I N F O R M A T I K

Model Checking Infinite-state Systems in CLP

Giorgio Delzanno
Andreas Podelski

MPI–I–98–2–012 July 1998

FORSCHUNGSBERICHT RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Im Stadtwald 66123 Saarbrücken Germany

Authors' Addresses

Giorgio Delzanno
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken
delzanno@mpi-sb.mpg.de

Andreas Podelski
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken
podelski@mpi-sb.mpg.de

Publication Notes

The present report has been submitted for publication elsewhere and will be copyrighted if accepted.

Acknowledgements

The authors would like to thank Stephan Melzer, Tevfik Bultan and Richard Gerber for fruitful discussions and encouragements, Christian Holzbaur for his help with the OFAI-clp(Q,R) library, and finally Supratik Mukhophadyay for comments.

Abstract

The verification of safety and liveness properties for infinite-state systems is an important research problem. Can the well-established concepts and the existing technology for programming over constraints as first-class data structures contribute to this research? The work reported in this paper is a starting point for the experimental evaluation of constraint logic programming as a conceptual basis and practical implementation platform for model checking. We have implemented an automated verification method in CLP using real and boolean constraints. We have used the method on a number of infinite-state systems that model concurrent programs using integers or buffers. The basis of the correctness of our implementation is a formal connection between CLP programs and the formalism used for specifying concurrent systems.

Keywords

Concurrent systems, model checking, constraint logic programming.

1 Introduction

Automated verification methods can today be applied to practical systems [McM93]. One reason for this success is that even large sets of states can be handled efficiently by using BDD's [BCM+90], which implement Boolean formulas. Boolean formulas are used as *implicit* representations of finite sets of states. The finiteness is an inherent restriction here. Many systems, however, operate on data values from an infinite domain and are intrinsically infinite-state; i.e., one cannot produce a finite-state model without abstracting away crucial properties. One important kind of examples are concurrent systems using counters or buffers. Although there is much recent effort in finding good data structures and algorithms for implicit representations of infinite sets of states, a breakthrough comparable with the event of BDD's is not in sight.

It is obvious that the metaphor of *constraints* is useful (if not unavoidable) for the implicit representation of sets of states (states are tuples of values for program variables). In this paper, we ask the question whether and how the well-established concepts and the existing technology for *programming* over constraints as first-class data structures can contribute to the research on verification of infinite-state systems. The work reported in this paper is a starting point for the experimental evaluation of constraint logic programming (CLP) [JM94] as a conceptual basis and practical implementation platform for model checking.

We have implemented an automated verification method based on model checking in CLP, and we have used the method on a number of infinite-state systems that model concurrent systems using integers or buffers. In order to properly represent complex examples our model is based on CLP programs with constraints over reals and over boolean variables (i.e. data and control variables).

The examples we consider here do not fall within classes of infinite-state systems for which decidability results are known, as in the case of *well-structured* infinite-state systems [FS98]. Thus, in general, to verify safety and liveness properties we have to apply approximation techniques as suggested by recent work in hybrid and real-time systems [HHWT97].

We have found that we should apply three criteria for the experimental evaluation. The first evaluation criterion must be whether the implementation of the method is *simple and natural.* This is the quality that determines the degree of confidence in the correctness of the implementation—obviously a central point. Another criterion is the *adaptability* and *programmability* of the considered method: can optimizations such as new execution strategies, the integration of different type of constraints, and the use of conservative approximations be incorporated directly, i.e., with relative ease and transparently with regard to correctness. This is important since verification is a hard task (undecidable in the general, infinite-state case) and often requires a fine-tuning of the method. These first two criteria are somewhat subjective, and the final conclusion must be left to the reader of our presentation. The third criterion is whether the implementation is competitive w.r.t. efficiency. An exhaustive comparison cannot be in the scope of this work. We have, however, tried several verification problems in the domain of infinite-state protocols (over integers or buffers) for which a case study on a different implementation platform already exists [BGP98, LS97].

Plan of the paper. In Section 2, we give a brief overview of related approaches. Then, after some preliminary notion of CLP (Section 3), we set the formal grounds for using CLP as a platform for model checking (Section 4). We exhibit a formal connection between CLP programs and *concurrent systems* (in the terminology of Shankar [Sha93]) which are a widely used formalism for specifying concurrent algorithms. This connection allows us to reduce the verification problem to reasoning about ground derivations, i.e., the *ground* operational semantics of CLP programs. In particular, we give a new characterization of *safety* and *liveness* properties of concurrent systems in terms of *transformations* of CLP programs. This view allows us to define a *on-the-fly* model checking algorithm in a well-founded setting, employing some well-known results on the non-ground semantics of logic programming [GDL95, MR89, RSS92] to symbolically represent set of ground states (Section 5). In Section 7 we discuss alternative strategies (forward/backward analysis) based on related research in constraint database languages.

Discovering and isolating flaws is one of the most important aims of verification tools. We briefly discuss this issues in Section 8.

In Section 6 we present several case studies where we have used the implementation (concurrent systems with unbounded variables, parametric systems, and system with complex control parts). The implementation is itself an interesting application of CLP-systems. Infact, it requires the use of different constraint solver (to handle real and boolean constraints), and the use of database-oriented techniques to efficiently store and retrieve intermediate results of the verification process. All these features are available in optimized form in SICStus Prolog. In the conclusion (Section 9) we summarize our evaluation of CLP as a platform for model checking according to the experience that we will present here and according to the three criteria given above.

2 Related Work

The work closest to ours (and, in fact, in part its inspiration) is by Bultan, Gerber and Pugh [BGP97, BGP98] who use Presburger arithmetic together with the Omega library as their implementation platform. They study some of the verification problems considered here; we give detailed comparisons in

Figure 23 in Section 9. It seems that their encoding of the model checker is not as natural and direct as ours. Our view of the model checking procedure in terms of the fixpoint semantics of logic programs allows us to apply some optimization (as we shall describe in the paper) based on the previous works such as [MR89, RSS92], which have not been applied in the approach in [BGP97]. In [BGL98] they present a composite approach combining BDD's and Integer Constraints which can be very useful to have a more efficient implementation of the boolean solver.

There exist other works relating logic programming and verification of concurrent and real-time systems. In [RRR$^+$97], XSB a logic programming language based on *tabling* is used to implement an efficient local model checker for a finite-state CCS-like value-passing language. Thanks to the power of SLG resolution, the resulting model checker, called XMC, is implemented in a natural way in XSB. Though both approaches provide for a simple implementation of the model checker, our implementation relies on meta-programming techniques and *extra* logical built-in predicates which are not necessary in the XSB-based implementation. However, XSB does not provide constraints. Furthermore, in our case-study we consider programs with potentially infinite least model (and we use approximation to deal with them). A possible integration of tabling with constraints and approximation techniques may be an interesting argument of future research.

The connection between transition systems induced by pushdown processes and *traditional* logic programs is observed in [CP98]. In the formal setting of the present paper the formalism for specifying concurrent algorithms is more high-level than transition systems; also, it lifts the characterization of temporal properties to the general case of constraints.

Fribourg and Richardson [FR96] use gap-order integer constraints in order to generate invariants for finite state systems. A transitions system is defined as a logic program with gap-order constraints [Rev93] and the initial constraints are propagated using forward analysis in order to derive invariants. Recently, Fribourg [Fri98] has extended this idea to gap-order real constraints in order to model real-time automata. However, as already remarked in Section 5, most of the examples that we considered here cannot be represented with gap-order constraints.

Urbina [Urb96] uses CLP($\mathcal{R}$) programs to specify 'hybrid systems'. He does not give a formal connection between hybrid automata (the standard model of hybrid systems) and CLP programs. He indicates several possible techniques for testing various properties of the 'hybrid system' programs, without, however, going into details or evaluating the techniques.

Among other works relating constraint programming and verification we recall [Mel97]. Melzer [Mel97] gives a conservative approximation of the test of liveness properties of Petri nets by solving linear constraints in 2lp. Here, constraints (and not programs over constraints) are used to describe sets of possibly infinite executions.

Among the most important finite-state verification tools we mention SMV [McM93] whose effectiveness and scaleability has been demonstrated in several real examples. SMV is based on a symbolic representation of states through BDD's. SMV can handle *bounded* integer variables by representing them as a collection of boolean variables (i.e. a representation of their binary encoding). In [CABN97], Chan, Anderson, Beame and Notkin present an efficient representation of arithmetic constraints (linear and non-linear) using BDD variables. Such method is based on an external *constraint solver*, used to prune the states with unsatisfiable constraints. Their approach is based on *data memoryless* and *data invariant* constraints, i.e., they do not allow constraints of the form $X = Z + z$. This type of constraints corresponds to assignments and they play a central role for representing concurrent programs, as we have shown in our case-studies.

SPIN [Hol90] implements model checking of LTL properties (with partial order reductions) for *Promela* processes. It supports integer variables with bounded values. Hence, the examples presented in this paper cannot be encoded directly in SPIN (i.e. it would be necessary to fix a range for all the variables).

An exhaustive comparison in terms of efficiency our method with all the other verification tools like HyTech [HHWT97], Concurrency Factory [CGL$^+$94] and VIS [Gro96] is beyond the scope of the paper.

3 Preliminaries: CLP

In this section we will briefly recall the main definition of an instance CLP($\mathcal{D}$) of the CLP-scheme following [JM94]. The language of constraints consists of first order formulas built on a set of predicate symbols disjointed from the set of predicate symbols. Given a $\mathcal{D}$-valuation $\theta : V \to D$, and a set $\mathcal{C}$ of constraints, $\mathcal{D} \models \mathcal{C}\theta$ iff for every $c \in \mathcal{C}$, $c\theta$ evaluates to true in $\mathcal{D}$, written $\mathcal{D} \models c\theta$. A constraint c is solvable if there exists a valuation θ such that $\mathcal{D} \models c\theta$. A constraint c entails a constraint d if for each valuation θ $\mathcal{D} \models c\theta \to d\theta$.

A *program* is a set of clauses of the form $a \leftarrow c, \tilde{B}$ where a is an atom (the head) and $c, \tilde{B}$ (the body) is such that: $\tilde{B} = b_1, \ldots, b_n$ are atoms and c is a conjunction of constraints. A *fact* is a clause $a \leftarrow c$, where c is a conjuction of constraints only. A *goal* is a conjunction of atoms and constraints. Given a program P we will indicate by $[P]_\mathcal{D}$ the set of all the instances of the clauses in P wrt $\mathcal{D}$. Notice that if I is a set of facts then $[I]_\mathcal{D} = \{a\theta \mid (a \leftarrow c) \in I,\ \mathcal{D} \models c\theta\}$. The Herbrand base $\mathcal{B}$ is the set $\{p(\tilde{d}) \mid p \in \Pi, \tilde{d} \in \mathcal{D}^k\}$. A $\mathcal{D}$-interpretation is any subset of $\mathcal{B}$.

Ground semantics. Let P be a program and let $I \subseteq \mathcal{B}$. The *immediate consequence operator* is defined as follows.

$$T_P(I) = \{p(\tilde{d}) \in \mathcal{B} \mid \quad p(\tilde{d}) : -b_1, \ldots, b_n \ in \ \ [P]_\mathcal{D}, \ b_i \in I \ for \ i : 1, \ldots, n \quad n \geq 0\}.$$

T_P is monotonic and continuos w.r.t. set inclusion, i.e., there exist the least fixpoint and the greatest fixpoint of T_P.

Nonground Semantics. Given two constrained atoms $A = p(\tilde{x}) \leftarrow \psi$ and $B = p(\tilde{y}) \leftarrow \phi$ let $A \equiv B$ iff $[A]_\mathcal{D} = [B]_\mathcal{D}$. The non-ground Herbrand base $\mathcal{B}_N$ is defined as the set of constrained facts $p(\tilde{x}) \leftarrow \psi$ (defined over the considered language) modulo the relation $\equiv$. Note that $[\mathcal{B}_N]_\mathcal{D} = \mathcal{B}$ (i.e. the ground Herbrand base). In case $\mathcal{D}$ is solution compact there exist a bijection between ground and non ground interpretations: given a collection of ground facts I there exist a unique set of (class of equivalences w.r.t. $\equiv$ of) non ground atoms J which "represents" I, s.t., for each $A \in I$ there exists $B \in J$ s.t. $[B]_\mathcal{D} = A$.

Let $\mathcal{I}$ be the powerset of $\mathcal{B}_N$. The s-semantics fixpoint of CLP programs is defined over the lattice $(\mathcal{I}, \subseteq)$ of non-ground interpretations. The *non-ground immediate consequences operator* is defined over a collection of (equivalence classes of) facts $I \in \mathcal{I}$ as follows.

$$\begin{array}{lll} S_P(I) = \{p(\tilde{X}) \leftarrow c \mid & p(\tilde{x}) : -c', b_1, \ldots, b_n \ is \ a \ variant \ of \ r \in \ P, \\ & (a_i \leftarrow c_i) \in I \ for \ i : 1, \ldots, n, \quad n \geq 0 \\ & which \ share \ no \ variables, \\ & \mathcal{D} \models c \leftrightarrow c' \wedge \bigwedge_{i=1}^{n} (c_i \wedge a_i = b_i)\}. \end{array}$$

The S_P operator is monotonic and upward continuous over $\mathcal{I}$. In [JM94], the following properties are proved, under the assumption that the constraint domain $\mathcal{D}$ is *solution compact*:

- $T_P([I]_\mathcal{D}) = [S_P(I)]_\mathcal{D}$;
- $T_P \uparrow_\omega = lfp(T_P) = [lfp(S_P)]_\mathcal{D} = [S_P \uparrow_\omega]_\mathcal{D}$;
- $T_P \downarrow_\alpha = gfp(T_P)$, $\alpha \geq \omega$;
- $S_P \downarrow_\alpha = gfp(S_P)$, $\alpha \geq \omega$.

Furthermore, the following property holds.

Proposition 3.1 *Let $\mathcal{D}$ (the constraint domain) be solution compact. Given a CLP program P, $gfp(T_P) = [gfp(S_P)]_\mathcal{D}$.*

Proof 3.1 *We first prove that $gfp(T_P) \subseteq [gfp(S_P)]_\mathcal{D}$.
By definition, $gfp(T_P) = \bigcap_{i \geq \alpha} T_P^i(\mathcal{B})$. Since $T_P(\mathcal{B}) = [S_P(\mathcal{B}_N)]_\mathcal{D}$, it holds that $T_P^i(\mathcal{B}) = [S_P^i(\mathcal{B}_N)]_\mathcal{D}$. Thus, if $A \in \bigcap_{i \geq \alpha} T_P^i(\mathcal{B})$, then for each i, $A \in T_P^i(\mathcal{B})$ and there exists $B_i \in S_P^i(\mathcal{B}_N)$, s.t. $[B_i]_\mathcal{D} = A$. Hence, the B_i's*

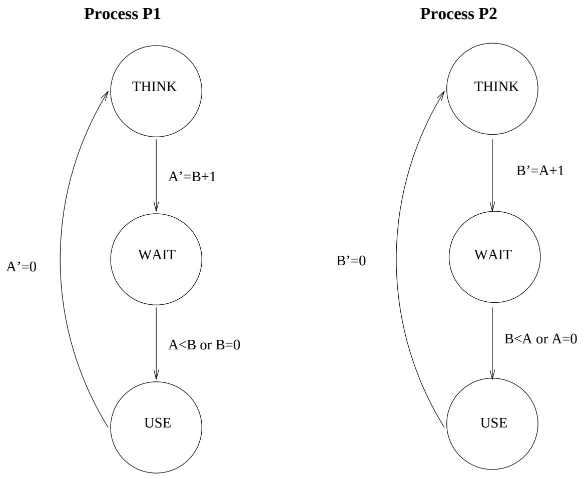


Figure 1: A graphic representation of the bakery algorithm.

are in the same equivalence class w.r.t. $\equiv$, *i.e.*, $[A]_\equiv \in \bigcap_{i\geq\alpha} S_P^i(\mathcal{B}_N)$, *i.e.*, $A \in [gfp(S_P)]_\mathcal{D}$.

It remains to prove that $[gfp(S_P)]_\mathcal{D} \subseteq gfp(T_P)$. *Since* $gfp(S_P)$ *is a fixpoint for* S_P, $[S_P(gfp(S_P))]_\mathcal{D} = [gfp(S_P)]_\mathcal{D}$. *Furthermore, by the property of* T_P *and* S_P, $[S_P(gfp(S_P))]_\mathcal{D} = T_P([gfp(S_P)]_\mathcal{D}) = [gfp(S_P)]_\mathcal{D}$. *Thus,* $[gfp(S_P)]_\mathcal{D}$ *is a fixpoint for* T_P, *i.e.*, $[gfp(S_P)]_\mathcal{D} \subseteq gfp(T_P)$.

4 Verification of Concurrent Systems in CLP

Concurrent systems consist of several processes executing simultaneously and interacting through some form of communication. In [Sha93], Shankar proposed a specification language based on *events* in which it is possible to encode traditional concurrent algorithms. Each event has an enabling condition, a predicate over a collection of *systems variables*, and an action, which updates the variables. Each process is associated with a *control variable* that indicates the atomic statement to be executed next by the process. *Data variables* keep trace of the internal state of processes. Communication is achieved by sharing data variables. The semantics of the specification language is given in terms of a *non-deterministic transition system* in which one event at a time is atomically executed. As an example, consider the following solution for the mutual-exclusion problem called the *bakery* algorithm [And91, MP95] (see Figure 1) written in a standard notation as follows:

$$\textbf{begin}\ \ turn_1 = turn_2 = 0; P_1 \mid\mid P_2\ \ \textbf{end}$$

where, $||$ indicates parallel execution of the subprograms P_1 and P_2. The process P_1 is modeled as follows:

repeat
  think : $turn_1 = turn_2 + 1$;
  wait : **when** $(turn_1 < turn_2 \vee turn_2 = 0)$ **do**
  use : [**critical section**; $turn_1 = 0$]
forever

The two processes enter in turn the critical section. The variables $turn_1, turn_2$ guarantee the fairness of the system. The labels *think, wait, use* correspond to control locations [MP95]. Note that the set of reachable states is *infinite* since the values of $turn_1$ and $turn_2$ are unbounded. Following [Sha93], this program can be encoded in the concurrent system of Figure 2. The set of

Control variables: $p_1, p_2 : \{think, wait, use\}$
Data variables: $turn_1, turn_2 : nat.$
Intial condition: $p_1 = think \wedge p_2 = think \wedge turn_1 = turn_2 = 0$
Events for $i, j : 1, 2,\ i \neq j$**:**

cond : $p_i = think$	**action** : $p'_i = wait \wedge turn'_i = turn_j + 1$
cond : $p_i = wait \wedge turn_i < turn_j$	**action** : $p'_i = use$
cond : $p_i = wait \wedge turn_j = 0$	**action** : $p'_i = use$
cond : $p_i = use$	**action** : $p'_i = think \wedge turn'_i = 0$

Figure 2: An event-based description of the bakery algorithm

events models the control flow of the two processes, where the two variables p_1 and p_2 act as program-pointers.

The encoding of concurrent programs into the event-based model has been stated in [Sha93]. For the sake of this paper, we will study the relationship between the event-based model and CLP programs. In the following we will recall Shankar's formalism. Let $\mathcal{V}$ be a set of variables such that for each $x \in \mathcal{V}$, x' (the primed version of x) is always contained in $\mathcal{V}$. We denote validity of a first-order formula ψ w.r.t. to a structure $\mathcal{D}$ and an assignment $\alpha : \mathcal{V} \rightsquigarrow \mathcal{D}$ by $\mathcal{D}, \alpha \models \psi$. Also, in the following $\alpha[\tilde{x} \mapsto \tilde{d}]$ will denote an assignment in which the tuple of variables $x_1, \ldots, x_n$, written $\tilde{x}$, is mapped into the tuple of values $d_1, \ldots, d_n$, written as $\tilde{d}$.

Definition 4.1 (Concurrent system [Sha93]) *A* concurrent system $\mathcal{S}$ *is a triple* $\langle V, \Theta, \mathcal{E} \rangle$ *s.t.*

- $V = x_1, \ldots, x_n$ *is a vector of variables in* $\mathcal{V}$,
- Θ *is a formula over* V *which represents the initial state of the system,*
- $\mathcal{E}$ *is a set of events, i.e., of pairs* $\langle \psi, \phi \rangle$ *where* ψ *is a formula over* V *(the enabling condition) and* ϕ *(the action) is a formula of the form* $(x'_1 = e_1 \wedge \ldots x'_n = e_n)$ *where* e_i *is an expression over* $x_1, \ldots, x_n$.

The primed variable x' is used to represent the value of x after the execution of an action.[1] In the examples we will use the notation **cond** : ψ **action** : ϕ to denote events. Given the tuple $(x_1, \ldots, x_n)$ of system variables a state s is a tuple $(d_1, \ldots, d_n)$ of values in $\mathcal{D}$.

Definition 4.2 (Transition Systems [Sha93]) *The transition system associated to a concurrent system $\langle V, \Theta, \mathcal{E}\rangle$ is a triple $\langle V, \Theta, \tau\rangle$ where the relation τ is defined as $\bigcup_{\xi \in \mathcal{E}} \tau_\xi$ and each transition is defined as*

$$\tau_{\langle\psi,\phi\rangle} = \{\langle \tilde{d}, \tilde{d'}\rangle \mid \mathcal{D}, \alpha[\tilde{x} \mapsto \tilde{d}, \tilde{x'} \mapsto \tilde{d'}] \models \psi \wedge \phi\}.$$

A *computation* is a (possibly infinite) sequence of states $s_0 s_1 s_2 \ldots s_i \ldots$ such that $\mathcal{D}, s_0 \models \Theta$ (i.e. s_0 is the *initial state*) and $s_{i+1} \in \tau(s_i)$.

By considering $\mathcal{D}$ as the structure of a constraint domain [JM94] it is possible to encode concurrent systems as CLP($\mathcal{D}$)-programs in a straightforward way. Again, let $(x_1, \ldots, x_n)$ be the vector of system variables. Given a set of states $\mathcal{C}$ and a set of atoms $\mathcal{A}$, $\mathcal{C} \simeq \mathcal{A}$ if $\mathcal{A} = \{p(d_1, \ldots, d_n) \mid (d_1, \ldots, d_n) \in \mathcal{C}\}$ where p is fixed predicate symbol with arity n. We extend $\simeq$ to a subset $\mathcal{C}$ of $\mathcal{D} \times \mathcal{D}$ as follows: $\mathcal{C} \simeq_{\leftarrow} \mathcal{A}$ if $\mathcal{A} = \{p(\tilde{d}) \leftarrow p(\tilde{d'}) \mid \langle \tilde{d}, \tilde{d'}\rangle \in \mathcal{C}\}$.

Definition 4.3 (CLP-encoding of Concurrent Systems) *Given a concurrent system $\mathcal{S} = \langle V, \Theta, \mathcal{E}\rangle$, the associated CLP-program $P_\mathcal{S}$ is defined by the following set of clauses:*

$$P_\mathcal{S} = \{\ p(x_1, \ldots, x_n) \leftarrow \psi \wedge \phi \wedge p(x'_1 \ldots, x'_n) \mid \langle\psi, \phi\rangle \in \mathcal{E}\ \}$$

and by the clause: initial $\leftarrow \Theta \wedge p(x_1, \ldots, x_n)$.

In the following we will consider a composite model in which a constraint ϕ can be formed by a conjunction of constraints ϕ_1 and ϕ_2 (sharing no variables) defined over different domains (specifically, real and boolean values). The relation between Shankar's specification programs and their encoding in CLP is stated by the following proposition, where T_P is the usual immediate consequences operator associated to a CLP program [JM94].

Theorem 4.1 *Let $\mathcal{S} = \langle V, \Theta, \mathcal{E}\rangle$ be a specification program and $\langle V, \Theta, \tau\rangle$ its induced transition system. Furthermore, let $P_\mathcal{S}$ be the encoding of $\mathcal{S}$ in CLP. Then, the following results hold:*

i) each ground (possibly infinite) derivation of $P_\mathcal{S} \cup \{\neg$ initial$\}$ is a computation (modulo $\simeq$) in $\mathcal{S}$ and vice versa;

ii) let $pred_\mathcal{S}(I) = \{d \mid \exists d' \in I.(d, d') \in \tau\}$, then $pred_\mathcal{S}(I) \simeq T_{P_\mathcal{S}}(I)$.

[1] In the examples we will omit assignments of the form $x' = x$.

System variables: $P_1, P_2 \in \{think, wait, use\}, Turn_1, Turn_2 \geq 0$.
Initial Condition:
$initial \leftarrow Turn_1 = 0, Turn_2 = 0, p(think, think, Turn_1, Turn_2)$.
Transitions:

$p(think, P_2, Turn_1, Turn_2)$	$\leftarrow$	$Turn_1' = Turn_2 + 1,$	$p(wait, P_2, Turn_1', Turn_2)$.
$p(wait, P_2, Turn_1, Turn_2)$	$\leftarrow$	$Turn_1 < Turn_2,$	$p(use, P_2, Turn_1, Turn_2)$.
$p(wait, P_2, Turn_1, Turn_2)$	$\leftarrow$	$Turn_2 = 0,$	$p(use, P_2, Turn_1, Turn_2)$.
$p(use, P_2, Turn_1, Turn_2)$	$\leftarrow$	$Turn_1' = 0,$	$p(think, P_2, Turn_1', Turn_2)$.
$p(P_1, think, Turn_1, Turn_2)$	$\leftarrow$	$Turn_2' = Turn_1 + 1,$	$p(P_1, wait, Turn_1, Turn_2')$.
$p(P_1, wait, Turn_1, Turn_2)$	$\leftarrow$	$Turn_2 < Turn_1,$	$p(P_1, use, Turn_1, Turn_2)$.
$p(P_1, wait, Turn_1, Turn_2)$	$\leftarrow$	$Turn_1 = 0,$	$p(P_1, use, Turn_1, Turn_2)$.
$p(P_1, use, Turn_1, Turn_2)$	$\leftarrow$	$Turn_2' = 0,$	$p(P_1, think, Turn_1, Turn_2')$.

Figure 3: CLP-encoding of the program in Figure 2.

Proof 4.1 *We first observe the following fact. Let* $\xi = \langle \psi, \phi \rangle$ *and let* $c_\xi = p(\tilde{x}) \leftarrow \psi \wedge \phi \wedge p(\tilde{x}')$ *be the corresponding clause in* $P_\mathcal{S}$*. Then, it holds that* $[c]_\mathcal{D} = \{p(\tilde{d}) \leftarrow p(\tilde{d'}) \mid \mathcal{D}, \alpha[\tilde{x} \mapsto \tilde{d}, \tilde{x}' \mapsto \tilde{d'}] \models \psi \wedge \phi\} \simeq_\leftarrow \tau_\xi$*. According to this, point* i) *easily follows by induction on the length of a computation (the vice versa on the length of a derivation). Point* ii) *immediately follows by definition of* T_P*.*

As an example, the system for the bakery algorithm of Fig. 2 is encoded in the CLP-program of Fig. 4. In order to discuss a CLP-based analysis of concurrent system, we need the following definitions.

Definition 4.4 (Programs with oracles) *Let* $\mathcal{F}$ *be a set of states and* P *be a CLP-program encoding a concurrent system* $\mathcal{S}$*. The (possibly infinite) programs with oracles* $P \oplus \mathcal{F}$ *and* $P \oslash \mathcal{F}$ *are defined as follows:*

- $P \oplus \mathcal{F} = P \cup \{p(\tilde{d}) \mid \tilde{d} \in \mathcal{F}\}$
- $P \oslash \mathcal{F} = \{p(\tilde{x}) \leftarrow c \wedge p(\tilde{x}') \wedge \tilde{x} = \tilde{d} \mid p(\tilde{x}) \leftarrow c \wedge p(\tilde{x}') \in P, \ \tilde{d} \in \mathcal{F}\}$

Remark 4.1 *Note that the following properties hold:* $T_{P \oplus \mathcal{F}} = \lambda I.\mathcal{F} \cup T_P(I)$ *and* $T_{P \oslash \mathcal{F}} = \lambda I.\mathcal{F} \cap T_P(I)$*.*

The definition of program with oracles will be useful to relate the ground semantics of logic program with CTL properties of concurrent systems. In practice, we will consider oracles which can be represented by a *finite set* of constrainted atoms of the form $p(\tilde{x}) \leftarrow c$. Furthermore, for each such a constraint c, we require that $\neg c$ can be represented by a *finite disjunction* of constraints. In the sequel of the paper we will show that this class of constrained atoms is powerful enough to express properties useful for the verification of safety and liveness condition.

4.1 Temporal Logic

For the sake of this paper, we restrict ourselves to consider the fragment of CTL, the branching-time *Computation Tree Logic* [Eme90], consisting of formulas built over $\wedge, \vee, \neg$ and the temporal connectives: EF (exists finally), EG (exists globally), AF (always finally), AG (always globally). These connectives are useful to express *safety* and *liveness* properties of transition systems. Given a CTL formula f and a state s_0 the semantics of the temporal operators w.r.t. the (maximal) computations of a transition system $\mathcal{S}$ is defined as follows (the semantics of other connectives is defined as usual w.r.t. the considered state):

- $s_0 \models EF(f)$ if for *some* path $(s_0 s_1 s_2 \ldots)$, for *some* i, $s_i \models f$;
- $s_0 \models EG(f)$ if for *some* path $(s_0 s_1 s_2 \ldots)$, for *all* i, $s_i \models f$;
- $s_0 \models AF(f)$ if for *all* paths $(s_0 s_1 s_2 \ldots)$, for *some* i, $s_i \models f$. Also, $AF(f) \equiv \neg\, EG(\neg f)$;
- $s_0 \models AG(f)$ if for *all* paths $(s_0 s_1 s_2 \ldots)$, for *all* i, $s_i \models f$. Also, $AG(f) \equiv \neg\, EF(\neg f)$.

According to [Eme90], in the following we will identify a *temporal property* $\mathcal{F}$ of the system $\mathcal{S}$ with the *set of states* satisfying it. In the following, we will identify $\mathcal{F}$ with the set of facts $\mathcal{F}'$ such that $\mathcal{F} \simeq \mathcal{F}'$. Furthermore, we will use $[F]_{\mathcal{D}}$ to denote the ground instances of a set of constrained facts F and $\mathcal{B}$ the Herbrand base of a CLP program, i.e. the set of all the possible states. The following results hold.

Theorem 4.2 (CTL and Fixed Points) *Let $\mathcal{S}$ be a transition system, P be its CLP-encoding, $\mathcal{F}$ a set of states, then*

$$
\begin{array}{rcl@{\quad}c@{\quad}l}
EF(\mathcal{F}) & \simeq & lfp(T_{P\oplus\mathcal{F}}) & = & [lfp(S_{P\oplus\mathcal{F}})]_{\mathcal{D}} \\
EG(\mathcal{F}) & \simeq & gfp(T_{P\oslash\mathcal{F}}) & = & [gfp(S_{P\oslash\mathcal{F}})]_{\mathcal{D}} \\
AF(\mathcal{F}) & \simeq & \mathcal{B} \setminus gfp(T_{P\oslash(\neg\mathcal{F})}) & = & \mathcal{B} \setminus [gfp(S_{P\oslash(\neg\mathcal{F})})]_{\mathcal{D}} \\
AG(\mathcal{F}) & \simeq & \mathcal{B} \setminus lfp(T_{P\oplus(\neg\mathcal{F})}) & = & \mathcal{B} \setminus [lfp(S_{P\oplus(\neg\mathcal{F})})]_{\mathcal{D}}.
\end{array}
$$

Proof 4.2 *Following [Eme90], $EF(\mathcal{F}) = \mu I.\mathcal{F} \cup pred_{\mathcal{S}}(I)$, $EG(\mathcal{F}) = \nu I.\mathcal{F} \cap pred_{\mathcal{S}}(I)$. By Theorem 4.1, $EF(\mathcal{F}) = \mu I.\mathcal{F} \cup T_{P_{\mathcal{S}}}(I)$ and $EG(\mathcal{F}) = \nu I.\mathcal{F} \cap T_{P_{\mathcal{S}}}(I)$. We recall the μ and /nu correspond to the least and greatest fixpoint operators. Thus, by remark 4.1, $EF(\mathcal{F}) = lfp(T_{P_{\mathcal{S}}\oslash\mathcal{F}})$ and $EG(\mathcal{F}) = lfp(T_{P_{\mathcal{S}}\oplus\,\neg\mathcal{F}})$. The relation with the ground and non-ground least fixpoint semantics is given in [JM94, GDL95]. We give a proof of the fact $gfp(T_P) = [gfp(S_P)]_{\mathcal{D}}$ in Section 4. The other results can be derived using the laws of the connectives EG and EF.*

We focus now on two important classes of properties.

A *safety* property asserts that the program never enters a bad state, i.e., one in which some variables have undesirable values. Mutual exclusion is an important example for the verification of concurrent systems. A safety property can be expressed as the temporal formula $AG(\neg\mathcal{U}) = \neg EF(\mathcal{U})$ where $\mathcal{U}$ represents the set of the *unsafe states*. It is satisfied if and only if $\Theta \cap EF(\mathcal{U}) = \emptyset$ [Eme90]. Thus, the following result holds.

Corollary 4.3 (Safety properties) *The safety property $AG(\neg\mathcal{U})$ is satisfied by the concurrent system $\mathcal{S}$ if and only if*

$$\Theta \cap lfp(T_{P_\mathcal{S} \oplus \mathcal{U}}) = \emptyset.$$

A *liveness* property asserts that the program eventually enters a good state, i.e., one in which the variables all have desirable values, e.g. starvation-freedom for concurrent systems. Starvation freedom can be expressed as the temporal formula $AG(\mathcal{W} \rightarrow AF(\mathcal{C})))$ where $\mathcal{W}$ represents the states in which a process *waits* to enter the critical section and $\mathcal{C}$ the states in which the process is in the critical section. Such aproperty is satisfied if and only if $\Theta \cap EF(\mathcal{W} \wedge \neg AF(\mathcal{C})) = \emptyset$ [Eme90]. Thus, the following result holds.

Corollary 4.4 (Liveness properties) *The liveness property $AG(\mathcal{W} \rightarrow AF(\mathcal{C})))$ is satisfied by the concurrent system $\mathcal{S}$ if and only if*

$$\Theta \cap lfp(T_{P'}) = \emptyset,$$

where $P' = P_\mathcal{S} \oplus (\mathcal{W} \cap gfp(T_{P_\mathcal{S} \oslash \neg\mathcal{C}}))$.

In the following section we will discuss how to make effective the application of the above illustrated characterizations.

5 Model Checking

As shown in [GDL95, JM94] the non-ground semantics of CLP programs based on the operator S_P can be used to *simulate* the corresponding ground semantics based on the operator T_P. Such operator is defined as follows.

Definition 5.1 (S_P operator [JM94, GDL95]) *Let L be an collection of constrained facts of the form $p(\tilde{r}) \leftarrow \psi$.*[2] *Then,*

$$\begin{array}{rll} S_P(L) = \{ \ B & | & (p(\tilde{t}) \leftarrow p(\tilde{s}) \wedge \phi) \in P, \\ & & (p(\tilde{r}) \leftarrow \psi) \in L, \quad \tilde{x} = var(\tilde{t}), \\ & & \mathcal{D} \models \gamma \leftrightarrow \exists_{-\tilde{x}}(\phi \wedge \psi \wedge \tilde{s} = \tilde{r}), \\ & & B = p(\tilde{t}) \leftarrow \gamma \ \ \} \end{array}$$

Since S_P relies on *unification* (or better simplification of constraints) instead of *instantiation*, it is possible to explore the search space of a program by

[2] In case of composite domains: $\phi = \phi_b \wedge \phi_r$, $\psi = \psi_b \wedge \psi_r$, $\gamma = \gamma_b \wedge \gamma_r$, $\mathcal{R} \models \gamma_r \leftrightarrow \exists_{-\tilde{x}}(\phi_r \wedge \psi_r \wedge \tilde{s} =_r \tilde{r})$ and $\mathcal{R} \models \gamma_b \leftrightarrow \exists_{-\tilde{x}}(\phi_b \wedge \psi_b \wedge \tilde{s} =_b \tilde{r})$, where $t =_s r$ isolates the constraints of type s from $t = r$.

using a symbolic representation of its states: constrained facts are used to symbolically represent the set of their instances (e.g. $A = p(X,Y) \leftarrow X > Y$ is used to denote $[A]_\mathcal{D} = \{p(d,d') \mid d > d'\}$). However, the computation of the S_P-based semantics can still be problematic. Consider the CLP($\mathcal{R}$)-program $\{p(X) \leftarrow Y = X - 1, p(Y)\}$ and the fact $p(X) \leftarrow X > 0$. Clearly, the least model of T_P consists of all $p(d)$ such that $d \in \mathcal{R}^+$ and it can be reached in one step (considering an infinite set of initial facts). On the other hand, the least fixpoint of the S_P operator is reached after ω-steps. It consists of the constrained atoms $\{p(X) \leftarrow X > 0, p(X) \leftarrow X > 1, p(X) \leftarrow X > 2, \ldots\}$. Thus, in general the S_P operator introduces *redundant* elements which may lead to infinite chains. Adapting the ideas in [MR89, RSS92] to our specific setting we will use the S_P operator in conjunction with a subsumption test. Unfortunately, in case of linear constraints over the reals there exists no polynomial time algorithm for subsumption [Sri92]. However, given a constrained fact A and a collection of facts I, a sufficient condition for I *subsumes* A is the following: $\exists B \in I$ s.t. B subsumes A. In the examples discussed in the next section we have applied such a condition in order to reduce the complexity of the algorithm. Infact, in many practical cases the complete test is not necessarry to achieve termination. Now, let us introduced the definition of the on-the-fly model checker used in our prototype. We say that the constrained atom A *subsumes* B if $[B]_\mathcal{D} \subseteq [A]_\mathcal{D}$. We extend this relation to set of constrained atoms in the natural way.

Definition 5.2 (Irredundant interpretations) *A set of constrained atoms I is irredundant if there exists no $A \in I$ such that for a distinct element $B \in I$, A subsumes B.*

Given an interpretation I let *reduce*(I) be the corresponding irredundant interpretation. Such operation may require n^2 subsumption tests where n is the cardinality of I.[3] Note that $[I]_\mathcal{D} = [reduce(I)]_\mathcal{D}$. Let P be a binary program (without facts), and UNSAFE be a set of constrained facts. The symbolic representation $\mathcal{F}$ of the least model of P is computed by the procedure LFP in Fig. 4. Then, the following result holds.

Proposition 5.1 *Let P be a binary program, if the procedure* LFP *terminates, then the resulting collection of facts $\mathcal{F}$ is such that $[\mathcal{F}]_\mathcal{D} = lfp(T_P)$.*

Proof 5.1 *Let $T_P \uparrow_0 = [\text{UNSAFE}]_\mathcal{D}$, $T_P \uparrow_{i+1} = T_P(T_P \uparrow_i)$, and $T_P \uparrow_\omega = \bigcup_i T_P \uparrow_i$. Note that P is a binary program without facts, thus, this definition of $lfp(T_P)$ is equivalent to the standard one. We prove that at the i-th step of the algorithm $[\text{LAST}]_\mathcal{D} = T_P \uparrow_i$. The fact holds for $i = 0$. Let us assume that it holds for $j \leq i$. All the steps of the algorithm preserve the set of ground instances of the corresponding set of constrained facts.*

[3] In case of linear contraints over $\mathcal{R}$, the cost of each subsumption tests (e.g. checking *entailment*) is polynomial in the size of the constraints of the considered atoms.

procedure LFP
LAST = CURRENT = *reduce*(UNSAFE);
loop
LAST = *reduce*(S_P(LAST));
if *initial* $\in$ LAST
then exit(*the system is unsafe*);
else if CURRENT *subsumes* LAST
then $\mathcal{F}$ = CURRENT;
exit(*the system is safe*);
else CURRENT = *reduce*(CURRENT $\cup$ LAST);
end

procedure GFP
CURRENT = $\{p(\tilde{x}) \leftarrow true\}$
loop
NEXT = *reduce*(S_P(CURRENT));
if NEXT *subsumes* CURRENT
then $\mathcal{G}$ = CURRENT;
exit(*greatest fixpoint*);
else CURRENT = NEXT;
end

Figure 4: Informal procedures for the least and greatest fixpoint

Furthermore, $[S_P(\text{LAST}_i)]_{\mathcal{D}} = T_P([\text{LAST}_i]_{\mathcal{D}})$. *Thus, by inductive hypothesis,* $[\text{LAST}_{i+1}]_{\mathcal{D}} = T_P(T_P \uparrow_i) = T_P \uparrow_{i+1}$*, as desired.*

A similar idea can be applied to the computation of the greatest fixpoint of T_P (in case it is reachable in a finite number of steps). The symbolic representation $\mathcal{G}$ of the greatest fixpoint of T_P is computed by the procedure GFP in Fig. 4.

Proposition 5.2 *Let P be a binary program and let $gfp(T_P)$ be computable in a finite number of steps. Then, the collection of facts $\mathcal{G}$ computed by the procedure* GFP *is such that* $[\mathcal{G}]_{\mathcal{D}} = gfp(T_P)$.

Proof 5.2 *We first note that* $p(\tilde{x}) \leftarrow true \equiv \mathcal{B}_N$*, i.e.,* $[p(\tilde{x}) \leftarrow true]_{\mathcal{D}} = [\mathcal{B}_N]_{\mathcal{D}}$. *The proof is by induction on the number of steps of the algorithm, by noting that* $[S_P(\text{CURRENT})]_{\mathcal{D}} = T_P([\text{CURRENT}]_{\mathcal{D}})$.

5.1 Computing Upper Bounds to Least and Greatest Fixpoints

CTL properties of concurrent systems are, in general, undecidable. However, decidability results have been found for many examples of infinite-state systems, the so called *well-structured* systems [FS98], (e.g. real-time automata [ACD90]). Similarly, there exist classes of CLP programs for which the fixpoint semantics is guaranteed to terminate, e.g., Datalog$^{<_z}$ [Rev93] an extension of Datalog with gap-order constraints. Unfortunately, such results cannot be applied to many intersesting examples of concurrent programs. The reason seems to be the presence of *assigments* constraints of the form $X = Y + z$ which are not allowed in the above mentioned classes of system (programs).

In order to analyze a generic concurrent system (in which the guards and the actions contain linear constraints) it is possible to adopt techniques developed in the field of abstract interpretation. In particular, Cousot and

```
procedure ALFP
 LAST = CURRENT = reduce(UNSAFE);
 loop
   LAST = reduce(S_P(LAST));
   if initial ∈ LAST
     then exit(don't know);
     else if CURRENT subsumes LAST
          then  F = CURRENT;
                exit(the system is safe);
          else  LAST = reduce(upper(LAST, CURRENT));
                CURRENT = reduce(LAST ∪ CURRENT);
end
```

Figure 5: Least fixpoint with approximation.

Halbwachs [CH78], Halbwachs, Proy and Romanoff [HPR96], and Henzinger and Ho [HH95] have studied approximation techniques based on a geometric view of constraints (e.g. *convex hull*, *extrapolation*, *widening* and *narrowing*). In our prototype we have used a sort of *widening* operator (see [CH78]), to compute an upper bound $\mathcal{F}^{\#}$ of the least fixpoint of a program. Note infact that, if $\mathcal{F} \subseteq \mathcal{F}^{\#}$ and $(\Theta \cap [\mathcal{F}^{\#}]_{\mathcal{D}} = \emptyset)$ then $(\Theta \cap [\mathcal{F}]_{\mathcal{D}} = \emptyset)$. The modified procedure is shown in Fig. 5. Here $upper(I, J)$ is such that $[I]_{\mathcal{D}} \subseteq [upper(I, J)]_{\mathcal{D}}$. The idea is to relax the constraints in I by using the information computed in the previous steps (i.e. J). More precisely:

Definition 5.3 (Approximation) *A constrained atom A in $upper(I, J)$ is obtained by a constrained atom $B = p(\tilde{x}) \leftarrow c_1 \wedge \ldots \wedge c_n \in I$, by removing all constrained atoms c_i such that for some $C = p(\tilde{x}) \leftarrow d_1 \wedge \ldots \wedge d_m \in J$:*

- $[B]_{\mathcal{D}} \cap [C]_{\mathcal{D}} \neq \emptyset$,
- *for some d_j, d_j strictly entails c_i, i.e., $\mathcal{D} \models d_j \rightarrow c_i$ and $\mathcal{D} \not\models c_i \rightarrow d_j$.*

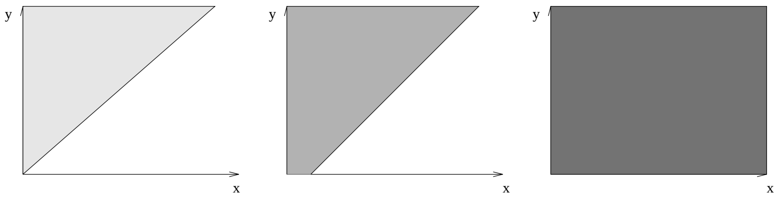
For instance, let us consider the two facts $B = p(X, Y) \leftarrow X \geq 0, Y \geq 0, X < Y$ and $C = p(X, Y) \leftarrow X \geq 0, Y \geq 0, X < Y + 1$. As shown in Figure 6, $upper(\{C\}, \{B\}) = p(X, Y) \leftarrow X \geq 0, Y \geq 0$, i.e., the algorithm tries to guess the direction of growth of the regions represented by the constrained facts. The following property holds.

Proposition 5.3 *Let P be a binary program, if the procedure* ALFP *terminates, then the collection of facts $\mathcal{F}^{\#}$ is such that $lfp(T_P) \subseteq [\mathcal{F}^{\#}]_{\mathcal{D}}$.*

Proof 5.3 *It follows from proposition 5.1 and from the fact*

$$[\text{LAST}]_{\mathcal{D}} \subseteq [upper(\text{LAST}, \text{CURRENT})]_{\mathcal{D}}$$

, which holds since the procedure 'upper' just relaxes the constraints in the facts contained in LAST.



$X \geq 0, Y \geq 0, X < Y$ $\quad$ $X \geq 0, Y \geq 0, X < Y + 1$ $\quad$ $upper(C, B)$

Figure 6: The widening operator *upper* in action.

Definition 5.3 is slightly different from Cousot and Halbwachs's widening operator [CH78], namely $J \nabla I$. They relax the constraint of the elements in J, removing all the contraints in J which do not hold in I. Though their definition ensures termination, our operator may yield finer grained results. Consider the following rule $p(X) \leftarrow p(Y), X < 5$ and the single fact $A = p(X) \leftarrow X > 6$. After one step, the least fixpoint iteration defined as $I_0 = \{A\}$, $I_{i+1} = I_i \nabla S_P(I_i)$ converges to the least fixpoint $p(X) \leftarrow \mathit{true}$ (since $X > 6$ is not implied by $p(X) \leftarrow X < 5 \vee X > 6$), whereas, our algorithm converges in two steps to the exact least fixpoint $\{p(X) \leftarrow X < 5, p(X) \leftarrow X > 6\}$.

For liveness properties, we can compute an upper bound $\mathcal{G}^{\#}$ of the greatest fixpoint by stopping the iterations after a fixed number of steps. Since $T_{P \oplus I} \subseteq T_{P \oplus I^{\#}}$, whenever $I \subseteq I^{\#}$, if $\mathcal{G} \subseteq \mathcal{G}^{\#}$ and $(\Theta \cap lfp(T_{P \oplus (\mathcal{W} \cap \mathcal{G}^{\#})}) = \emptyset)$ then it follows that $(\Theta \cap lfp(T_{P \oplus (\mathcal{W} \cap \mathcal{G})}) = \emptyset)$. Furthermore, the outermost fixpoint computation can be approximated using the above described technique. Hence we have:

$$((\Theta \cap lfp(T_{P \oplus (\mathcal{W} \cap \mathcal{G}^{\#})}))^{\#} = \emptyset) \;\Rightarrow\; (\Theta \cap lfp(T_{P \oplus (\mathcal{W} \cap \mathcal{G})}) = \emptyset)$$

Based on these ideas we have implemented a *fixpoint-machine* to compute exact (when possible) and approximate fixpoints and to verify the above discussed properties.

5.2 Notes on the implementation

Our prototype has been developed in SICStus Prolog (around 340 lines of code). Such a system provides: efficient solving of linear constraints over rationals and reals (Holzbaur's $clp(Q, R)$ library [Hol95]) and of boolean constraints; efficient storage and retrieval of data (thanks to the efficient management of the run-time database of SICStus); interchangeability between uninterpreted and interpreted constraints terms. The combination of meta- and database-oriented programming allows us to efficiently manipulate programs and intermediate results of the analysis: the constrained facts computed by the procedures discussed above are asserted in the program database (or in case of very large systems they can be stored in external files) and a backtracking-based programming style is used to compute each iteration step. It is interesting to show how to implement the *semantic* operations over constrained atoms by using the above mentioned meta-facilities of the $clp(Q,R)$ library. As an example, subsumption of constrained atoms can be implemented as shown in Figure 7, where according to the SICStus syntax a constrained atom $p(\tilde{t}) \leftarrow c_1 \wedge \ldots \wedge c_n$ is represented as $(p(\tilde{t}), \{c_1, \ldots, c_n\})$. The predicate `subsumes_chk` is a SICStus built-in predicate to check sub-

```
constrained_fact_subsumption((Atom,Const),(Atom1,Const1)):-
   subsumes_chk(Atom,Atom1),
   unify_with_occurs_check(Atom,Atom1),
   constraints_subsumption(Consts,Const1).

constraints_subsumption({},{}).

constraints_subsumption({C},{D}):-
   call_residue(entails({C},(D)),_).

constraints_subsumption({C},{}):-
   call_residue({C},_).

entails(C,D):-
   C,entailed(D).
```

Figure 7: A fragment of code.

sumption of terms, whereas `call_residue` and `entailed` are predicates of the $clp(Q,R)$-library. The former allows us to locally check satisfiability of constraints (and to retrieve the corresponding normal form): the store is local to the invocation of `call_residue`. The latter checks if the constraint passed as parameters is entailed by the current store. The combination of *call_residue* (encapsulation) and *entailed*, as described by the third clause in the example, provides us the desired *local* subsumption test. The approximation sketched in the previous section is implemented by using the entailment predicated described above. When a newly added atom A must be added, it is first compared with an existing atom B in the database. The algorithm first unifies a copy A' of A with B. Then, if a conjunct of A' has to be removed, the corresponding conjunct in A is removed, as well. The process successfully terminates if the constraints obtained after unifying A' and B are satisfiable. The entire program is very small (340 lines of SICStus code) and it is independent from the considered concurrent system and the property to verify. As an example, we include a simplified version of the program in the appendix: program A handles real constraints only with approximation techniques. The code of the complete prototype is more

BAKERY
System variables: $p(P_1, P_2, A, B),\ A, B \geq 0$.
Initial Condition: $initial \leftarrow A = 0, B = 0, p(think, think, A, B)$.
Transitions:

$$
\begin{array}{lcll}
p(think, P, A, B) & \leftarrow & A \geq 0, A_1 = B + 1, & p(wait, P, A_1, B). \\
p(wait, P, A, B) & \leftarrow & A < B, & p(use, P, A, B). \\
p(wait, P, A, B) & \leftarrow & B = 0, & p(use, P, A, B). \\
p(use, P, A, B) & \leftarrow & A \geq 0, A_1 = 0, & p(think, P, A_1, B). \\
p(P, think, A, B) & \leftarrow & B \geq 0, B_1 = A + 1, & p(P, wait, A, B_1). \\
p(P, wait, A, B) & \leftarrow & B < A, & p(P, use, P, A, B). \\
p(P, wait, A, B) & \leftarrow & A = 0, & p(P, use, P, A, B). \\
p(P, use, A, B) & \leftarrow & B \geq 0, B_1 = 0, & p(P, think, A, B_1).
\end{array}
$$

Figure 8: A graphic representation of the bakery algorithm.

elaborated and includes a treatment of boolean constraints.

We have easily incorporated other forms of analysis (e.g. *forward reachability* analysis and mixed backward/forward) in our implementation, by using trasformations of the original CLP programs such as the *magic-set template* algorithm [RSS92]. We will discuss this point in the next section.

6 Examples

In this section we will discuss some examples of concurrent systems with an infinite state space, in which the data variables may have unbounded values. We also discuss example of verification of parametric representations of concurrent systems, i.e., we add extra variables (e.g. to represent the size of a buffer) which allow us to verify *program-schemes.* The following examples are defined for integer and boolean data variable. The enabling conditions over intger variables are defined as systems of constraints of the form $A\mathbf{x} \leq b$. In our approach we will consider the real relaxation of the integer problem. The conditions over boolean variables are defined as propositional formulas. We will also use flat terms to denote locations.

6.1 Bakery Algorithm

The program BAKERY in Figure 8 corresponds to the concurrent system depicted in Figure 1. In such a CLP-program we add constraints of the form $A \geq 0$ to the variables which do not occur in the body of the clauses. This allows us to take into account the type information associated to the data variables in Figure 2. The mutual exclusion property of the bakery algorithm of Figure 2 can be formulated as the CTL-property: $AG(\neg(p_1 = use \wedge p_2 = use))$. Now, let P be the CLP-program Figure 8. The set of *unsafe states* $\mathcal{U}$ is represented by the fact $p(P_1, P_2, A, B) \leftarrow\ P_1 = use \wedge P_2 = use \wedge A \geq 0 \wedge B \geq 0$. According to Corollary 4.3, the property holds iff

$$p(think, think, 0, 0) \notin [lfp(S_{P \oplus \mathcal{U}})]_{\mathcal{R}}$$

The fixpoint can be computed in a finite number of steps (the result is shown in Figure 9). The execution required by our prototype is 0.3s on a

$$\begin{array}{ll}
p(wait, wait, x, 0) \leftarrow x >= 0.0 & p(wait, use, x, 0) \leftarrow x >= 0.0 \\
p(use, wait, x, 0) \leftarrow x >= 0.0 & p(use, wait, x, y) \leftarrow x - y > 0.0, y >= 0.0 \\
p(think, wait, x, 0) \leftarrow x >= 0.0 & p(wait, use, x, y) \leftarrow x - y < 0.0, x >= 0.0 \\
p(think, use, x, 0) \leftarrow x >= 0.0 & p(wait, think, 0, x) \leftarrow x >= 0.0 \\
p(use, think, 0, x) \leftarrow x >= 0.0 & p(wait, use, 0, x) \leftarrow x >= 0.0 \\
p(use, use, x, y) \leftarrow true & p(wait, wait, 0, x) \leftarrow x >= 0.0 \\
 & p(use, wait, 0, x) \leftarrow x >= 0.0
\end{array}$$

Figure 9: Testing mutual exclusion for BAKERY.

Sun-Sparc Station 4, OS 5.5.1.

Now, let $\mathcal{W}$ be the set of states s.t. $P_1 = wait$ and $\mathcal{C}$ be the set of states s.t. $P_1 = use$. The starvation freedom property for the bakery algorithm is represented by the CTL assertion $AG(p_1 = wait \rightarrow AF(p_1 = use))$. According to Corollary 4.4, the considered system is safe iff

$$p(think, think, 0, 0) \notin [lfp(S_{P'})]_{\mathcal{R}}$$

where $P' = P \oplus (\mathcal{W} \cap [gfp(S_{P \oslash \neg \mathcal{C}})]_{\mathcal{R}})$. The program P' is depicted in Fig. 10. The inner computation, i.e., $gfp(S_{P \oslash \neg \mathcal{C}})$ converges after 9 steps. The resulting fixed point is shown in Figure 11. The intersection with $\mathcal{W}$ can be computed by simply selecting the facts consistent with the constraint $P_1 = wait$.

$$\begin{array}{llll}
p(think, P_2, A, B) & \leftarrow & A_1 = B + 1, A \geq 0, & p(wait, P_2, A_1, B). \\
p(wait, P_2, A, B) & \leftarrow & A < B, & p(use, P_2, A, B). \\
p(wait, P_2, A, B) & \leftarrow & B = 0, & p(use, P_2, A, B). \\
p(wait, think, A, B) & \leftarrow & B_1 = A + 1, B \geq 0, & p(wait, wait, A, B_1). \\
p(wait, wait, A, B) & \leftarrow & B < A, & p(wait, use, P_2, A, B). \\
p(wait, wait, A, B) & \leftarrow & A = 0, & p(wait, use, P_2, A, B). \\
p(wait, use, A, B) & \leftarrow & B \geq 0, B_1 = 0, & p(wait, think, A, B_1). \\
p(think, think, A, B) & \leftarrow & B_1 = A + 1, B \geq 0, & p(think, wait, A, B_1). \\
p(think, wait, A, B) & \leftarrow & B < A, & p(think, use, P_2, A, B). \\
p(think, wait, A, B) & \leftarrow & A = 0, & p(think, use, P_2, A, B). \\
p(think, use, A, B) & \leftarrow & B \geq 0, B_1 = 0, & p(think, think, A, B_1).
\end{array}$$

Figure 10: $bakery \oslash \neg(P_1 = use)$.

The outer fixpoint computation converges in 1 step. The initial state does not belong to the resulting collection of facts. This proves the property. The entire computation is performed automatically by our prototype in 1.3s.

$$
\begin{array}{ll}
p(wait, use, 0, x) \leftarrow x >= 0.0 & p(wait, wait, 0, x) \leftarrow x >= 0.0 \\
p(think, think, 0, x) \leftarrow x >= 0.0 & p(wait, think, 0, x) \leftarrow x >= 0.0 \\
p(think, wait, 0, x) \leftarrow x >= 0.0 & p(think, use, 0, x) \leftarrow x >= 0.0
\end{array}
$$

Figure 11: Testing starvation freedom for BAKERY.

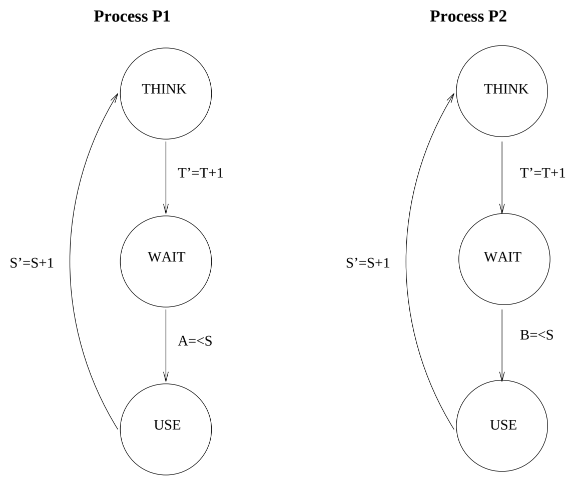


Figure 12: Automata for the ticket algorithm.

6.2 Ticket Algorithm

The ticket-algorithm shown in Figure 12 (see [And91]) is another solution to the mutual-exclusion problem. Differently from the bakery algorithm, priorities are handled using two global variables T and S. The variable T is used to assign new priorities (tickets) to processes waiting for entering their critical section. The variable S is used to keep the value of the ticket of the next process to be served. The resulting translation in CLP, program TICKET, in given in Figure 13. Similarly to the analysis of the bakery algorithm, the mutual exclusion property for the ticket algorithm can be characterized as follows:

$$p(think, think, 0, 0, n, n) \not\in [lfp(S_{\mathcal{P}\oplus\mathcal{U}})]_{\mathcal{R}}, \quad for\ n \in \mathcal{Z}.$$

where $\mathcal{U}$, the set of unsafe states, is represented by $p(P_1, P_2, A, B, T, S) \leftarrow P_1 = use \wedge P_2 = use \wedge A \geq 0 \wedge B \geq 0 \wedge T \geq 0 \wedge S \geq 0$. The computation of the fixpoint diverges. Let $\mathcal{F}^{\#}$ be the limit of the approximated chain for $S_{\mathcal{P}\oplus\mathcal{U}}$ as defined in Section 5. By Prop. 5.3, the mutual-exclusion property of the ticket algorithm holds if

$$p(think, think, 0, 0, n, n) \not\in [\mathcal{F}^{\#}]_{\mathcal{R}}, \quad for\ any\ n \in \mathcal{Z}.$$

TICKET

System variables: $p(P_1, P_2, A, B, T, S)$.

Initial Condition: $initial \leftarrow T = S, p(think, think, 0, 0, T, S)$.

Transitions:

$$
\begin{array}{lll}
p(think, P, A, B, T, S) & \leftarrow & T >= 0, T_1 = T + 1, \quad p(wait, P, T, B, T_1, S). \\
p(P, think, A, B, T, S) & \leftarrow & T >= 0, T_1 = T + 1, \quad p(P, wait, A, T, T_1, S). \\
p(wait, P, A, B, T, S) & \leftarrow & A =< S, \quad p(use, P, A, B, T, S). \\
p(P, wait, A, B, T, S) & \leftarrow & B =< S, \quad p(P, use, A, B, T, S). \\
p(use, P, A, B, T, S) & \leftarrow & S >= 0, S_1 = S + 1, \quad p(think, P, A, B, T, S_1). \\
p(P, use, A, B, T, S) & \leftarrow & S >= 0, S_1 = S + 1, \quad p(P, think, A, B, T, S_1).
\end{array}
$$

Figure 13: The CLP-program for the ticket algorithm.

Figure 14 shows the resulting approximated fixpoint computed by the prototype. Notice that the condition above holds since the only fact with $p_1 =$

$$
\begin{array}{l}
p(wait, use, a, b, t, s) \leftarrow t - s =< 1.0, a >= 0.0, t >= 0.0, b >= 0.0 \\
p(use, use, a, b, t, s) \leftarrow t - s =< 1.0, t >= 0.0, b >= 0.0, a >= 0.0 \\
p(use, wait, a, b, t, s) \leftarrow b - s =< 0.0, a >= 0.0, b >= 0.0, t >= 0.0 \\
p(wait, wait, a, b, t, s) \leftarrow b >= 0.0, t >= 0.0, a >= 0.0, s - t >= -1.0, s - a >= 0.0 \\
p(wait, think, a, b, t, s) \leftarrow t - s =< 0.0, a >= 0.0, t >= 0.0, b >= 0.0 \\
p(use, think, a, b, t, s) \leftarrow t - s =< 0.0, t >= 0.0, a >= 0.0, b >= 0.0 \\
p(think, use, a, b, t, s) \leftarrow t - s =< 0.0, t >= 0.0, b >= 0.0, a >= 0.0 \\
p(think, think, a, b, t, s) \leftarrow t - s =< -1.0, t >= 0.0, b >= 0.0, a >= 0.0 \\
p(think, wait, a, b, t, s) \leftarrow t - s =< 0.0, b >= 0.0, t >= 0.0, a >= 0.0 \\
p(wait, use, a, b, t, s) \leftarrow a - s =< 0.0, a >= 0.0, b >= 0.0, t >= 0.0 \\
p(wait, wait, a, b, t, s) \leftarrow t >= 0.0, b >= 0.0, a >= 0.0, s - b >= 0.0, s - a >= 0.0 \\
p(wait, wait, a, b, t, s) \leftarrow a >= 0.0, t >= 0.0, b >= 0.0, s - t >= -1.0, s - b >= 0.0 \\
p(use, wait, a, b, t, s) \leftarrow t - s =< 1
\end{array}
$$

Figure 14: Testing mutual exclusion for TICKET.

$p_2 = think$ is $p(think, think, A, B, S, T) \leftarrow T < S - 1, A \geq 0, B \geq 0, S \geq 0$. The initial state is not an instance of the instances of this fact. The execution required 2.8s.

Now, let $\mathcal{W}$ be the assertion $p_1 = wait$ and $\mathcal{C}$ be the assertion $p_1 = use$. Again, by the results Section 4.1, process p_1 is starvation free iff the following condition is true:

$$p(think, think, 0, 0, n, n) \notin [lfp(S_{P'})]_{\mathcal{R}} \quad for\ any\ n \in \mathcal{Z}.$$

where $P' = P \oplus (\mathcal{W} \cap [gfp(S_{P \oslash \neg \mathcal{C}})]_{\mathcal{R}})$. The program P' is depicted in Figure 15. The exact inner greatest fixed point can be computed in 2.7s. Using the least fixpoint computation with approximation, an upper-bound of the outer fixpoint can be computed in 3.5s for a total execution time of 6.3s.

$$
\begin{array}{lcll}
p(think, P, A, B, T, S) & \leftarrow & T >= 0, T_1 = T + 1, & p(wait, P, T, B, T_1, S). \\
p(think, think, A, B, T, S) & \leftarrow & T >= 0, T_1 = T + 1, & p(think, wait, A, T, T_1, S). \\
p(wait, think, A, B, T, S) & \leftarrow & T >= 0, T_1 = T + 1, & p(wait, wait, A, T, T_1, S). \\
p(wait, P, A, B, T, S) & \leftarrow & A =< S, & p(use, P, A, B, T, S). \\
p(think, wait, A, B, T, S) & \leftarrow & B =< S, & p(think, use, A, B, T, S). \\
p(wait, wait, A, B, T, S) & \leftarrow & B =< S, & p(wait, use, A, B, T, S). \\
p(think, use, A, B, T, S) & \leftarrow & S >= 0, S_1 = S + 1, & p(think, think, A, B, T, S_1). \\
p(wait, use, A, B, T, S) & \leftarrow & S >= 0, S_1 = S + 1, & p(wait, think, A, B, T, S_1).
\end{array}
$$

Figure 15: $ticket \oslash \neg(P_1 = use)$.

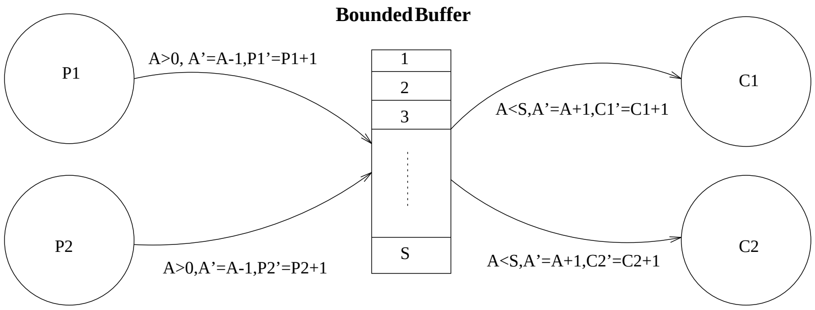


Figure 16: A producer/consumer with bounded buffer.

Producer-consumer algorithm are other interesting examples of concurrent programs. We will discuss next some examples considered in [BGP98].

6.3 Bounded Buffer

Let us first consider a concurrent system of two producers and two consumers connected by a buffer of size s. A variable A will denote the number of *empty* cells in the buffer. The behaviour of one of the procesess is depicted in Fig. 16. The resulting CLP-program BBUFFER given in Figure 17. The first invariant that we want to prove is $AG(p_1 + p_2 - (c_1 + c_2) = s - a)$ which holds iff

$p(A, P_1, P_2, C_1, C_2, S) \leftarrow A = S, P_1 = 0, P_2 = 0, C_1 = 0, C_2 = 0 \notin S_{P \oplus \mathcal{U}}$ where $\mathcal{U}$ is the set:

$$
\begin{array}{l}
p(A, P_1, P_2, C_1, C_2, S) \leftarrow P_1 + P_2 - (C_1 + C_2) + A < S. \\
p(A, P_1, P_2, C_1, C_2, S) \leftarrow P_1 + P_2 - (C_1 + C_2) + A > S.
\end{array}
$$

The property can be proved by our model checker in 0.2s. Another safety condition is given by

$$AG(0 \leq p_1 + p_2 - (c_1 + c_2) \leq s).$$

Using the previous invariant we can write the safety property as $AG(0 \leq a \leq s)$. Since we are interested in the integer-solutions of the problem, it is easy

BBUFFER
System variables: $p(A, P_1, P_2, C_1, C_2, S)$.
Initial Condition:
$initial \leftarrow p(A, P_1, P_2, C_1, C_2, S), A = S, P_1 = P_2 = C_1 = C_2 = 0, S >= 1.$
Transitions:

$$
\begin{array}{lcl}
p(A, P_1, P_2, C_1, C_2, S) & \leftarrow & A > 0, A' = A - 1, P_1' = P_1 + 1, p(A', P_1', P_2, C_1, C_2, S). \\
p(A, P_1, P_2, C_1, C_2, S) & \leftarrow & A > 0, A' = A - 1, P_2' = P_2 + 1, p(A', P_1, P_2', C_1, C_2, S). \\
p(A, P_1, P_2, C_1, C_2, S) & \leftarrow & A < S, A' = A + 1, C_1' = C_1 + 1, p(A', P_1, P_2, C_1', C_2, S). \\
p(A, P_1, P_2, C_1, C_2, S) & \leftarrow & A < S, A' = A + 1, C_2' = C_2 + 1, p(A', P_1, P_2, C_1, C_2', S).
\end{array}
$$

Figure 17: CLP program for producer/consumer with bounded buffer.

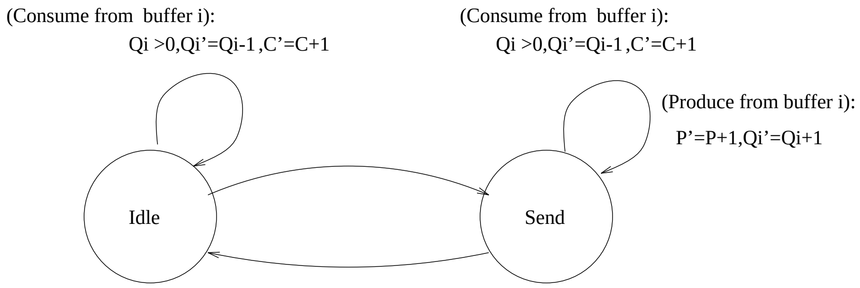


Figure 18: A producer/consumer with unbounded buffer.

to see that the above program is safe iff the simplified program below is safe:

$$
\begin{array}{lcl}
p(A, S) & \longleftarrow & A \geq 1, A' = A - 1, p(A', S). \\
p(A, S) & \longleftarrow & A \leq (S - 1), A' = A + 1, p(A', S).
\end{array}
$$

The unsafe states can be characterized as $\mathcal{U} = p(A, S) \leftarrow A \leq -1, p(A, S) \leftarrow A \geq (S + 1)$. This makes the fixpoint computation converge in one step to $\mathcal{U}$ thus proving the safety condition.

6.4 Unbounded Buffer

Let us consider now a system with one producer and one consumer connected by two unbounded buffers 18. The system can be represented by an automaton with two states: *idle*, in which only the consumer is active (to weaken the producer), and *send*, in which both processes are active. The index Q_i keeps track of the number of *nonempty* cells in the buffer i. The corresponding CLP-program UBUFFER is given in Figure 19. We can prove the invariant $P \geq C$ by proving that $P = C + Q_1 + Q_2, Q_1 \geq 0, Q_2 \geq 0$. Thus, we set the unsafe states $\mathcal{U}$ to be

$$
\begin{array}{l}
p(\_, P, Q_1, Q_2, C) \leftarrow P > C + Q_1 + Q_2, Q_1 >= 0, Q_2 >= 0. \\
p(\_, P, Q_1, Q_2, C) \leftarrow P < C + Q_1 + Q_2, Q_1 >= 0, Q_2 >= 0.
\end{array}
$$

UBUFFER

System variables: $p(A, P_1, P_2, C_1, C_2, S)$.

Initial Condition: $p(\_, P, Q_1, Q_2, C) \leftarrow P = C = Q_1 = Q_2 = 0.$

Transitions:

$$
\begin{array}{lll}
p(idle, P, Q_1, Q_2, C) & \leftarrow & p(send, P, Q_1, Q_2, C). \\
p(send, P, Q_1, Q_2, C) & \leftarrow & p(idle, P, Q_1, Q_2, C). \\
p(send, P, Q_1, Q_2, C) & \leftarrow & P_1 = P + 1, Q_1^1 = Q_1 + 1, p(send, P_1, Q_1^1, Q_2, C). \\
p(send, P, Q_1, Q_2, C) & \leftarrow & P_1 = P + 1, Q_2^1 = Q_2 + 1, p(send, P_1, Q_1, Q_2^1, C). \\
p(S, P, Q_1, Q_2, C) & \leftarrow & Q_1 > 0, Q_1^1 = Q_1 - 1, C_1 = C + 1, p(S, P, Q_1^1, Q_2, C_1). \\
p(S, P, Q_1, Q_2, C) & \leftarrow & Q_2 > 0, Q_2^1 = Q_2 - 1, C_1 = C + 1, p(S, P, Q_1, Q_2^1, C_1).
\end{array}
$$

Figure 19: CLP-program for producer/consumer with unbounded buffer.

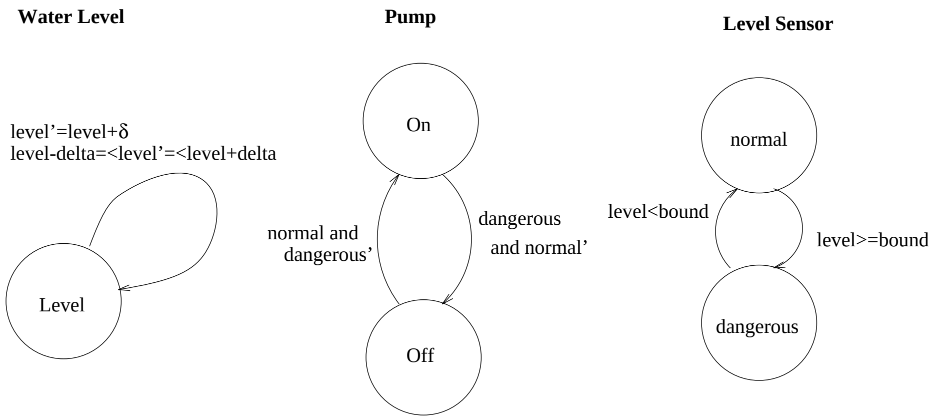


Figure 20: Components of a water level controler.

Applying the approximation used for the ticket algorithm the computation converges in 3 steps. The initial state is not part of the resulting fixpoint.

6.5 Water Level Controler

In many real examples the control location and (part of the data variables) can be expressed by *boolean* variables. Introducing an explicit type of constraints for this type of variables allows to have more concise representation of the transition rules. Let us consider the following example. We want to specify the behaviour of a *water-level controler*. The value of a *Level* sensor is monitored from time to time. Whenever the Level is higher than a fixed *Bound* a *Pump* is activated in order to restore a safe state. Such a system can be represented as the concurrent system obtained by composing the automata in Figure 20, in which a special program non-deterministically modify the level of the water.

We can represent the *control* part of the system by using boolean variables in combination with *arithmetic* constraints to monitor the sensors. Let *Level, Bound* and *Delta* be real variables, where *Delta* is an upper bound to the possible variation of the level (whose value depends on the speed of the incoming water). Furthermore, let *Normal, Dangerous* and *Pump* be boolean variables. Let us consider boolean expressions built over $*$ (and), $+$ (or), $\neg$ (not), etc. Then we can specify the considered system as the program WATER LEVEL CONTROLER given in Appendix B. Note that *Bound* and *Delta* are free variables, i.e., the representation is parametric over the dangerous level and over the maximum variation of the water.

By hypothesis, arithmethic and boolean constraints share no variables. For such an example we can prove properties such as $AG(Level > Bound \rightarrow Pump)$, i.e., involving both arithmethic and boolean constraints.

6.6 Mutual Exclusion

Another solution to the mutual exclusion problem is given (in the 2-process case) by the algorithm MUT-AST [LS97] shown in Appendix B.

State 1 is the initial state, while state 6 corresponds to the critical section. Each process enters in the critical section when $y = 0$. The safety property for the 2-case $(p(6, 6, \_, \_, \_) \not\in [lfp(S_P)]_{\mathcal{D}})$ can be proved in 0.2s without approximation.

More interesting, by applying the abstraction technique shown in [LS97] it is possible to define a transition system (program NETWORK in Appendix B) for a parameterized network of MUT-AST-processes. This technique attempts to avoid the state-explosion problem which occurs in the composition of several processes. The resulting system (CLP-program) has a state with 14 variables which represent the number of processes in a given state. Variable n_1 keeps track of the number of process which are in their initial states (initially n_1 is equal to the number of processes of the system). The variable n_8 keeps track of the number of processes which are inside the critical section. The safety property for the network is proved if $n_8 \leq 1$. Our prototype proved the safety condition in 0.6s using the approximation.

7 Other Strategies

A number of techniques used in bottom-up query evaluation of deductive database languages with constraints can be exploited to make the analysis efficient. In particular, it is interesting to analyze the result of the *magic templates algorithm* [RSS92] for the CLP-programs we considered in our setting. Let us restrict to the verification of safety properties. The programs have the simplified form $p(\bar{t}) \leftarrow c, p(\bar{s})$ where c is a constraint. The only facts correspond to the set of unsafe states $\mathcal{U}$ added in the backward analysis. The analysis is aimed at proving that the initial states are not instances of

$p(think, P, A, B, T, S) \leftarrow T_1 = T + 1, p'(think, P, A, B, T, S), p(wait, P, T, B, T_1, S).$
$p(P, think, A, B, T, S) \leftarrow T_1 = T + 1, p'(P, think, A, B, T, S), p(P, wait, A, T, T_1, S).$
$p(wait, P, A, B, T, S) \leftarrow A =< S, p'(wait, P, A, B, T, S), p(use, P, A, B, T, S).$
$p(P, wait, A, B, T, S) \leftarrow B =< S, p'(P, wait, A, B, T, S), p(P, use, A, B, T, S).$
$p(use, P, A, B, T, S) \leftarrow S_1 = S + 1, p'(use, P, A, B, T, S), p(think, P, A, B, T, S_1).$
$p(P, use, A, B, T, S) \leftarrow S_1 = S + 1, p'(P, use, A, B, T, S), p(P, think, A, B, T, S_1).$
$p(use, use, A, B, T, S) \leftarrow p'(use, use, A, B, T, S).$
$p'(wait, P, T, B, T_1, S) \leftarrow T_1 = T + 1, p'(think, P, A, B, T, S).$
$p'(P, wait, A, T, T_1, S) \leftarrow T_1 = T + 1, p'(P, think, A, B, T, S.$
$p'(use, P, A, B, T, S) \leftarrow A =< S, p'(wait, P, A, B, T, S).$
$p'(P, use, A, B, T, S) \leftarrow B =< S, p'(P, wait, A, B, T, S).$
$p'(think, P, A, B, T, S_1) \leftarrow S_1 = S + 1, p'(use, P, A, B, T, S).$
$p'(P, think, A, B, T, S_1) \leftarrow S_1 = S + 1, p'(P, use, A, B, T, S).$
$p'(think, think, A, B, T, S) \leftarrow A = B, T = S.$

Figure 21: Program transformed by magic templates.

the resulting fixed point. The transformation based on magic templates for this class of programs is defined as follows.

Proposition 7.1 (Magic set for concurrent systems) *Let P be the encoding of the concurrent system $\mathcal{S} = \langle V, \Theta, \mathcal{E} \rangle$ s.t. $p(\tilde{x}) \rightarrow c_0$ represents the initial state Θ. The program P' is obtained as follows:* for each rule $p(\tilde{t}) \leftarrow c, p(\tilde{s})$ in P, add the modified rule $p(\tilde{t}) \leftarrow c, p'(\tilde{t}), p(\tilde{s})$, and add the rule $p'(\tilde{s}) \leftarrow c, p'(\tilde{t})$. *Finally, add $p'(\tilde{x}) \leftarrow c_0$.*

Proposition 7.2 (Soundness [RSS92]) *The programs P' is equivalent to P w.r.t. the set of answers to the query ?-$p(\tilde{x}) \wedge c_0$.*

Corollary 7.1 (Magic sets and model checking) *Let P and P' be the programs defined above. Then,*

$$lfp(T_P) \cap \Theta = lfp(T_{P'}) \cap \Theta = [lfp(S_{P'})]_{\mathcal{D}} \cap \Theta.$$

The magic templates transformation is aimed at merging bottom-up and top-down analysis by filtering the immediate consequence computation of the original clauses with the *magic* goals added to their bodies. In our restricted setting, the bottom-up analysis of the transformed program corresponds to the bottom-up analysis of the original program filtered by a preliminary *reachability* analysis. To illustrate this, let us consider the magic templates transformation of the CLP-program for the *ticket* algorithm shown in Fig. 21. Note that the link connecting the primed and the non-primed predicate definitions is given by the clause

$$p(use, use, A, B, T, S) \leftarrow p'(use, use, A, B, T, S).$$

Thus, non-primed facts are produced if and only if the unsafe state is inferred by the forward analysis on the primed clauses. In the special case

$$
\begin{array}{llcll}
1- & p(think,P,A,B) & \leftarrow & A_1 = B+1, B \geq 0, & p(wait,P,A_1,B). \\
2- & p(wait,P,A,B) & \leftarrow & A < B, A \geq 0, & p(use,P,A,B). \\
3- & p(wait,P,A,B) & \leftarrow & B = 0, A \geq 0, & p(use,P,A,B). \\
4- & p(use,P,A,B) & \leftarrow & B \geq 0, A_1 = 0, & p(think,P,A_1,B). \\
5- & p(P,think,A,B) & \leftarrow & B_1 = A+1, A \geq 0, & p(P,wait,A,B). \\
6- & p(P,wait,A,B) & \leftarrow & B < A, B \geq 0, & p(P,use,P,A,B). \\
7- & p(P,wait,A,B) & \leftarrow & A = 0, B \geq 0, & p(P,use,P,A,B). \\
8- & p(P,use,A,B) & \leftarrow & A \geq 0, B_1 = 0, & p(P,think,A,B_1).
\end{array}
$$

Figure 22: Introducing typos in the bakery algorithm.

of the ticket algorithm, the fixpoint computation on the program in Fig. 21 terminates without producing non-primed facts (i.e. the unsafe state is not-reachable) in 0.6s. However, in general the forward analysis can be non-terminating as well as the backward analysis. The previous transformed scheme can be used to mix reachability and backward analysis for instance by restricting the approximation to p'-predicates only so as to compute an upper bound for the set of reachable states.

8 Error Tracing

When the verification procedure cannot prove the safety of a system, it is possible to further analyze the output of the fixpoint computation (i.e. the least fixpoint of the program $P \oplus \mathcal{U}$ where $\mathcal{U}$ represent the unsafe states) trying to reconstruct the possible causes of the failure. Let us consider the modified version of the bakery algorithm in Fig. 8. We first insert some *auxiliary* information (e.g. a pointer to the clause and the fact used to produce a new fact) in the data stored during the fixpoint computation. The model-checker terminates the analysis detecting an occurence of the initial state $p(think,think,0,0)$ in the resulting fixed point. However, we can now trace all the paths from the initial state to an unsafe state which occur in the fixed point. In this case, we obtain the following erroneous paths (where $A \leadsto_i B$ denotes that the fact A has been produced by fact B using clause i):

a. $p(think,think,A,0), A \geq 0 \;\leadsto_1\; p(wait,think,A,0), A > 0$
$\leadsto_3\; p(use,think,A,B), A > B, B \geq 0 \;\leadsto_5\; p(use,wait,A,B), A > B$
$\leadsto_6\; p(use,use,A,B)$

b. $p(think,think,0,0) \;\leadsto_5\; p(think,wait,0,0) \;\leadsto_7\; p(think,use,A,0), A \geq 0$
$\leadsto_1 p(wait,use,A,0) \;\leadsto_3\; p(use,use,A,B)$

c. $p(think,think,A,B), A \geq 0, B \geq 0 \;\leadsto_1\; p(wait,think,A,B), A \geq B, B \geq 0$
$\leadsto_5\; p(wait,wait,A,B), A > B, B \geq 0 \;\leadsto_6\; p(wait,use,A,B), A > 0, B \geq 0$
$\leadsto_8\; p(wait,think,A,0), A > 0 \;\leadsto_3\; p(use,think,A,B), A > B, B \geq 0$
$\leadsto_5\; p(use,wait,A,B), A > B \;\leadsto_6\; p(use,use,A,B)$

The three paths share a common malicious behaviour. When the process p_2 switch from *thinking* to *waiting* (clause 5) it does not update its priority. In this way either process p_2 can enter the critical section when p_2 is already inside it (trajectory a. and b.) or p_1 can enter the critical section when p_2 is already inside it since the priority of p_2 remains set to zero (trajectory c.). Using this information a closer look at the suspect clause 5 of the program in Fig. 8 reveals that the variable B_1 does not occur in the call of the predicate p in its body.

9 Conclusions

In the present paper we have shown how CLP can be useful interpreted as a model for specification and verification of concurrent systems. First of all logical variables are useful in constraint formulae describing infinite sets of states. Furthermore, the bottom-up evaluation of CLP programs with large sets of facts has been extensively studied in the context of Constraint Data Bases (e.g. the magic set transformation [RSS92]), also in terms of implementation technology, although the intended applications have been quite different up to now.

Finally, we find that our implementation is most *simple and natural.* This is due to the fact that the three languages for, respectively, implementing the method, specifying the concurrent systems and expressing properties of states, operate on the same level of abstraction (namely, the mathematical meaning of constraints). We believe that the high *adaptability* to specific verification problems has become apparent. This adaptability may be traced back to the previous argument about the abstraction level as well as to the general flexibility of CLP programming. Also, note that we can prove all of our optimizations correct. Regarding *efficiency*, we give a table of the execution times for the verification problems in our case studies in Figure 23. We find the fact that the CLP implementation performs so well on these examples impressive. We explain it by the homogeneity of the implementation (e.g., no conversions between different constraint encodings are needed) and the fact that constraint handling and data base functionality are, as integral parts of the CLP paradigm, already present in an optimized form.

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model checking for real-time systems. In *Proc. Fifth IEEE Symp. on Logic in Computer Science, Philadelphia*, pages 414–425, 1990.

[And91] G. R. Andrews. *Concurrent programming: principles and practice.* Addison-Wesley, Menlo Park, CA, 1991.

SAFETY	C	ET	EN	ERT	ERN	AT	AN	ART	ARN
BAKERY	8	0.3s [2.85s]	18	0.4s	16	-	-	-	-
TICKET	6	/	/	/	/	2.5s [7.32s]	15	2.6s	13
MUT-AST	20	0.2s	20	0.2s	20	-	-	-	-
NETWORK	16	/	/	/	/	0.9s	3	1.5s	3
BBUFFER (1)	4	0.5s	2	0.5s	2	-	-	-	-
BBUFFER (2)	4	0.01s	2	0.01s	2	-	-	-	-
UBUFFER	6	/	/	/	/	5.7s [1.37s]	13	3.7s	6

LIVENESS	ET	AT
BAKERY	1.8s [7.64s]	-
TICKET	/	5.8s [27.03s]

Figure 23: Benchmarks: The programs have been executed on a SUN-Sparc Station 4: C=number of clauses, E=exact, A=with approximation, R=with elimination of redundant facts, T=execution time, N=number of produced facts, /=non-termination, $-$=not tested. We indicate within brackets the timings obtained in [BGP98] on a SUN-Sparc Station 5 for the verification of the same properties.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In John C. Mitchell, editor, *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society Press.

[BGL98] T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: A composite approach. In *Proceedings of the 1998 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98). Technical Report CS-TR-3822, UMIACS-TR-97-62. Department of Computer Science, University of Maryland, College Park, August 1997*, pages 113–123, March 1998.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetics. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, LNCS 1254, pages 400–411. Springer, Haifa, Israel, July 1997.

[BGP98] T. Bultan, R. Gerber, and W. Pugh. Model Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations and Experimental Results, February 1998.

[CABN97] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, LNCS 1254, pages 316–327. Springer, Haifa, Israel, July 1997.

[CGL+94] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. The concurrency factory – practical tools for specification, simulation, verification, and implementation of concurrent systems. In *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms, Princeton, NJ*, May 1994.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1978.

[CP98] W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In Bernhard Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, March-April 1998. To appear.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers (North-Holland), 1990.

[FR96] L. Fribourg and J. Richardson. Symbolic verification with gap-order constraints, 1996. Technical Report LIENS-93-3, Laboratoire d'Informatique, Ecole Normale Superieure, Paris.

[Fri98] L. Fribourg. A closed-form evaluation for extended timed automata, March 1998. Technical Report LSV-98-2, Laboratoire Spécification et Vérification, Ecole Normale Supérieure de Cachan.

[FS98] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere!, April 1998. Technical Report LSV-98-4, Laboratoire Spécification et Vérification, Ecole Normale Supérieure de Cachan.

[GDL95] M. Gabbrielli, M.G. Dore, and G. Levi. Observable semantics for constraint logic programs. *Journal of Logic and Computation*, 2(5):133–171, 1995.

[Gro96] The VIS Group. Vis: A system for verification and synthesis. In R. Alur and T. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 428–432. Springer, New Brunswick, NJ, July 1996.

[HH95] T. A. Henzinger and P.-H. Ho. A note on abstract-interpretation strategies for hybrid automata. In P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, editors, *Hybrid Systems II*, Lecture Notes in Computer Science 999, pages 252–264. Springer-Verlag, 1995.

[HHWT97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. In O. Grumberg, editor, *CAV 97: Computer-aided Verification*, Lecture Notes in Computer Science 1254, pages 460–463. Springer-Verlag, 1997.

[Hol90] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, November 1990.

[Hol95] C. Holzbaur. OFAI clp(Q,R) Manual, Edition 1.3.3, 1995.

[HPR96] N. Halbwachs, Y-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis, 1996. Technical Report, Verimag, Gieres, Paris.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19/20:503–582, May-July 1994.

[LS97] D. Lesens and H. Saidi. Automatic verification of parameterized networks of processes by abstraction, July 1997. Proceedings of the International Workshop on Verification Infinite State Systems (INFINITY 97), Bologna.

[McM93] K. McMillan. *Symbolic Model Checking.* Kluwer, Boston;Dordrecht;London, 1993.

[Mel97] S. Melzer. Verification of parallel systems using constraint programming. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming - CP97*, volume 1330 of *LNCS*, pages 92–106. Springer-Verlag, October 1997.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag New York, 1995.

[MR89] M. J. Maher and R. Ramakrishnan. Déjà vu in fixpoints of logic programs. In Ross A. Lusk, Ewing L.; Overbeek, editor, *Proceedings of the North American Conference on Logic Programming (NACLP '89)*, pages 963–980, Cleveland, Ohio, October 1989. MIT Press.

[Rev93] P. Z. Revesz. A closed-form evaluation for Datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116(1):117–149, 1993.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, LNCS 1254. Springer, Haifa, Israel, July 1997.

[RSS92] R. Ramakrishnan, D. Srivistava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. *The State of the Art in Computer Systems and Software Engineering*, 1992.

[Sha93] U. A. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, Sept 1993.

[Sri92] D. Srivistava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. In *2nd International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale*, 1992.

[Urb96] L. Urbina. Analysis of hybrid systems in clp(r). In *Principles and Practice of Constraint Programming, CP96*, Lectures Notes in Computer Science 1118, pages 451–467. Springer-Verlag, 1996.

A SICStus Prolog code

```
%% dynamic predicates

:- multifile s/3.
:- multifile r/3.
:- multifile rd/3.
:- multifile w/2.
:-dynamic s/3.
:-dynamic r/3.
:-dynamic rd/3.
:-dynamic w/2.
:-dynamic cf/1.
:-dynamic red/0.
:-use_module(library(terms)).
:-use_module(library(lists)).
:-use_module(library(clpr)).

% CLP- Utility Library

% entails(C,D): C entailes D
%  - C,D are constraints of the form {c1,...,cn}

entails(C,D):-
 C,entailed(D).

% factsubsumed(H,D,H1,D1): (H1,D1) subsumes (H,D)
%   - H,H1 are atoms
%   - D,D1 are constraints

factsubsumed(H,D,H1,D1):-
 subsumes_chk(H1,H),
 unify_with_occurs_check(H,H1),
 subsumed(D,D1).

subsumed({},{}).

subsumed({C},{D}):-
 call_residue(entails({C},(D)),_).

subsumed({C},{}):-
 call_residue({C},_).

get_constraints([[_]-{C}|R],{C,N}):-
```

```
 get_constraints(R,{N}),!.

get_constraints([[_]-{C}|R],{C}):-
 get_constraints(R,{}),!.

get_constraints([],{}):-!.

get_constraints1([],{}):-!.

get_constraints1([[_]-{C}],{C}):-!.

get_constraints1([[_]-{C}|R],{C,N}):-
 get_constraints1(R,{N}),!.

% unify(C,D,R):  R is equivalent to (C and D)

unify({},{},{}):-!.

unify({},C,R):-
 call_residue(C,S),
 get_constraints(S,R).

unify({D},{},R):-
 call_residue({D},S),
 get_constraints(S,R).

unify({D},{C},R):-
 call_residue({C,D},S),
 get_constraints(S,R).

% project(C,T,PC):
%   PC is the projection of C over the variables of T

project({}, _, {}) :- !.

project(Cons, Term, PCons) :-
    !,
    retractall(projection_temporary(_)),
    assert((projection_temporary(Term) :- Cons)),
    call_residue(projection_temporary(Term), TmpPCons),
    get_constraints(TmpPCons, PCons).

% satisfiable(C,D):  C and D is satisfiable
```

```
satisfiable({},{}).

satisfiable({},{D}):-
 call_residue({D},_).

satisfiable({D},{}):-
 call_residue({D},_).

satisfiable({D},{C}):-
 call_residue({D},_),
 call_residue({C},_).

% simplify(C,D).

simplify({},{}):-!.

simplify(C,D):-
 call_residue(C,R),
 get_constraints(R,D).

% approx(C,AC)  AC is an upper bound for C

approx({C},{C1,_}):-
 call_residue(entails({C1},C),_),
 \+(call_residue(entails({C},C1),_)),!.

approx({C},{_,D}):-!,
 approx({C},{D}).

approx({C},{C1}):-
 call_residue(entails({C1},C),_),
 \+(call_residue(entails({C},(C1)),_)).

% upper(C,D,OC,AOC): uses C to relax OC into AOC

upper({C,T},D,{Co,To},Ca):-
 approx({C},D),!,
 upper({T},D,{To},Ca).

upper({C,T},D,{Co,To},Ca):-!,
 upper({T},D,{To},Ca1),
 combine({Co},Ca1,Ca).

upper({C},D,{Co},{}):-
```

```
 approx({C},D),!.

upper({C},_,{Co},{Co}).

combine({C},{},{C}).

combine({C},{D},{C,D}).

% time(G,T,M,P,C) T is the execution time for G,

time(Goal,T,M,P,C):-
        statistics(runtime,[T1,_]),
        Goal,
        statistics(runtime,[T2,_]),
        statistics(memory,[M,_]),
        statistics(program,[P,_]),
        statistics(choice,[C,_]),
        T is ((T2-T1)//100)/10.
       % T is T2-T1.

% clear: to clear the DB

clear:- clearR,clearS,clearW.

clearS:-
 retractall(s(_,_,_)).

clearR:-
 retractall(r(_,_,_)).

clearW:-
 retractall(w(_,_)).

% turn_zero(I): erases all the facts s(J,A,B) with J<I
%               and turns the facts s(I,A,B) into s(0,A,B)

turn_zero(I):-
 s(K,A,B),
 K<I,
 retract(s(K,A,B)),
 fail.

turn_zero(I):-
 retract(s(I,A,B)),
```

```
 assert(s(0,A,B)),
 fail.

turn_zero(_).

% select_state(A): erase all the states s(K,B,C) s.t. A=\=B

select_states(A,D):-
 retract(s(K,B,C)),
 atom_cons_unify(B,A,B,D,C,E),
 assert(s(K,B,E)),
 fail.

select_states(_,_).

% count: counts the number of facts of the form s(A,B,C)

count:-
 assert(cf(0)),
 s(_,_,_),
 retract(cf(I)),
 J is I+1,
 assert(cf(J)),
 fail.

count:-
 cf(I),
 nl,
 write('Number of facts:'),write(I),
 retract(cf(I)).

% onR,offR: to activate/disactivate the
%  elimination of redundant atoms

onR:-assert(red).
offR:-retract(red).

approximate(I,H,C,D):-
 s(J,H1,D1),
 J<I,
 copy_term(t(H,C),t(Hc,Cc)),
 unify_with_occurs_check(Hc,H1),
 upper(Cc,D1,C,D),
 satisfiable(Cc,D1).
```

```
approximate(_,_,C,C).

atom_cons_unify(H,B,B1,C,C1,D2):-
 unify_with_occurs_check(B1,B),
 unify(C,C1,D),
 project(D,H,D1),
 simplify(D1,D2).

is_not_subsumed(H,D1):-
 \+( (s(_,H1,D3),factsubsumed(H,D1,H1,D3)) ).

is_not_subsumed(J,H,D1):-
 \+( (s(J,H1,D3),factsubsumed(H,D1,H1,D3)) ).

reduce(J,H,D1):-
  s(J,H1,D3),
  factsubsumed(H1,D3,H,D1),
  retract(s(J,H1,D3)),
  fail.

reduce(_,_,_).

%%  MAIN ALGORITHMS

% Exact Tp (Upward)

ucollect(I,J):-
 r(H,B,C),
 s(I,B1,C1),
 atom_cons_unify(H,B,B1,C,C1,D1),
 is_not_subsumed(H,D1),   % forall K<=J
 (red -> reduce(_,H,D1);true),
 assert(s(J,H,D1)).

% Approximated Tp (Upward)

acollect(I,J):-
 r(H,B,C),
 s(I,B1,C1),
 atom_cons_unify(H,B,B1,C,C1,D1),
% newapproximate(I,H,D1,D2),
 is_not_subsumed(H,D1),
 approximate(I,H,D1,D2),
```

```
 is_not_subsumed(H,D2),  % forall K<=J
 (red -> reduce(_,H,D2);true),
%is_not_subsumed(J,H,D1),
 assert(s(J,H,D2)).

% Exact Tp (Downward)

dcollect(I,J):-
 rd(H,B,C),
 s(I,B1,C1),
 atom_cons_unify(H,B,B1,C,C1,D1),
 is_not_subsumed(J,H,D1),  % only for J
% reduce(J,H,D1),
 assert(s(J,H,D1)).

%% Downward Iterations

testj(J,A,C):-
  s(J,B,D),
  factsubsumed(A,C,B,D),!,
  fail.

testj(_,_,_).

test(I,J,K):-
 s(I,A,C),
 testj(J,A,C),
 gfixpoint(J,K).

test(_,J,J).

gfixpoint(I,K):-
 J is I+1,
 setof(_,dcollect(I,J),_),
 !,
 test(I,J,K).

gfixpoint(I,I).

%% Approximated Upward Iterations

afixpoint(I):-
 J is I+1,
 setof(_,acollect(I,J),_),
```

```
 !,
 afixpoint(J).

afixpoint(_).

%% Exact  Upward Iterations

fixpoint(I):-
 J is I+1,
 setof(_,ucollect(I,J),_),
 !,
 fixpoint(J).

fixpoint(_).

gfp(Program):-
 clear,
 [Program],
 time(gfixpoint(0),T,M,P,C),
 write('Fix Point Reached'),
 listing(s/3),
 nl,
 write('Execution Time: '), write(T),
 write('Memory:         '), write(M),
 write('Program Memory: '), write(P),
 write('Choice Points:  '), write(C),
 count.

lfp(Program,Mode):-  %Mode: a (approx.) / n (normal)
 clear,
 [Program],
 (Mode=n -> time(fixpoint(0),T,M,P,C);
  (Mode=a -> time(afixpoint(0),T,M,P,C);
     time(wfixpoint(0),T,M,P,C)
  )
 ),
 write('Fix Point Reached'),
 listing(s/3),
 nl,
 write('Execution Time: '), write(T),nl,
 write('Memory:         '), write(M),nl,
 write('Program Memory: '), write(P),nl,
 write('Choice Points:  '), write(C),nl,
 count.
```

```
% Testing Starvation

st:-
 gfixpoint(0,K),
 listing(s/3),nl,
 turn_zero(K),
 w(A,C),
 select_states(A,C),
 afixpoint(0).

sf(Prog):-
 clear,
 [Prog],
 time(st,T,M,P,C),
 nl,
 listing(s/3),nl,
 write('Execution Time: '), write(T),
 count.
```

B List of examples

MUT-AST

System variables: $p(P1, P2, T1, T2, Y)$.

Initial Condition: $initial \leftarrow p(P1, P2, T1, T2, Y), P1 = 1, P2 = 1, Y = 1.$

Transitions:

$p(1, P, T1, T2, Y) \leftarrow T1 = -1, p(2, P, T1, T2, Y).$
$p(2, P, T1, T2, Y) \leftarrow T1 < 0, p(3, P, T1, T2, Y).$
$p(2, P, T1, T2, Y) \leftarrow T1 > 0, p(1, P, T1, T2, Y).$
$p(3, P, T1, T2, Y) \leftarrow Y > 0, p(4, P, T1, T2, Y).$
$p(4, P, T1, T2, Y) \leftarrow T11 = Y - 1, Y1 = Y - 1, p(5, P, T11, T2, Y1).$
$p(5, P, T1, T2, Y) \leftarrow T1 = 0, p(6, P, T1, T2, Y).$
$p(6, P, T1, T2, Y) \leftarrow p(7, P, T1, T2, Y).$
$p(5, P, T1, T2, Y) \leftarrow T1 > 0, p(7, P, T1, T2, Y).$
$p(5, P, T1, T2, Y) \leftarrow T1 < 0, p(7, P, T1, T2, Y).$
$p(7, P, T1, T2, Y) \leftarrow Y1 = Y + 1, p(7, P, T1, T2, Y1).$
$p(P, 1, T1, T2, Y) \leftarrow T2 = -1, p(P, 2, T1, T2, Y).$
$p(P, 2, T1, T2, Y) \leftarrow T2 < 0, p(P, 3, T1, T2, Y).$
$p(P, 2, T1, T2, Y) \leftarrow T2 > 0, p(P, 1, T1, T2, Y).$
$p(P, 3, T1, T2, Y) \leftarrow Y > 0, p(P, 4, T1, T2, Y).$
$p(P, 4, T1, T2, Y) \leftarrow T21 = Y - 1, Y1 = Y - 1, p(P, 5, T1, T21, Y1).$
$p(P, 5, T1, T2, Y) \leftarrow T2 = 0, p(P, 6, T1, T2, Y).$
$p(P, 6, T1, T2, Y) \leftarrow p(P, 7, T1, T2, Y).$
$p(P, 5, T1, T2, Y) \leftarrow T2 > 0, p(P, 7, T1, T2, Y).$
$p(P, 5, T1, T2, Y) \leftarrow T2 < 0, p(P, 7, T1, T2, Y).$
$p(P, 7, T1, T2, Y) \leftarrow Y1 = Y + 1, p(P, 7, T1, T2, Y1).$

WATER LEVEL CONTROLER

System variables:

$LevelNormal, Bound, Delta : real,$
$Normal, Dangerous, Pump : boolean.$

Initial Condition:

$$initial \leftarrow Level < Bound \wedge Normal \wedge \neg Pump$$
$$p(Level, Normal, Dangerous, Pump, Bound, Delta).$$

Transitions:

$$\begin{array}{l} p(Level, Normal, Dangerous, Pump, Bound, Delta) \leftarrow \\ \quad Level < Bound, Level1 > Bound, Level1 \leq Level + Delta, \\ \quad Level1 \geq Level - Delta, \\ \quad Normal * \neg Normal1 * \neg Dangerous * Dangerous1 * \\ \quad ((Pump * Dangerous) + (\neg Pump * Normal)) * \\ \quad ((Pump1 * Dangerous1) + (\neg Pump1 * Normal1)), \\ \quad p(Level1, Normal1, Dangerous1, Pump1, Bound, Delta). \end{array}$$

$$\begin{array}{l} p(Level, Normal, Dangerous, Pump, Bound, Delta) \leftarrow \\ \quad Level > Bound, Level1 \leq Bound, Level1 \leq Level + Delta, \\ \quad Level1 \geq Level - Delta, \\ \quad \neg Normal * Normal1 * Dangerous * \neg Dangerous1 * \\ \quad ((Pump * Dangerous) + (\neg Pump * Normal)) * \\ \quad ((Pump1 * Dangerous1) + (\neg Pump1 * Normal1)), \\ \quad p(Level1, Normal1, Dangerous1, Pump1, Bound, Delta). \end{array}$$

$$\begin{array}{l} p(Level, Normal, Dangerous, Pump, Bound, Delta) \leftarrow \\ \quad Level > Bound, Level1 > Bound, Level1 \leq Level + Delta, \\ \quad Level1 \geq Level - Delta, \\ \quad (Normal =:= Normal1) * (Dangerous =:= Dangerous1) * \\ \quad ((Pump * Dangerous) + (\neg Pump * Normal)) * \\ \quad ((Pump1 * Dangerous1) + (\neg Pump1 * Normal1)), \\ \quad p(Level1, Normal1, Dangerous1, Pump1, Bound, Delta). \end{array}$$

$$\begin{array}{l} p(Level, Normal, Dangerous, Pump, Bound, Delta) \leftarrow \\ \quad Level \leq Bound, Level1 \leq Bound, Level1 \leq Level + Delta, \\ \quad Level1 \geq Level - Delta, \\ \quad (Normal =:= Normal1) * (Dangerous =:= Dangerous1) * \\ \quad ((Pump * Dangerous) + (\neg Pump * Normal)) * \\ \quad ((Pump1 * Dangerous1) + (\neg Pump1 * Normal1)), \\ \quad p(Level1, Normal1, Dangerous1, Pump1, Bound, Delta). \end{array}$$

NETWORK **System variables:** $N1, \ldots, N14 : int.$

Initial Condition:

$initial \leftarrow N1 = Max, N2 = \ldots N14 = 0, Y = 1, p(N1, \ldots, N14, Y).$

Transitions:

$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N1 >= 1, N1p = N1 - 1, N21 = N2 + 1,$
$p(N1p, N21, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N2 >= 1, Y > 0, N21 = N2 - 1, N31 = N3 + 1,$
$p(N1, N21, N31, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N3 >= 1, Y > 1, N31 = N3 - 1, N41 = N4 + 1, Y = Y - 1,$
$p(N1, N2, N31, N41, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N3 >= 1, Y = 1, N31 = N3 - 1, N51 = N5 + 1, Y = Y - 1,$
$p(N1, N2, N31, N4, N51, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N3 >= 1, Y < 1, N31 = N3 - 1, N61 = N6 + 1, Y = Y - 1,$
$p(N1, N2, N31, N4, N5, N61, N7, N8, N9, N10, N11, N12, N13, N14, Y)$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N4 >= 1, Y < 1, N41 = N4 - 1, N121 = N12 + 1,$
$p(N1, N2, N3, N41, N5, N6, N7, N8, N9, N10, N11, N121, N13, N14, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N5 >= 1, N51 = N5 - 1, N81 = N8 + 1,$
$p(N1, N2, N3, N4, N51, N61, N7, N81, N9, N10, N11, N12, N13, N14, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N6 >= 1, N61 = N6 - 1, N71 = N7 + 1,$
$p(N1, N2, N3, N4, N5, N61, N71, N8, N9, N10, N11, N12, N13, N14, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N7 >= 1, N71 = N7 - 1, N1p = N1 + 1, Y1 = Y + 1,$
$p(N1p, N2, N3, N4, N5, N6, N71, N8, N9, N10, N11, N12, N13, N14, Y1).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N8 > 1, N81 = N8 - 1, N91 = N9 + 1,$
$p(N1, N2, N3, N4, N5, N6, N7, N81, N91, N10, N11, N12, N13, N14, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N9 >= 1, N91 = N9 - 1, N101 = N10 + 1, Y1 = Y + 1,$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N91, N101, N11, N12, N13, N14, Y1).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N10 >= 1, N101 = N101 - 1, N111 = N11 - 1,$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N101, N111, N12, N13, N14, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N11 >= 1, N111 = N11 - 1, N1p = N1 + 1,$
$p(N1p, N2, N3, N4, N5, N6, N7, N8, N9, N10, N111, N12, N13, N14, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N12 >= 1, N121 = N12 - 1, N131 = N13 + 1, Y1 = Y + 1,$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N121, N131, N14, Y1).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N13 >= 1, N131 = N13 - 1, N141 = N14 + 1,$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N131, N141, Y).$
$p(N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N14, Y) \leftarrow$
$N14 >= 1, N141 = N14 - 1, N1p = N1 + 1,$
$p(N1p, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, N13, N141, Y).$

I N F O R M A T I K

Below you find a list of the most recent technical reports of the research group *Logic of Programming* at the Max-Planck-Institut für Informatik. They are available by anonymous ftp from our ftp server `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Birgit Hofmann
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid *E*-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-97-2-012	L. Bachmair, H. Ganzinger, A. Voronkov	Elimination of Equality via Transformation with Ordering Constraints
MPI-I-97-2-011	L. Bachmair, H. Ganzinger	Strict Basic Superposition and Chaining
MPI-I-97-2-010	S. Vorobyov, A. Voronkov	Complexity of Nonrecursive Logic Programs with Complex Values
MPI-I-97-2-009	A. Bockmayr, F. Eisenbrand	On the Chvátal Rank of Polytopes in the 0/1 Cube
MPI-I-97-2-008	A. Bockmayr, T. Kasper	A Unifying Framework for Integer and Finite Domain Constraint Programming
MPI-I-97-2-007	P. Blackburn, M. Tzakova	Two Hybrid Logics
MPI-I-97-2-006	S. Vorobyov	Third-order matching in $\lambda\rightarrow$-Curry is undecidable
MPI-I-97-2-005	L. Bachmair, H. Ganzinger	A Theory of Resolution
MPI-I-97-2-004	W. Charatonik, A. Podelski	Solving set constraints for greatest models
MPI-I-97-2-003	U. Hustadt, R.A. Schmidt	On evaluating decision procedures for modal logic
MPI-I-97-2-002	R.A. Schmidt	Resolution is a decision procedure for many propositional modal logics