Verify algorithms' properties with the MSR($\mathcal{C}$) prototype

Alessandro Bulzacchi
Computational Models and Languages

Year 2011-2012

Contents

1 Introduction

MSR($\mathcal{C}$)[1][2] is a tool to model-check infinite state systems, extending Petri Nets ideas. The goal of this project is to verify properties of given algorithms for self-organising networks.

MSR($\mathcal{C}$) is a tool for the automated verification of infinite state systems. In many practical cases verification problems can be reduced to problems related to Petri Nets. Reachability procedures can be used to verify the original property on the resulting Petri Net. To achieve this goal we will work in a formal setting based on multiset rewriting over first order atomic formulas (MSR). We will illustrate this idea with some examples.

In section 2 we will introduce the formalism MSR($\mathcal{C}$) that we use as specification language, assertional language (used to reason on properties of the specification) and we define our verification procedure to handle the extended specifications. In section 3 we study some practical case-studies. In section 4 we address some conclusions and discuss future work.

In this paper we are going to revise and reuse the work presented in [1, 2].

The code in this article is written in Sicstus Prolog v.3.1. Sicstus Prolog is available in the Engineering faculty's labs in Cesena. If interested contact the faculty's personnel.

1.1 Constraint Multiset Rewriting

Petri Nets can be viewed in fact as *multiset rewriting systems* defined over a finite alphabet $\mathcal{P}$. The symbols in $\mathcal{P}$ represent the set of places of a Petri Net. A Petri Net marking can be modeled as a multiset of symbols from $\mathcal{P}$: **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 $\mathcal{P}$. For instance, given $\mathcal{P}$ = {null,use,idle,busy}, we can view a multiset M = null | null | idle ("|" being the multiset constructor) as a Petri Net marking with two tokens in place null and one in place idle. The multiset rewriting rule

$$\boldsymbol{think} \mid \boldsymbol{idle} \longrightarrow \boldsymbol{use} \mid \boldsymbol{busy}$$

can be viewed as Petri Net transition that removes a token from think, a token from idle, and puts one token in use and one token in busy. 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) naturally extends this connection to Petri Nets with colored tokens, the colors being first-order terms attached to atomic formulas representing tokens.

If we annotate rules with constraints, we achieve a clear separation between process structure and declarations of data relations. As an example, in a rule

like

$$\textbf{\textit{think}} \mid \textbf{\textit{count(x) wait(y)}} \mid \textbf{\textit{count(}}x'\textbf{\textit{)}} : x = y \wedge x' = x + 1$$

the variable x, y, and x' are universally quantified, and x = y $\wedge, \ldots$ is a constraint defined over them. It is important to note that in this setting rewriting is restricted to multisets of atomic formulas only. Intuitively, a multisets of atomic formulas is interpreted as a pool of active and concurrently executing processes. Atomic formulas denote the current state of individual processes. 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. Specially, several correctness properties can be verified by searching for error traces following the so-called backward reachability search scheme.

1.2 Verification Procedures

The method we use to check parameterized safety properties is symbolic backward reachability, focused on safety properties whose violations can be expressed via upward-closed sets of configurations. Our symbolic representation of upward-closed sets of configurations is based on the notion of constrained configurations, that is multisets 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 assertional language, we provide symbolic operation needed to implement backward reachability, namely we give a definition of a symbolic predecessor operator for, which is sound and complete for any constraint system.

1.3 Case-studies

We will use a software called MSR($\mathcal{C}$) to analyze several practical case-studies. MSR($\mathcal{C}$) is a CLP prototype for Symbolic Backward Analysis written in Sicstus Prolog with the support of clp(Q,R) library. 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. As a first example, we studied mutual exclusion properties for different formulation of the Ticket Protocol. This protocol has an infinite-state space. We will use this example to describe in more detail the MSR($\mathcal{C}$) specification. As second example, we report the result of the analysis of a given algorithms for self-organising networks.

2 Constraint Multiset Rewriting

The framework called MSR[1] is based on multiset rewriting systems defined over first-order atomic formulas and it has been introduced by Cervesato for the formal specification of cryptographic protocols. 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($\mathcal{C}$). First of all, we introduce the notion of constraint system.

2.1 Constraint Systems

A constraint system is a tuple $\mathcal{C} = \langle \mathcal{V}, \mathcal{L}, \mathcal{D}, \mathcal{I}, \sqsubseteq^c \rangle$ where:

1. $\mathcal{V}$ is a denumerable set of variables;
2. $\mathcal{L}$ is a first-order language (the assertional language) defining a set of formulas with free variables in $\mathcal{V}$(the constraints), closed with respect to variable renaming, existential quantification and conjunction, and allowing equalities between variables;
3. $\mathcal{D}$ is a possibly infinite set (the interpretation domain);
4. $\mathcal{I}(\varphi)$ is a set of mappings $\mathcal{V} \to \mathcal{D}$ (the set of solutions of a constraint $\varphi \in \mathcal{L}$) that preserves the usual semantics of equalities, $\wedge$ and $\exists$ (intersection and projection of the solutions);
5. $\sqsubseteq^c$ is a relation such that $\varphi \sqsubseteq^c \psi$ implies $\mathcal{I}(\psi) \subseteq \mathcal{I}(\varphi)$ (the entailment relation: we say that ψ entails φ).

We assume that $\mathcal{L}$ contains constraints, denoted **'true'**, and **'false'**, which are identically true and identically false in $\mathcal{D}$. By analogy with constraint programming, further requirements on constraint systems, like solution compactness, can be imposed. We often denote the conjunction between constraints with a simple comma. We refer to a generic mapping $\sigma : \mathcal{V} \to \mathcal{D}$ as an evaluation for the variables in $\mathcal{V}$. We use the notation $\langle x_1 \mapsto d_1, x_2 \mapsto d_2 \rangle$ to denote an evaluation σ mapping x_1 to d_1, and so on, and the notation $\sigma_{|x}$ to denote the restriction of the evaluation σ to the variables x. We also say that a constraint φ is satisfiable if $\mathcal{I}(\varphi) \neq 0$.

2.2 The MSR($\mathcal{C}$) language

In the following we will use the notion of multisets. We use $\oplus$ and $\ominus$ to denote multiset union and 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 $\mathcal{P}$ 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 multiset of atomic formulas $p(A_1, \ldots, A_k)$ (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 ϵ.

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

Let $\mathcal{P}$ be a finite set of predicate symbols, and $\mathcal{C} = \langle \mathcal{V}, \mathcal{L}, \mathcal{D}, \mathcal{I}, \sqsubseteq^c \rangle$ a constraint system. An MSR($\mathcal{C}$) rule over $\mathcal{P}$ and $\mathcal{C}$ has the form $\mathcal{M} \longrightarrow \mathcal{M}' : \varphi$, 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 $\mathcal{M} \longrightarrow \epsilon : \varphi$ and $\epsilon \longrightarrow \mathcal{M} : \varphi$ 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 ground formulas.

Let $\mathcal{P}$ be a finite set of predicate symbols, and $\mathcal{C} = \langle \mathcal{V}, \mathcal{L}, \mathcal{D}, \mathcal{I}, \sqsubseteq^c \rangle$ 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 $d_1, \ldots, d_n$ are distinct variables in $\mathcal{D}$.

In the following, we will denote by $\sigma(\mathcal{M})$ the application of a solution $\sigma : \mathcal{V} \to \mathcal{D}$ 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 $v_i \in \mathcal{D}, i : 1, \ldots, m$.
Based on this definition, we introduce the notion of rule instance.
Given an MSR($\mathcal{C}$) rule $R = \mathcal{M} \longrightarrow \mathcal{M}' : \varphi$, over $\mathcal{P}$ and $\mathcal{C}$, the set of ground instances of R, denoted Inst(R), is the set of ground multiset rewrite rules defined as follows:

$$Inst(R) = \{\sigma(\mathcal{M}) \longrightarrow \sigma(\mathcal{M}') | \sigma \in \mathcal{I}(\varphi)\}$$

A central notion for our semantics is that of 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 multiset inclusion as follows.

Given two multisets of atomic formulas $\mathcal{M} \preceq \mathcal{N}$ if and only if $Occ_A(\mathcal{M}) \leq Occ_A(\mathcal{N})$ for every atomic formula $\mathcal{A}$, where $Occ_A(\mathcal{M})$ denotes the number of occurrences of $\mathcal{A}$ in $\mathcal{M}$.

An MSR($\mathcal{C}$) specification is a tuple $\langle \mathcal{P}, \mathcal{C}, \mathcal{I}, \mathcal{R} \rangle$ where

- $\mathcal{P}$ is a finite set of predicate symbols,
- $\mathcal{C}$ is a constraint system,
- $\mathcal{I}$ is a set of configurations (the initial configurations),
- $\mathcal{R}$ 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 $\mathcal{M}$ be a configuration. A rule $\mathcal{H} \longrightarrow \mathcal{B} : \varphi$ from $\mathcal{R}$ is enabled at $\mathcal{M}$ via the solution of the constraint φ if and only if $\sigma(\mathcal{H}) \preceq \mathcal{M}$.

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

A run is a sequence of configurations $\mathcal{M}_0\mathcal{M}_1\mathcal{M}_2 \ldots \mathcal{M}_i \ldots$ with $\mathcal{M}_0 \in \mathcal{I}$ and for which there exist $R_0R_1 \ldots \in \mathcal{R}$ such that $\mathcal{M}_i \Rightarrow_{Ri} \mathcal{M}_{i+1}$ for $i \geq 1$.

Let $\mathcal{S}$ a set of configurations. The successor operator *Post* is defined as

$$Post = \{\mathcal{M}' | \mathcal{M} \Rightarrow_R \mathcal{M}', \mathcal{M} \in \mathcal{S}, R \in \mathcal{R}\},$$

whereas, the predecessor operator *Pre* is defined as

$$Pre = \{\mathcal{M} | \mathcal{M} \Rightarrow_R \mathcal{M}', \mathcal{M}' \in \mathcal{S}, R \in \mathcal{R}\}.$$

The reflexive and transitive closures of the predecessor and successor operators are denoted *Pre** and *Post**, respectively.

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

2.3 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 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 (think about a simple counter incremented at every rule application), or both.

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.

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.

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 $\mathcal{S}$ a set of configurations. The upward closure of $\mathcal{S}$, denoted *Up(S)* , is defined as

$$Up(S) = \{\mathcal{N} | \mathcal{M} \preceq \mathcal{N}, \mathcal{M} \in S\}$$

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*.

$$Up(Pre(S)) \subseteq Pre(Up(S)) \text{ for any set } \mathcal{S} \text{ of configurations.}$$

In general the reverse implication does not hold; however, the following property holds.

$$\text{If } \mathcal{S} \text{ is } \textit{upward-closed}, \text{ then } Up(Pre(S)) = Pre(Up(S))$$

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.

2.4 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 $\mathcal{C} = \langle \mathcal{V}, \mathcal{L}, \mathcal{D}, \mathcal{I}, \sqsubseteq^c \rangle$. A constrained configuration is a multiset of atomic formulas over $\mathcal{P}$ and $\mathcal{C}$, annotated with a constraint, having the form

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

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 $\mathcal{M} : \varphi$ a constrained configuration. The set of ground instances of $\mathcal{M} : \varphi$ is defined as

$$\mathit{Inst}(\mathcal{M} : \varphi) = \{\sigma(\mathcal{M}) | \sigma \in \mathcal{I}(\varphi)\}.$$

Following the intuition we explained in the previous section, instead of taking the set of instances as flatdenotation of a set of constrained configurations $\mathbf{S}$, we choose the following rich denotation.

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

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

2.5 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 $x_1 = y_1 \wedge \ldots \wedge x_k = y_k$, provided $p = q$ and $k = l$. Unification is defined then as follows.

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

$$\theta = \varphi \wedge \psi \wedge \textstyle\bigwedge_{i=1}^{n} \wedge A_i = B_{j_i}$$

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

Below, given E (a rule, constraint, or constrained configuration) we call E' a variant of E if E' is obtained applying a renaming $\imath$ to E, in other words $E' = \imath(E)$ (a renaming is a mapping from variables of to fresh variables).

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

$$\begin{gathered}\mathbf{Pre}(\mathbf{S}) = \{\mathcal{A} \oplus \mathcal{N} : \exists x_1 \ldots x_k.\theta | (\mathcal{A} \longrightarrow \mathcal{B} : \psi) \in \mathcal{R}, (\mathcal{M} : \varphi) \in \mathbf{S}, \\ \mathcal{M}' \preceq \mathcal{M}, \mathcal{B}' \preceq \mathcal{B}, (\mathcal{M}' : \varphi) =_\theta (\mathcal{B}' : \psi), \\ \mathcal{N} = \mathcal{M} \ominus \mathcal{M}',\end{gathered}$$

and $x_1 \ldots x_k$ are all variables not in $\mathcal{A} \oplus \mathcal{N}$

The new operator enjoys the following property.

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). Then $[\![\mathbf{Pre}(\mathbf{S})]\!] = Pre([\![\mathbf{S}]\!])$.

In order to define a symbolic reachability algorithm, we still need a comparison operator between constrained configurations.

Let $\langle \mathcal{P}, \mathcal{C}, \mathcal{I}, \mathcal{R} \rangle$ be an MSR($\mathcal{C}$) specification. We call a relation $\sqsubseteq^m$ between constrained configurations $(\mathcal{M}, \mathcal{N}, \ldots)$ entailment relation whenever $\mathcal{M} \sqsubseteq^m \mathcal{N}$ implies $[\![\mathcal{N}]\!] \subseteq [\![\mathcal{M}]\!]$.

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 $\mathcal{C}$ and entailment relation $\sqsubseteq^m$) in our setting, as follows.

Given an MSR($\mathcal{C}$) specification $\langle \mathcal{P}, \mathcal{C}, \mathcal{I}, \mathcal{R} \rangle$, an entailment relation $\sqsubseteq^m$ for constrained configurations, and a set $\mathbf{U}$ 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:

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

2. if the fixpoint computation terminates we check that the initial configurations $\mathcal{I}$ representing the initial state of the system, are not contained in the denotation of the resulting set of constrained configurations ($\mathcal{I} \cap [\![\mathbf{Pre^*(U)}]\!] = 0$).

The pseudo-code of the algorithm is:

```
Procedure Pre*(U : set of constrained configurations)
 S := U;
 R := ∅;
 while S ≠ ∅ do
   begin
    remove (M : φ) from S;
    if there are no (N' : ψ) ∈ R s.t. (M : φ) entails (N' : ψ) then
         add (M : φ) to R
         S := S ∪ Pre({M : φ});
    endif
  end
 endwhile
end.
```

In general, the algorithm is not terminating, since it is possible to encode undecidable reachability problems (for example for two counter machines) as verification problems of generic MSR($\mathcal{C}$) specifications. However, if we can prove that $\sqsubseteq^m$ is a well-quasi ordering, then the previous procedure turns out to be a complete algorithm to compute $[\![Pre*(U)]\!]$. 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.

2.6 Implementation in CLP

Most CLP systems can be naturally used as implementation platform for the verification methodology discussed previously. 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 use a prototype in which MSR($\mathcal{C}$) 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 atomic 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.

3 Case-studies

As first application we studied mutual exclusion properties for the ticket protocol. This protocol has an infinite-state space even for system configurations with only two processes. We have considered a single-server formulation with an arbitrary but finite number of dynamically generated clients but a single shared resource. This example has been modeled using multiset rewriting rules with linear constraints that support arithmetic operations like increment and decrement of data variables. This model is 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.

The second application of this method was the analysis of algorithms for self-organising networks. We want to model a network like a directed graph of nodes where for each couple of node identified by a name or a number(i1,i2) there is a link(i1,i2) able to communicate in a direct way. Starting from a intial node set to zero, we want to verify that the algorithm take each node of the network on the minimum distance from the initial node.

3.1 Ticket protocol

The ticket protocol is a 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 described below (where we use $P|Q$ to denote the interleaving parallel execution of P and Q, and $\langle\cdot\rangle$ to denote atomic fragments of code).

THE SYSTEM WITH n PROCESSES

Program
global var $s, t : integer;$
begin
$t := 0;$
$s := 0;$
$P_1 \mid \ldots \mid P_n;$
end.

THE i-th COMPONENT

Process P_i ::=
local var $a : integer;$
repeat forever
$think :$ $\langle\ a := t;$
$t := t + 1;\ \rangle$
$wait :$ **when** $\langle\ a = s\ \rangle$ **do**
$use :$ CRITICAL SECTION
$\langle\ s := s + 1;\ \rangle$
end.

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 global state of the protocol consists

of the current values of s, t, and of the local variables of $n$processes. This implies that any instance of the protocol gives rise to an infinite-state system. 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.

The infinite collection of admissible initial states consists of all configurations with an arbitrary but finite number of thinking processes and two counters having the same initial value ($t = s$). The specification is shown below.

Initial States

$$init \longrightarrow count(t) \mid turn(s) \ : \ t = s$$

Dynamic Generation

$$\epsilon \longrightarrow think \ : \ true$$

Individual Behaviour

$$think \mid count(t) \longrightarrow wait(a) \mid count(t') \ : \ a = t \ \wedge \ t' = t + 1$$

$$wait(a) \mid turn(s) \longrightarrow use \mid turn(s') \ : \ a = s \ \wedge \ s' = s$$

$$use \mid turn(s) \longrightarrow think \mid turn(s') \ : \ s' = s + 1$$

Termination

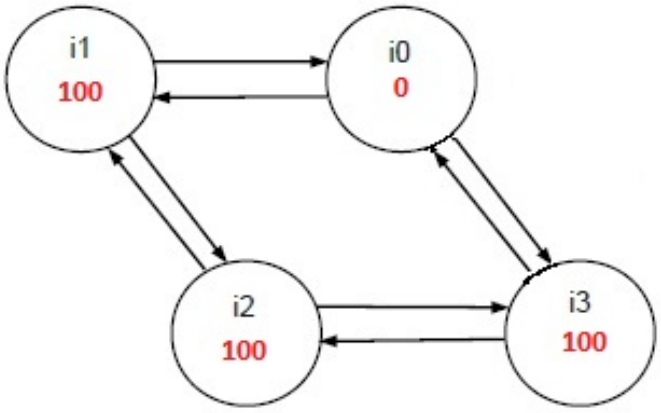
$$think \longrightarrow \epsilon \ : \ true$$

The initial configuration is the predicate init, the seed of all possible runs of the protocol. The counters are represented here via the atoms count(t) and turn(s). Thinking clients are represented via the propositional symbol think, and can be generated dynamically via the second rule. The behavior of an individual client is described via the third block, in which the relation between the local variable and the global counters are represented via constraints. At the end, we allow thinking processes to terminate their execution as specified via the last rule. The previous rules are independent from the current number of clients in the system. Note that in this specification we keep an explicit representation of the data variables. Furthermore, we do not out 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.

3.2 Self-organizing networks

Given a model of a system we want to test automatically whether this model meets a given specification or not. Typically, the specification contains safety requirements such as the absence of deadlocks and similar critical states that can lead the system to crash or to not terminate.

Our system is composed by a ring network like the one shown below where $I_0, \ldots, I_n, n > 0$ are the names of the nodes, and $0, \ldots, 100$ represent a random value of the node, our network will initially have a starting node, v(I,0). Subsequently, a finite number of secondary nodes stores the same initial value, for example 100, v(I_n,100), $n \neq 0$.

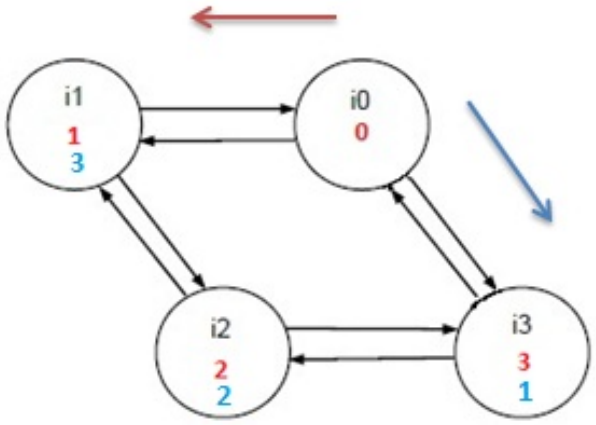


By applying the following rule of behavior we want to prove safety properties:

$$v(I,N) + link(I,I2) + v(I2,M) \longrightarrow v(I,N) + link(I,I2) + v(I2,MM)$$
$$\{M > N + 1 \wedge MM = N + 1\}$$

.

By applying this rule, we want to verify that the network evolves as follow: starting from the initial configuration, $v(I,0) + link(I,I1) + v(I1,100)$, if the constraint $M > N + 1$ is verified, then the value of the next node, v(I_2,100), becomes $MM = N + 1$. So we will obtain: $v(I,0) + link(I,I1) + v(I1,1)$. The new value MM, 1 represents also the shortest path from the selected node to the starting one.



To verify this algorithm with MSR($\mathcal{C}$) prototype we first need to write the correct specification as we did for the ticket protocol algorithm.

Initial State

Init ⟶ v(0,N) + link(0,1) + v(1,M) + link(1,2) + v(2,M) + link(2,3) + v(3,M) + link(3,0) + v(0,N) + link(0,3) + v(3,M) + link(3,2) + v(2,M) + link(2,1) + v(1,M) + link(1,0)

Rule of Behavior

v(I,N) + link(I,I2) + v(I2,M) ⟶ v(I,N) + link(I,I2) + v(I2,MM) : M > N+1 , MM = N+1

Initial State is the rule for the initialization phase. $v(0,N) + link(0,1) + v(1,M) + link(1,2) + v(2,M) + link(2,3) + v(3,M) + link(3,0) + v(0,N) + link(0,3) + v(3,M) + link(3,2) + v(2,M) + link(2,1) + v(1,M) + link(1,0)$ represents a simple ring network with four nodes and bi-directional arc.

Rule of Behavior describes how the network evolves over time.

Moreover, we need to find a set of unsafe states and ckeck them one by one so that we can verify safety properties. Given the simplicity of this network, we can easily identify which states are unsafe. Analyzing the network we can deduce that the follows can never happen:

- the value of nodes is a negative number;
- the value of the starting node is different from zero;
- the new calculated value of a node is bigger than its predecessor;
- the new calculated value of a node never drop below a certain threshold (e.g. the value of the third node can never drops below 2);
- the value of a node between two others is less than these;

Putting together all these unsafe states, we will approach a critical issue that is the size of search-space. This one should be kept as small as possible otherwise we might incur into a memory overflow error or an apparent non-termination. MSR($\mathcal{C}$) uses a backward reachability search scheme that is its start from an unsafe configuration and than it trys to go backward to the initial state. If it succeeds the system is unsafe. After having individually checked an unsafe state we can put it into an invariant. Invariants are used to prune the backward search: all constrained multisets subsumed by an invariant are removed at each step of the search so we can make searching slim and faster.

3.2.1 The MSR verification prototype

Now we have to adapt the previously specification to an MSR specification. Given a MSR specification and a specification of a set of unsafe states, MSR saturates the set of symbolic predecessors of the unsafe states during a backward reachability exploration of the MSR specification. The input syntax is related to Prolog term-based syntax. It is first necessary to define the MSR system, the unsafe states and some possible invariants. The syntax of MSR rules is as follow

rule(Head, Body,Constraint,N).

- Head and Body are Prolog lists of terms with free variables $[T_1, ..., Tn]$ or [ ] (empty list). Lists are treated as multisets by MSR.

  Terms are written as

  - $f(t_1, \ldots, t_n)$ where t_i is a term
  - c (constant, lower case)
  - X (variable, upper case)

- Constraint is a CLP constraints $\{C_1, \ldots, C_n\}$ or {} (true) where C_i is a relation over variables occurring in Head and Body

- N_i is constant used as identifier of the rule

Unsafe states are represented via the fact

unsafe($[U_1, ..., U_k]$**).**

where U_i is an unsafe term(Multiset,Constraint)

- Multiset is a list of terms

- Constraint is a constraint with free variables in Multiset (as for rules)

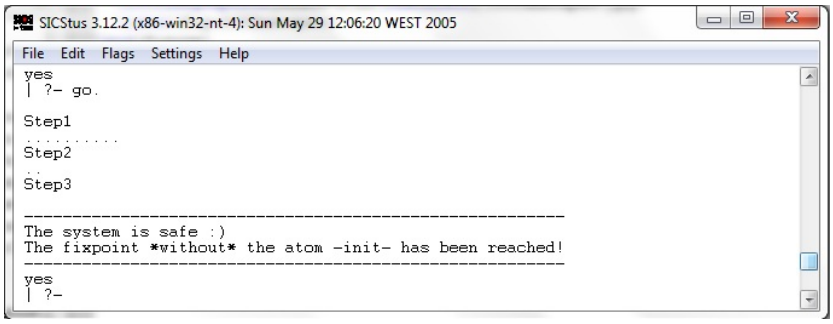
Invariants have a similar structure with "unsafe" terms replaced by "inv".

Invariants are used to prune the backward search: all constrained multisets subsumed by an invariant are removed at each step of the search.

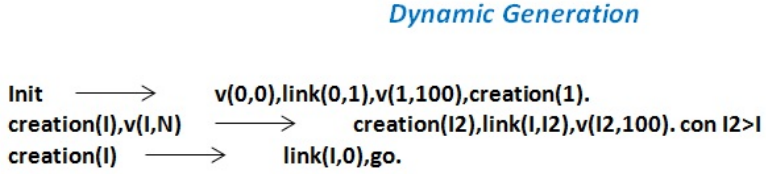
This is a MSR specification with a set of unsafe states and some possible invariants.

```
rule([init], [v(0,N),link(0,1),v(1,M),link(1,2),
              v(2,M),link(2,3),v(3,M),link(3,0)],
             {N=0 , M=100},rule1).


rule([v(I,N),link(I,I2),v(I2,M)],
     [v(I,N),link(I,I2),v(I2,MM)],
     {M>N+1 , MM=N+1},rule2).


unsafe([
      %this is a list of state useful to test rule2
       unsafe([v(I,N),link(I,I2),v(I2,N2)],{N=1 ,N2=2}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{N=3,N2=4}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{N=1,N2=1}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{N=2,N2=_}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{N=1}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{I=0,I2=1,N=0})
       ]).


invariants([
            inv([v(I,_),link(I,I2),v(I2,_)],{I2=I}),
            inv([v(_,N)],{N<0}),
            inv([v(0,N)],{N>0}),
            inv([v(I,N),link(I,I2),v(I2,N2)],{I2>I , N2<N})
            ]).
```

After launching the specification MSR will start building all predecessors. If the computation terminates we will be able to know if the system is safe or not.



After analyzing a 4 nodes ring network, we can try to find a rule for the analysis of a generic ring network with k nodes ($k \geq 1$). For this purpose we can think a rule like:



Dynamic Generation can replace the Initialization Rule mentioned before. That rule allows the generation of a dynamic ring network with N nodes.

With a similar rule we can generate various types of network, like a grid network. For the generation of a grid network we could do a porting from this two prolog programs to an MSR specification.

```
%example 1;
size(3,2).
near(p(X,Y),p(X2,Y2)):-size(N,M),
(X2 is X+1;X2=X;X2 is X-1),X2>=0,X2<N,
(Y2 is Y+1;Y2=Y;Y2 is Y-1),Y2>=0,Y2<M.
```

```
%example 2;
node(c(X,Y)):-member(X,[0,1,2,3]),member(Y,[0,1,2,3]).
link(c(X,Y),c(X,YY)):-node(c(X,Y)),
(YY is Y-1; YY is Y+1),node(c(X,YY)).
link(c(X,Y),c(XX,Y)):-node(c(X,Y)),
(XX is X-1; XX is X+1),node(c(XX,Y)).
```

Revising this specification we should be able to write a correct MSR specification to obtain a rule for a dynamic generation of grid network.

4 Conclusion

The MSR($\mathcal{C}$) protype is an excellent tool to model-check infinite state systems. However, the MSR($\mathcal{C}$) presents some significant defects. For instance, given a state we can not see which are the next ones. Furthermore, we do not have a method to debug the system so we can not verify if a single specification is actually correct. In fact, we could generate a specification whose starting configuration is a deadlock so "safe", so it is safe for definition.

Given that MSR($\mathcal{C}$) is obsolete because it is not updated, we have at our disposal another tool called PFS Prototype[3].

4.1 PFS Prototype

In this chapter we are going to revise and reuse the work presented in [3].

PFS is a prototype for the verification of parameterized systems. A parameterized system consists of an arbitrary number of identical state machines running in parallel. For instance, many cache coherence protocols as well as mutual exclusion algorithms fit into this modeling framework. The prototype performs backward reachability over an over-approximation of the system in order to check safety properties.

4.1.1 Models

The model we adopt is that of parameterized systems where each process is a finite-state machine; i.e., each process has a finite set of finite domain variables (including the control state). Transition within a process can be one of the following forms:

1. A process can change the values of its local variables and possibly move to another control state.
2. A process can check if all other processes (or all processes to its left/right) satisfy a given condition over their local variables. In case yes, the process proceeds in a similar manner to rule (1).
3. A process can check if at least one other process (or one process to its left/right) satisfies a given condition over its local variables. If yes, the process proceeds in a similar manner to rule (1).

Possible extensions consist in adding:

- finite domain shared (or global) variables which maybe change during transitions like local variables (supported by the prototype);
- broadcast transition where a process (called initiator) forces all processes which satisfy a given condition to change states simultaneously and possibly their local variables as well (supported by the prototype);

- dynamic creation and deletion of processes (supported by the prototype);
- binary communication where two processes change state simultaneously if they statisfy given conditions (not supported).

4.1.2 Properties

PFS checks parameterized system models against safety properties. For instance, given a model a configuration of the system is a sequence of process states each describing the values of local variables and the current control state. If we assume that in the process model there is a critical control state, then the set of unsafe configurations corresponds to the set of sequences containing at least two process states where the control location is critical. The goal is to check if the set of unsafe configurations is unreachable from the set of "initial" configurations regardless of the number of processes in the system.

4.2 Future Work

As described above, we can study strengths and weaknesses of the MSR($\mathcal{C}$) and PFS prototype. So we could take the best features and try to implement a new prototype using Xtext. In this way can easily develop a proper language for the MSR-PSF specification.

References

[1] G. Delzanno: *The MSR(C) language and the corresponding verification methodology*

`http://www.disi.unige.it/person/DelzannoG/MSR/INTRO/`.

[2] G. Delzanno: *An Overview of MSR(C): A CLP-based Framework for the Symbolic Verification of Parameterized Concurrent Systems.*

[3] P. A. Abdulla, G. Delzanno, N. B. Henda, A. Rezine: *Regular Model Checking without Transducers.*