, and \( \mathcal{C} = \langle \mathcal{V}, \mathcal{L}, \mathcal{P}, \mathcal{I}, \subseteq^c \rangle \) a constraint system. A ground atomic formulas over \( \mathcal{P} \) and \( \mathcal{C} \) has the form \( p(d_1, \ldots, d_n) \) (with \( n \geq 0 \)), where \( p \in \mathcal{P} \) and \( d_1, \ldots, d_n \in \mathcal{D} \).

In the following, we will denote by \( \sigma(\mathcal{M}) \) the application of a solution \( \sigma : \mathcal{V} \rightarrow \mathcal{D} \) to a multiset of atomic formulas \( \mathcal{M} \). Formally, \( \sigma(A_1 | \ldots | A_k) = \sigma(A_1) | \ldots | \sigma(A_k) \), and \( \sigma(p(x_1, \ldots, x_m)) = p(d_1, \ldots, d_m) \) if \( \sigma(x_i) = d_i \) for \( x_i \in \mathcal{V} \) and \( v_i \in \mathcal{D}, i : 1, \ldots, m \).

Given an MSR(\( \mathcal{C} \)) rule \( R = \mathcal{M} \rightarrow \mathcal{M}' : \varphi \), over \( \mathcal{P} \) and \( \mathcal{C} \), the set of ground instances of \( R \), denoted \( \text{Inst}(R) \), is the set of ground multiset rewrite rules defined as follows:

\[
\text{Inst}(R) = \{ \sigma(\mathcal{M}) \rightarrow \sigma(\mathcal{M}') \mid \sigma \in \mathcal{P}(\varphi) \}
\]

A central notion for our semantics is that of \textit{current configuration} given below.

Let \( \mathcal{P} \) be a finite set of predicate symbols, and \( \mathcal{C} \) a constraint system. An MSR(\( \mathcal{C} \)) configuration is a multiset of ground atomic formulas over \( \mathcal{P} \) and \( \mathcal{C} \).

Configurations can be ordered with respect mutiset inclusion as follows. Given two multisets of atomic formulas \( \mathcal{M} \preccurlyeq \mathcal{N} \) if and only if \( \text{Occ}_A(\mathcal{M}) \leq \text{Occ}_A(\mathcal{N}) \) for every atomic formula \( A \), where \( \text{Occ}_A(\mathcal{M}) \) denotes the number of occurrences of \( A \) in \( \mathcal{M} \).

An MSR(\( \mathcal{C} \)) specification is defined as follows.

**Definition 2.3** [MSR(\( \mathcal{C} \)) Specifications] An MSR(\( \mathcal{C} \)) specification is a tuple \( \langle \mathcal{P}, \mathcal{C}, \mathcal{I}, \mathcal{R} \rangle \), where
is a finite set of predicate symbols,
\[ \mathcal{P} \]
is a constraint system,
\[ \mathcal{C} \]
is a set of configurations (the \textit{initial} configurations),
and \[ \mathcal{R} \] is a set of MSR(\mathcal{C}) rules over \mathcal{P} and \mathcal{C}.

The operational semantics of an MSR(\mathcal{C}) specification \langle \mathcal{P}, \mathcal{C}, \mathcal{I}, \mathcal{R} \rangle can be defined as follows. Let \( M \) be a configuration. A rule \( \mathcal{H} \rightarrow \mathcal{B} : \varphi \) from \( \mathcal{R} \) is \textit{enabled at} \( M \) \textit{via} the solution \( \sigma \in \mathcal{I}(\varphi) \) of the constraint \( \varphi \) if and only if \( \sigma(\mathcal{H}) \preceq M \).

Now, suppose \( R = \mathcal{H} \rightarrow \mathcal{B} : \varphi \) is enabled at \( M \) via \( \sigma \in \mathcal{I}(\varphi) \). Firing this rule at \( M \) yields the new configuration \( M' \), written \( M \Rightarrow_R M' \), if and only if \( M = \sigma(\mathcal{H}) \oplus Q \), and \( M' = \sigma(\mathcal{B}) \oplus Q \).

A \textit{run} is a sequence of configurations \( M_0 M_1 M_2 \ldots M_i \ldots \) with \( M_0 \in \mathcal{I} \) and for which there exist \( R_0 R_1 \ldots \in \mathcal{R} \) such that \( M_i \Rightarrow_{R_i} M_{i+1} \) for \( i \geq 0 \).

Let \( S \) a set of configurations. The successor operator \( \text{Post} \) is defined as
\[
\text{Post}(S) = \{ M' | M \Rightarrow_R M', M \in S, R \in \mathcal{R} \},
\]
whereas, the predecessor operator \( \text{Pre} \) is defined as
\[
\text{Pre}(S) = \{ M | M \Rightarrow_R M', M' \in S, R \in \mathcal{R} \}.
\]

The reflexive and transitive closures of the predecessor and successor operators are denoted \( \text{Pre}^* \) and \( \text{Post}^* \), respectively.

The reachability set is defined as \( \text{Post}^*(\mathcal{I}) \), \( \mathcal{I} \) being the initial states of the MSR(\mathcal{C}) specification under consideration.

2.3 \textit{An example: The Ticket Mutual Exclusion Protocol}

The ticket protocol is a \textit{mutual exclusion} protocol designed for multi-client systems operating on a shared memory. The protocol is based on a first-in first-served access policy. The algorithm is given in Fig.1 (where we use \( P | Q \) to denote the \textit{interleaving parallel execution} of \( P \) and \( Q \), and \( \langle \cdot \rangle \) to denote atomic fragments of code). The protocol works as follows. Initially, all clients are thinking, while \( t \) and \( s \) store the same initial value. When requesting the access to the critical section, a client stores the value of the current ticket \( t \) in its local variable \( a \). A new ticket is then emitted by incrementing \( t \). Clients wait for their turn until the value of their local variable \( a \) equals the value of \( s \).

After the elaboration inside the critical section, a process releases it and the current turn is updated by incrementing \( s \). During the execution, the \textit{global state} of the protocol consists of the current values of \( s \), \( t \), and of the local variables of \( n \) processes. As remarked in [10], even for \( n = 2 \) (only 2 clients), the values of the local variables of individual processes as well as \( s \) and \( t \) may get \textit{unbounded}. This implies that \textit{any instance} of the scheme of Fig. 1 gives rise to an infinite-state
Program

\[
\begin{align*}
\text{global var } s, t &: \text{ integer; } \\
\text{begin } \\
&\quad t := 0; \\
&\quad s := 0; \\
&\quad P_1 \mid \ldots \mid P_n; \\
\text{end.}
\end{align*}
\]

Fig. 1. The Ticket Protocol: \( n \) is a parameter of the protocol.

The algorithm is supposed to work for any value of \( n \), and it should also work if new clients enter the system at running time.

Multiset rewriting allows us to give an accurate and flexible encoding of the ticket protocol.

Let us first consider a \textit{single} shared resource controlled via the counters \( t \) and \( s \) as described in Section 2.3. The infinite collection of admissible initial states consists of all configurations with an \textit{arbitrary} but finite number of thinking processes and two counters having the same initial value \( (t = s) \). The specification is shown in Fig. 2. The initial configuration is the predicate \textit{init}, the seed of all possible runs of the protocol. The counters are represented here via the atoms \textit{count}(t) and \textit{turn}(s). Thinking clients are represented via the propositional symbol \textit{think}, and can be generated \textit{dynamically} via the second rule. The behavior of an individual client is described via the third block of rules of Fig. 2, in which the relation between the local variable and the global counters are represented via \textit{dc}-constraints. Finally, we allow thinking processes to terminate their execution as specified via the last rule of Fig. 2. The previous rules are independent of the current number of clients in the system. Note that in our specification we keep an explicit representation of the data variables; furthermore, we do not put any restrictions on their values. As a consequence, there are runs of our model in which \( s \) and \( t \) grow without any bound as in the original protocol. A sample run of a system with 2 clients (as in [10]) is shown in Fig. 3.

Let us consider now an \textit{open} system with an \textit{arbitrary} but \textit{finite} number of \textit{shared resources}, each one controlled by two local counters \( s \) and \( t \). We specify this scenario by associating a \textit{unique identifier} to each resource and to use it to stamp the corresponding pair of counters. Furthermore, we exploit non-determinism in order to simulate the capability of each client to choose which resource to use. The resulting specification is shown in Fig 4. We have considered an \textit{open} system in which new clients can be generated \textit{dynamically} via a \textit{demon} process. The process \textit{demon}(n) maintains a local counter \( n \) used to generate a new identifier, say \( id \), and to associate it to a newly created resource represented via the pair \textit{count}(id, t) and
Initial States

\[ \text{init} \quad \rightarrow \quad \text{count}(t) \mid \text{turn}(s) : t = s \]

Dynamic Generation

\[ \varepsilon \quad \rightarrow \quad \text{think} : \text{true} \]

Individual Behavior

\[ \text{think} \mid \text{count}(t) \quad \rightarrow \quad \text{wait}(a) \mid \text{count}(t') : a = t \land t' = t + 1 \]

\[ \text{wait}(a) \mid \text{turn}(s) \quad \rightarrow \quad \text{use} \mid \text{turn}(s') : a = s \land s' = s \]

\[ \text{use} \mid \text{turn}(s) \quad \rightarrow \quad \text{think} \mid \text{turn}(s') : s' = s + 1 \]

Termination

\[ \text{think} \quad \rightarrow \quad \varepsilon : \text{true} \]

Fig. 2. Ticket protocol for multi-client, single-server system, with an example of run.

\[ \text{init} \Rightarrow \ldots \Rightarrow \text{think} \mid \text{count}(8) \mid \text{turn}(8) \Rightarrow \text{think} \mid \text{think} \mid \text{count}(8) \mid \text{turn}(8) \]
\[ \Rightarrow \text{wait}(8) \mid \text{think} \mid \text{count}(9) \mid \text{turn}(8) \Rightarrow \text{wait}(8) \mid \text{wait}(9) \mid \text{count}(10) \mid \text{turn}(8) \]
\[ \Rightarrow \text{use} \mid \text{wait}(9) \mid \text{count}(10) \mid \text{turn}(8) \Rightarrow \text{use} \mid \text{wait}(9) \mid \text{count}(10) \mid \text{turn}(8) \mid \text{think} \]
\[ \Rightarrow \text{think} \mid \text{wait}(9) \mid \text{count}(10) \mid \text{turn}(9) \mid \text{think} \]

Fig. 3. Example of run.

turn(id, s). A thinking process non-deterministically chooses which resource to wait for by synchronizing with one of the counters in the system (the first rule of the third block in Fig. 4). After this choice, the algorithm behaves as usual w.r.t. to the chosen resource id. The termination rules can be specified as natural extensions of the single-server case. Note that in this specification the sources of infiniteness are the number of clients, the number of shared resources, the values of resource identifiers, and the values of tickets. An example of run is shown in Fig. 5.

3 An Assertional Language for MSR(\(\mathcal{C}\))

Since we can easily encode the formalism of two counters machines into MSR(\(\mathcal{C}\)) , reachability problems are in general undecidable for MSR(\(\mathcal{C}\)) specifications.

Even for fragments of MSR(\(\mathcal{C}\)) that are not Turing powerful, the reachability set of an MSR(\(\mathcal{C}\)) specification might be infinite and thus it might be extremely difficult to explore it. The source of infiniteness may be either the increasing size (number of elements) of generated configurations, or the unboundedness of the values attached to atomic formulas (e.g. think about a simple counter incremented at every rule application), or both.
Initial States

\[ \text{init} \rightarrow \text{demon}(n) : true \]

Dynamic Process and Server Generation

\[ \epsilon \rightarrow \text{think} : true \]

\[ \text{demon}(n) \rightarrow \text{demon}(n') \mid \text{count}(id,t) \mid \text{turn}(id',s) : \]

\[ n' = n + 1 \land t = s \land id = n \land id' = id \]

Individual Behavior

\[ \text{think} \mid \text{count}(id,t) \rightarrow \text{think}(r) \mid \text{count}(id',t') : r = id \land id' = id \land t' = t \]

\[ \text{think}(r) \mid \text{count}(id,t) \rightarrow \text{wait}(r',a) \mid \text{count}(id',t') : \]

\[ r = id' \land a = t \land t' = t + 1 \land r' = r \land id' = id \]

\[ \text{wait}(r,a) \mid \text{turn}(id,s) \rightarrow \text{use}(r',a') \mid \text{turn}(id',s') : \]

\[ r = id \land a = s \land a' = a \land s' = s \land r' = r \land id' = id \]

\[ \text{use}(r,a) \mid \text{turn}(id,s) \rightarrow \text{think} \mid \text{turn}(id',s') : r = id \land s' = s + 1 \land id' = id \]

Termination

\[ \text{think}(r) \rightarrow \epsilon : true \]

\[ \text{think} \rightarrow \epsilon : true \]

Fig. 4. Ticket protocol for multi-server, multi-client systems.

\[ \text{init} \Rightarrow \text{demon}(3) \Rightarrow \text{count}(3,0) \mid \text{turn}(3,0) \mid \text{demon}(4) \Rightarrow \ldots \]

\[ \ldots \Rightarrow \text{count}(3,0) \mid \text{turn}(3,0) \mid \text{think} \mid \text{think} \mid \text{demon}(4) \]

\[ \Rightarrow \text{count}(3,0) \mid \text{turn}(3,0) \mid \text{count}(4,8) \mid \text{turn}(4,8) \mid \text{think} \mid \text{think} \mid \text{demon}(5) \Rightarrow \ldots \]

\[ \Rightarrow \text{count}(3,0) \mid \text{turn}(3,0) \mid \text{count}(4,8) \mid \text{turn}(4,8) \mid \text{think}(4) \mid \text{think}(3) \mid \text{demon}(5) \]

\[ \Rightarrow \text{count}(3,0) \mid \text{turn}(3,0) \mid \text{count}(4,9) \mid \text{turn}(4,8) \mid \text{wait}(4,8) \mid \text{think}(3) \mid \text{demon}(5). \]

Fig. 5. Example of run.

The only possibility of automatically analyzing the behavior of this infinite-state specification consists of finding adequate finite, symbolic representations of infinite collections of configurations. In order to achieve this goal, in this section we will introduce an assertional language based on the notion of upward-closed sets of configurations.

To explain this idea, let us consider again the Example 1. Suppose we are interested in proving that the specification of the example discussed in the previous sections satisfies the following invariant: all reachable states satisfy the mutual exclusion property only one process per time is in state use. Instead of proving it directly, we can try to disprove it as follows: we first select all possible configurations that violate it, and then show that they are not reachable from the initial state \text{init}. The
set of interesting violations $U$ consists of configurations like $use(v) \parallel use(w) \oplus U'$ where $U'$ is any possible multiset. Then, we note that $U$ can be generated from the (infinite) set of configurations $M = \{use(v) \mid use(w) \models v \geq 0, w \geq 0\}$ by taking the upward closure with respect to multiset inclusion: if a configuration contains any ground instance of $M$, then it violates itself the invariant.

As discussed in [1], the negation of an invariant property often enjoys the property of being upward closed with respect to an inclusion ordering defined over sets of configurations. As a first approximation of the problem of exploring an infinite state space, we could then start the exploration from the unsafe states, using minimal violations to represent them, and reason backward by applying the predecessor operator $Pre$ (i.e., computing weakest pre-conditions). Let us formally define the notion of upward closed sets.

Let $\langle \mathcal{P}, \mathcal{C}, \mathcal{I}, \mathcal{R} \rangle$ be an MSR($\mathcal{C}$) specification, and $S$ a set of configurations. The upward closure of $S$, denoted $Up(S)$, is defined as

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

We say that $S$ is upward closed if $S = Up(S)$.

Upward-closed sets of configurations have interesting properties with respect to the predecessor operator $Pre$.

**Proposition 3.1 ([15])** $Up(Pre(S)) \subseteq Pre(Up(S))$ for any set $S$ of configurations.

In general the reverse implication does not hold.

**Example 3.2** Consider $p \rightarrow q_1 \parallel q_2$ 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 3.3 ([15])** 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. The previous properties do not suffice to finitely represent and manipulate sets upward closed sets of configurations. In fact, we still have to solve the problem of finitely represent what we called the minimal violations. This is not always possible. However, in the example of the single server ticket algorithm they have a regular structure that we can exploit by translating the extensional description

$$M = \{use(v) \mid use(w) \models v \geq 0, w \geq 0\}$$

into the following intensional description, called a constrained configuration

$$use(x) \mid use(y) : true$$

where $x, y$ are free variables. In other words, by annotating a non ground configuration (i.e., in which formulas have free variables) with a constraint we can implicitly
represent all unsafe configurations we are interested in for our example. They will correspond to the upward closure of the set of ground instances of the constrained configuration above. It is important to note that this techniques is just an heuristics through which we try to embed as many configurations as possible inside a regular structure. In the following section we will formalize these ideas and show how to use them in combination with the backward reachability approach proposed in [1].

3.1 Symbolic Representation via Constrained Configurations

In order to finitely represent the generators of an upward closed set of configurations, we use a rich assertional language based on the notion of constrained configurations, i.e. multisets of (non ground) atomic formulas annotated with constraints.

Let \( \langle \mathcal{P}, \mathcal{C}, \mathcal{I}, \mathcal{R} \rangle \) be an MSR(\( \mathcal{C} \)) specification, with \( \mathcal{C} = \langle \mathcal{V}, \mathcal{L}, \mathcal{P}, \mathcal{I}, \subseteq \rangle \). A constrained configuration is a multiset of atomic formulas over \( \mathcal{P} \) and \( \mathcal{C} \), annotated with a constraint, having the form

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

where \( p_1, \ldots, p_n \in \mathcal{P}, \varphi \in \mathcal{L} \) is a satisfiable constraint, and \( x_{11}, \ldots, x_{nk_n} \) are distinct variables in \( \mathcal{V} \).

The set of ground instances of a constrained configuration can be defined as follows.

Let \( \langle \mathcal{P}, \mathcal{C}, \mathcal{I}, \mathcal{R} \rangle \) be an MSR(\( \mathcal{C} \)) specification, and \( M : \varphi \) a constrained configuration. The set of ground instances of \( M : \varphi \) is defined as

\[
\text{Inst}(M : \varphi) = \{ \sigma(M) \mid \sigma \in \mathcal{I}(\varphi) \}.
\]

As an example, let \( C \) be the constrained configuration \( \text{use}(x) \mid \text{user}(y) : \text{true} \). Then, if \( \text{LC} \)-constraints are interpreted over non-negative integers \( \text{Inst}(C) \) contains configurations like \( \text{use}(1) \mid \text{use}(2), \text{use}(4) \mid \text{use}(6) \), etc.

The previous definition can be extended to sets of constrained configurations (denoted \( S, S', \ldots \)) in the natural way:

\[
\text{Inst}(S) = \bigcup_{C \in S} \text{Inst}(C)
\]

Following the intuition we explained in the previous section, instead of taking the set of instances as flat denotation of a set of constrained configurations \( S \), we choose the following rich denotation.

Let \( \langle \mathcal{P}, \mathcal{C}, \mathcal{I}, \mathcal{R} \rangle \) be an MSR(\( \mathcal{C} \)) specification, and \( S \) a set of constrained configurations (with distinct variables from each other). The rich denotation of \( S \), denoted \( [S] \), is given by the upward closure of the set of its ground instances, i.e.

\[
[S] = \text{Up}(\text{Inst}(S))
\]
Example 3.4 Let $C$ be defined as $use(x) \mid user(y) : true$. Then, $\llbracket C \rrbracket$ contains configurations like $use(1) \mid use(2)$ as well as $use(1) \mid use(2) \mid use(0)$, etc.

4 A Symbolic Verification Procedure

The next step toward an effective verification procedure consists of defining a symbolic predecessor operator working on the assertional language according to the rich denotation described in the previous section. Let us first introduce the notion of unification between two multisets of atomic formulas (with distinct variables).

Firstly, given two atomic formulas $A = p(x_1, \ldots, x_k)$ and $B = q(y_1, \ldots, y_l)$, we use $A = B$ as a shorthand for the constraint $x_1 = y_1 \land \ldots \land x_k = y_k$, provided $p = q$ and $k = l$. Unification is defined then as follows.

Definition 4.1 Let $M = A_1 \mid \ldots \mid A_m : \phi$, and $N = B_1 \mid \ldots \mid B_m : \psi$ be two constrained configurations. We say that $\theta \in \mathcal{L}$ is a unifier for $M$ and $N$, written $M =_\theta N$, provided $m = n$, and the constraint

$$\theta = \phi \land \psi \land \bigwedge_{i=1}^{n} A_i = B_j,$$

is satisfiable, $j_1, \ldots, j_n$ being a permutation of $1, \ldots, n$.

Below, given $E$ (e.g., a rule, constraint, or constrained configuration) we call $E'$ a variant of $E$ if $E'$ is obtained applying a renaming $\iota$ to $E$, i.e. $E' = \iota(E)$ (a renaming is a mapping from variables of $E$ to fresh variables).

Definition 4.2 [The Pre Operator] The operator $\text{Pre}$ is defined on a set $S$ containing constrained multisets (with disjoint variables) as follows

$$\text{Pre}(S) = \{ (A \oplus N) : \exists x_1, \ldots, x_k. \theta) \mid (A \longrightarrow B : \psi) \in \mathcal{R}, (M : \phi) \in S, \text{ and } x_1, \ldots, x_k \text{ are all the variables that do not occur in } A \oplus N'. \}.$$

Let us give an example of application.

Example 4.3 Consider the constrained configuration

$$p(x, y) \longrightarrow q(x, y) : x \geq 0, y = 1$$
Given the singleton \( S \) with \( q(u, w) : u = 1, w \geq 0 \), \( \text{Pre}(S) \) 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 \( q(4, 1) \mid q(1, 5) \in \llbracket S \rrbracket \).
The latter constrained configuration can be obtained by setting \( M' = B' = \varepsilon \) (the empty multiset) when applying \( \text{Pre} \) to \( S \).

The new operator enjoys the following property.

**Theorem 4.4 ([15])** Let \( \langle P, C, I, R \rangle \) be an MSR(\( C \)) specification and \( S \) a set of constrained configurations (with distinct variables from each other). Then \( \llbracket \text{Pre}(S) \rrbracket = \text{Pre}(\llbracket S \rrbracket) \).

In order to define a symbolic reachability algorithm, we still need a comparison operator between constrained configurations. We discuss next the general requirements it should satisfy.

**Definition 4.5** Let \( \langle P, C, I, R \rangle \) be an MSR(\( C \)) specification. We call a relation \( \sqsubseteq^m \) between constrained configurations (\( M, N, \ldots \)) **entailment relation** whenever \( M \sqsubseteq^m N \) implies \( \llbracket N \rrbracket \subseteq \llbracket M \rrbracket \).

Based on this definition, we can define a **symbolic backward reachability** procedure that we can use to check safety properties whose negation can be expressed via an upward closed set of configurations. We can rephrase the backward reachability algorithm (which is parametric on the constraint system \( C \) and entailment relation \( \sqsubseteq^m \)) in our setting, as follows.

**Definition 4.6** Given an MSR(\( C \)) specification \( \langle P, C, I, R \rangle \), an entailment relation \( \sqsubseteq^m \) for constrained configurations, and a set \( U \) of constrained configurations (with distinct variables from each other) representing a set of unsafe states, the symbolic backward reachability procedure consists of the following two steps:

(i) we first compute \( \text{Pre}^*(U) \): starting from \( U \), we repeatedly apply \( \text{Pre} \) to all stored constrained configurations. We stop when it is not possible to store new constrained configurations (i.e., for each new constrained configuration \( M \) we already computed \( N \) such that \( N \sqsubseteq^m M \));

(ii) if the fixpoint computation terminates we check that the initial configurations \( I \), representing the initial states of the system, are not contained in the denotation of the resulting set of constrained configurations (i.e., \( I \cap \llbracket \text{Pre}^*(U) \rrbracket = 0 \)).

The pseudo-code of the algorithm is given in Fig. 6.

In general, the algorithm is not terminating, since it is possible to encode undecidable reachability problems (e.g. for two counter machines) as verification problems of generic MSR(\( C \)) specifications. However, following [1, 18], if we can
Procedure $\text{Pre}^*(U:\text{set of constrained configurations})$

$S := U;$
$R := \emptyset;$

while $S \neq \emptyset$ do
begin
    remove $(M : \varphi)$ from $S$;
    if there are no $(N : \psi) \in R$ s.t. $(M : \varphi)$ entails $(N : \psi)$ then
        add $(M : \varphi)$ to $R$
        $S := S \cup \text{Pre}((M : \varphi));$
    endif
end
endwhile
end.

Fig. 6. Symbolic Backward Reachability

prove that $\sqsubseteq^m$ is a well-quasi ordering, then the previous procedure turns out to be a complete algorithm to compute $\text{Pre}^*([[U]])$. In order to check that a safety properties holds we also need that the emptiness test for the intersection with the initial states is decidable.

5 Implementation in CLP

Most CLP systems can be naturally used as implementation platform for the verification methodology discussed in this paper. CLP systems like the clp(Q,R) library of Sicstus Prolog provides both symbolic data structures to represent multiset-based specification languages and encapsulation operators for handling constraints at the object level. These features allowed us to implement a prototype in which MSR($\varepsilon'$) specifications are represented via unit clauses that combine terms with variables and constraints like:

$$r([p(X), q(Y)], [r(Z), t(W)], \{X > Y, Z = Y, W = Y + 1\}).$$

Here a multiset is represented as a list of atomic formulas like $[p(X), q(Y)]$ and a constraints is represented (following the clp(Q,R) syntax) as a list of linear arithmetic constraints like $\{X > Y, Z = Y, W = Y + 1\}$.

In this setting the symbolic predecessor and the entailment operator can be naturally implemented by using unification (to handle multiset matching) and querying the constraint solver (for checking satisfiability, for projection, and for entailment).
This way, we must be able to transform constraints from uninterpreted terms to active objects. Furthermore, we must be able to retrieve the results produced by the constraint solver (that can be viewed as an oracle that puts its output on the constraint store). The clp(Q,R) library provides special built-in predicates for this sort of computations. Specifically, it provides a predicate that given a constraint in input return its simplified form (if satisfiable). Furthermore, it provides predicates that can be used to define (in the similar input output fashion) projection over individual variables.

Using these features we have defined a library for handling (sets of) constrained configurations and we have incorporate it within a least fixpoint engine with entailment as termination test. The prototype has been used in several applications as discussed in the following section.

6 Applications

As a first application we studied mutual exclusion properties for different formulations of the ticket protocol. As shown in [10], this protocol has an infinite-state space even for system configurations with only 2 processes. We have considered both a multi-client, single-server formulation, i.e. with an arbitrary but finite number of dynamically generated clients but a single shared resource, and a multi-client, multi-server system in which both clients and servers are created dynamically. Both examples have been modeled using multiset rewriting rules with linear constraints, that support arithmetic operations like increment and decrement of data variables.

Our models are faithful to the original formulation of the algorithm, in that we do not abstract away global and local integer variables attached to individual clients that in fact can still grow unboundedly. Using our symbolic backward reachability procedure combined with the dynamic use of abstractions, we have automatically verified that both models are safe for any number of clients and servers and for any values of local and global variables [8].

The second application of our method was the analysis of a coherence protocol for virtual shared memory proposed by Li and Hudak in [24] and previously analyzed in [20]. Using our technique, we found an inconsistency in one of the Colored Petri Nets model proposed in [20]. After having corrected the error, we have automatically verified a new model of the protocol in which the number of threads, processors, and pages of virtual memory are unbounded parameters [8].

Finally, we have automatically validate coherence protocols for multiprocessors systems (M.S.I., M.E.S.I., Synapse) in which number of processors, cache lines and memory locations are left as unbounded parameters [15].

We are currently working on the application of this methodology to the validation of security and authentication protocols and of abstraction of concurrent programs.
7 Related Works

Our work is inspired to the approach of [2,4], where existential regions are proposed as symbolic representation of configurations for parameterized Timed Petri Nets. In our framework we consider however problems and constraint systems that do not depend on the notion of time.

Networks of finite-state processes can be analyzed using the automata theoretic approach of [7,12,23], where sets of global states are represented as regular languages, and transitions as relations on languages. Symbolic exploration can then be performed using operations over automata with ad hoc accelerations or with automated abstractions techniques.

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 previous features also distinguish our approach from the verification with invisible invariants method of [5]. Contrary, we follow here the paradigm of symbolic model checking with rich assertional languages [23], trying to isolate decidable classes for which backward reachability terminates. The two approaches can be used to attack similar problems using different point-of-views.

Our ideas are related to previous works connecting Constraint Logic Programming and verification, see e.g. [16]. In this setting transition systems are encoded via CLP programs used to encode the global state of a system and its updates. We refine this idea by using multiset rewriting and constraints to locally specify updates to the global state. The notion of constrained multiset extends that of constrained atom of [16]. The locality of rules allows us to consider rich denotations (upward-closures) instead of flat ones (instances) like, e.g., in [16]. This way, we can lift the approach to the parameterized case.

Finally, the use of constraints, backward reachability, structural invariants and better quasi orderings seem all ingredients that distinguish our hybrid method from classical approaches based on multiset and AC rewriting techniques (see e.g. [11,26]).

References


[16] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, 
Proceedings 5th International Conference on Tools and Algorithms for Construction 
and Analysis of Systems (TACAS’99), volume 1579 of LNCS, pages 223–239, 

Proceedings 14th International Symposium on Logic in Computer Science (LICS’99), 


[20] K. Fisler, C. Girault. Modelling and Model Checking a Distributed Shared Memory 

editor, Proceedings 9th International Workshop on Logic Program Synthesis and 
Transformation (LOPSTR’99), volume 1817 of LNCS, pages 30–41, Venezia, Italy, 

Symposium on Principles of Programming Languages (POPL’87), pages 111–119, 

with rich assertional languages. In O. Grumberg, editor, Proceedings 9th International 
Conference on Computer Aided Verification (CAV’97), volume 1254 of LNCS, pages 


Technology, Uppsala University, 2000.

[26] M. Rusinowitch and L. Vigneron. Automated Deduction with Associative and 
Commutative Operators. Applicable Algebra in Engineering, Communication and 