Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Constraint-based Deductive Model Checking

Giorgio Delzanno and Andreas Podelski

Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
{delzanno|podelski}@mpi-sb.mpg.de

Abstract.

Contents

1 Introduction 1
2 Translating Concurrent Systems into CLP 2
3 Expressing CTL Properties in CLP 4
4 Defining a Model Checking Method 5
5 Implementation in a CLP system 7
6 Infinite-State Integer Systems 9
7 Case-studies 13
8 Related Work 17
9 Conclusion and Future Work 18
A Preliminaries on CLP 21

1 Introduction

Automated verification methods can today be applied to practical systems [McM93]. One reason for this success is that implicit representations of finite sets of states through Boolean formulas can be handled efficiently via BDD's [BCM+90]. 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. There has been much recent effort in verifying such systems (see e.g. [AČJT96, BW98, BGP97, CJ98, HHWT97, HPR97, LPY97, SKR98]). One important research goal is to find appropriate data structures for implicit representations of infinite sets of states, and design model checking algorithms that perform well on practical examples.

It is obvious that the metaphor of *constraints* is useful, if not unavoidable for the implicit representation of sets of states (simply because constraints represent a relation and states are tuples of values). The question is whether and how the concepts and the systems for programming over constraints as first-class data structures (see e.g. [Pod94, Wal96]) can be used for the verification of infinite-state systems. The work reported in this paper investigates Constraint Logic Programming (see [JM94]) as a conceptual basis and as a practical implementation platform for model checking.

We present a translation from *concurrent systems* with infinite state spaces to CLP programs that preserves the semantics in terms of transition sequences. The formalism of 'concurrent systems' is a widely-used guarded-command specification language with shared variables promoted by Shankar [Sha93]. Using this translation, we exhibit the connection between states and *ground atoms*, between sets of states and *constrained facts*, between the pre-condition operator and the *logical consequence operator* of CLP programs, and, finally, between CTL properties (safety, liveness) and model-theoretic or denotational program semantics. This connection suggests a natural approach to model checking for infinite-state systems using CLP: model checking as as *deduction* of logical consequences of a CLP program. In fact, the model of a CLP program can be characterized as the fixpoint of the logical consequence operator, a specialization of *modus ponens*[1] to the fragment of Horn clauses. This operator is made effective by using constraint facts to implicitly represent set of ground atoms. Constraint facts can be manipulated symbolically via the operations defined on constraints (e.g. variable elimination, satisfibility and entailment tests).

The use of *deduction* to compute temporal properties allows us to enhance the model checking procedure by enriching the set of inference rules used to generate logical consequences of a CLP program. Similar techniques are used, e.g., in Constraint Databases [KKR95,

[1] If A and $A \rightarrow B$ hold, then B holds.

Concurrent Systems	$\rightsquigarrow$	CLP programs
Temporal Properties	$\rightsquigarrow$	Models of CLP Programs
Model Checking	$\rightsquigarrow$	Deduction

Fig. 1. A connection between two fields.

Rev93,RSS92] to improve the efficiency of bottom-up query evaluation.

We explore the potential of this approach practically by using one of the existing CLP systems with different constraint domains as an implementation platform. We have implemented an algorithm to generate models for CLP programs using constraint solvers over reals and Booleans. The implementation amounts to a simple and direct form of meta-programming: the input is itself a CLP program; constraints are syntactic objects that are passed to and from the built-in constraint solver; the fixpoint iteration is a source-to-source transformation for CLP programs.

As practical examples, we focus on the verification problem for systems with (unbounded) integer values; see e.g. [BW94,BW98,Bul98,BGP97,BGP98, Čer94,CJ98,FR96,SKR98]. The problem is undecidable for most classes of practical importance. Following [BW94,BW98], we apply our possibly non-terminating model checking procedure and we use abstractions to enforce or (simply speed-up) termination on practical examples.

As first abstraction, we consider the relaxation from integers to reals of the operators used in the definition of the model checking algorithms. We show that the relaxation is accurate for a wide class of integer systems (e.g. integer vector systems, Petri Nets, integral relational automata, discrete timed systems). This abstraction is used to make *each step* of the model checking procedure as efficient as possible.

In addition to the relaxation real-int, we present a set of inference rules to accelerate the computation of logical consequences of CLP programs. The rules are given through an inference system specialized to CLP programs with linear constraints. We show that the algorithm resulting from enhancing the model checking procedure (based on fixpoint computations) with the application of the acceleration rules still yields a full test for temporal properties. The correctness and accuracy of the method is ensured by the soundness of the inference rules. Given the rule-based nature of CLP programs, the acceleration rules can be naturally accomodated in our CLP implementation.

These abstractions allow us to solve the problems taken into considerations at acceptaple cost. Furthermore, the experiments show that, perhaps surprisingly, the powerful (triple-exponential time) decision procedure for Presburger Arithmetic used in other approaches [BGP98,SKR98] for the same verification problems is not needed; instead, the (polynomial-time) consistency and entailment tests for linear arithmetic constraints (without disjunction) that are provided by CLP systems are sufficient.

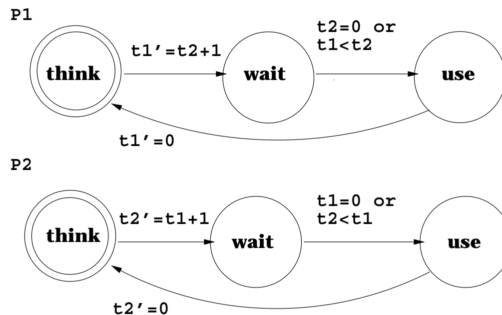


Fig. 2. Automata for the bakery algorithm.

Plan of the paper. In Section 2, we will introduce the formal setting of CLP (see also Appendix A), and the translation of concurrent systems to CLP programs. In Section 3, we will show how to express CTL properties in terms of CLP program semantics. In Section 4, we will turn the theory into practice discussing a model checking method. In Section 5, we will show how the method can be implemented using the features of existing CLP systems. In Section 6, we will present the techniques that can be used to analyze integer infinite-state systems using efficient constraint solving. In Section 7, we will present a number of case-studies. In Section 8, we will discuss related work. And, finally, in Section 8, we will conclude the paper with the future perspectives of our work.

2 Translating Concurrent Systems into CLP

We take the bakery algorithm (see [BGP97] and Fig. 2) as an example of a concurrent program, using the notation of [MP95]:

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

where $P_1 \mid\mid P_2$ is the parallel execution of the subprograms P_1 and P_2, and P_1 is defined by:

$$\begin{array}{ll} \textbf{repeat} & \\ \quad \textit{think}: & turn_1 := turn_2 + 1; \\ \quad \textit{wait}: & \textbf{when}\ turn_1 < turn_2\ \textbf{or}\ turn_2 = 0\ \textbf{do} \\ \quad \textit{use}: & \left[\begin{array}{l} \textbf{critical section}; \\ turn_1 := 0 \end{array}\right. \\ \textbf{forever} & \end{array}$$

and P_2 is defined symmetrically. The algorithm ensures the *mutual exclusion* property (at most one of two processes is in the critical section at every point of time). The integer values of the two variables $turn_1$ and $turn_2$ in reachable states are unbounded; note that a process can enter *wait* before the other one has reset its counter to 0.

The concurrent program above can be directly encoded as the concurrent system $\mathcal{S}$ in Figure 3 following the scheme in [Sha93]. Each process is associated with a *control variable* ranging over the control locations (i.e. program labels). The *data variables* correspond to the program variables. The states of $\mathcal{S}$ are tuples of control and data values, e.g. $\langle think, think, 0, 3\rangle$. The primed version of a variable in an action stands for its successor value. We omit conjuncts like $p_2' = p_2$ expressing that the value remains unchanged.

Following the scheme proposed in this paper, we translate the concurrent system for the bakery algorithm into the CLP program shown in Figure 2 (here, p is a dummy predicate symbol, $think, wait$, and use are constants, and variables are capitalized; note that we often separate conjuncts by commas instead of using "$\wedge$").

If the reader is not familiar with CLP, the following introduction together with Appendix A are all one needs to know for this paper. A CLP program is simply a logical formula, namely a universally quantifed conjunction of implications (like the one in Figure 2; the implications are usually called *clauses*). Its first reading is the usual first-order logic semantics. We give it a second reading as a non-deterministic sequential program. The program states are *atoms*, i.e., applications of the predicate p to values such as $p(think, think, 0, 3)$. The successor state of a state s is any atom s' such that the atom s is a direct logical consequence of the atom s' under the program formula. This again is the case if and only if the implication $s \leftarrow s'$ is an *instance* of one of the implications.

For example, the state $p(think, think, 0, 3)$ has as a possible successor the state $p(wait, think, 4, 3)$, since $p(think, think, 0, 3) \leftarrow p(wait, think, 4, 3)$ is an instance of the first implication for p (instantiate the variables with $P_2 = think$, $Turn_1 = 0$, $Turn_1' = 4$ and $Turn_2 = 3$).

A sequence of atoms such that each atom is a direct logical consequence of its successor in the sequence (i.e., a transition sequence of program states) is called a *ground derivation* of the CLP program.

In the following, we will always implicitly identify a state of a concurrent system $\mathcal{S}$ with the corresponding atom of the CLP program $P_\mathcal{S}$; for example, $\langle think, think, 0, 3\rangle$ with $p(think, think, 0, 3)$.

We observe that the transition sequences of the concurrent system $\mathcal{S}$ in Figure 3 are exactly the ground derivations of the CLP program $P_\mathcal{S}$ in Figure 2. Moreover, the set of all predecessor states of a set of states in $\mathcal{S}$ is the set of its direct logical consequences under the CLP program $P_\mathcal{S}$. We will show that these facts are generally true and use them to characterize CTL properties in terms of the denotational (fixpoint) semantics associated with CLP programs.

We will now formalize the connection between concurrent systems and CLP programs. We assume that for each variable x there exists another variable x', the primed version of x. We write $\mathbf{x}$ for the tuple of variables $\langle x_1, \ldots, x_n\rangle$ and $\mathbf{d}$ for the tuple of values $\langle d_1, \ldots, d_n\rangle$. We denote validity of a first-order formula ψ wrt. to a structure $\mathcal{D}$ and an assignment α by $\mathcal{D}, \alpha \models \psi$. As usual, $\alpha[\mathbf{x} \mapsto \mathbf{d}]$ denotes an assignment in which the variables in $\mathbf{x}$ are mapped to the values in $\mathbf{d}$. In the examples of Section 7 formulas will be interpreted over the domains of integers and reals. Note however that the following presentation is given for any structure $\mathcal{D}$.

A *concurrent system* (in the sense of [Sha93]) is a triple $\langle V, \Theta, \mathcal{E}\rangle$ such that

- V is the tuple $\mathbf{x}$ of control and data variables,
- Θ is a formula over V called the *initial condition*,
- $\mathcal{E}$ is a set of pairs $\langle\psi, \phi\rangle$ called *events*, where the *enabling condition* ψ is a formula over V and the *action* ϕ is a formula of the form $x_1' = e_1 \wedge \ldots x_n' = e_n$ with expressions $e_1, \ldots, e_n$ over V.

The primed variable x' appearing in an action is used to represent the value of x after the execution of an event. In the examples, we use the notation **cond** ψ **action** ϕ for the event $\langle\psi, \phi\rangle$ (omitting conjuncts of the form $x' = x$).

The semantics of the concurrent system $\mathcal{S}$ is defined as a transition system whose states are tuples $\mathbf{d}$ of values in $\mathcal{D}$ and the transition relation τ is defined by

$$\begin{aligned}\tau \;=\; \{\langle\mathbf{d}, \mathbf{d'}\rangle \mid\; & \mathcal{D}, \alpha[\mathbf{x} \mapsto \mathbf{d}] \models \psi,\\ & \mathcal{D}, \alpha[\mathbf{x} \mapsto \mathbf{d}, \mathbf{x'} \mapsto \mathbf{d'}] \models \phi,\\ & \langle\psi, \phi\rangle \in \mathcal{E}\}.\end{aligned}$$

The pre-condition operator $pre_\mathcal{S}$ of the concurrent system $\mathcal{S}$ is defined through the transition relation: $pre_\mathcal{S}(S) = \{\mathbf{d} \mid \text{exists } \mathbf{d}' \in S \text{ such that } \langle\mathbf{d}, \mathbf{d}'\rangle \in \tau\}$.

For the translation to CLP programs, we view the formulas for the enabling condition and the action as *constraints* over the structure $\mathcal{D}$ (see [JM94]). We introduce p for a dummy predicate symbol with arity n, and *init* for a predicate with arity 0. [2]

Definition 1 (Translation of concurrent systems to CLP programs). The concurrent program $\mathcal{S}$ is encoded as the CLP program $P_\mathcal{S}$ given below, if $\mathcal{S} = \langle V, \Theta, \mathcal{E}\rangle$ and V is the tuple of variables $\mathbf{x}$.

$$\begin{aligned}P_\mathcal{S} \;=\; & \{p(\mathbf{x}) \;\leftarrow\; \psi \wedge \phi \wedge p(\mathbf{x'}) \mid \langle\psi, \phi\rangle \in \mathcal{E}\} \;\cup\\ & \{init \;\leftarrow\; \Theta \wedge p(\mathbf{x})\}\end{aligned}$$

[2] Note that e.g. $p(think, P_2, Turn_1, Turn_2) \leftarrow \ldots$ in the notation used in examples is equivalent to $p(P_1, P_2, Turn_1, Turn_2) \leftarrow P_1 = think \wedge \ldots$ in the notation used in formal statements.

Control variables p_1, p_2 : $\{think, wait, use\}$
Data variables $turn_1, turn_2$: $int.$
Intial condition $p_1 = think \wedge p_2 = think \wedge turn_1 = turn_2 = 0$

Events		
	cond $p_1 = think$	**action** $p_1' = wait \wedge turn_1' = turn_2 + 1$
	cond $p_1 = wait \wedge turn_1 < turn_2$	**action** $p_1' = use$
	cond $p_1 = wait \wedge turn_2 = 0$	**action** $p_1' = use$
	cond $p_1 = use$	**action** $p_1' = think \wedge turn_1' = 0$
	... symmetrically for Process 2	

Fig. 3. Concurrent system $\mathcal{S}$ specifying the bakery algorithm

$init \leftarrow Turn_1 = 0, Turn_2 = 0, p(think, think, Turn_1, Turn_2)$,

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

... symmetrically for Process 2

Fig. 4. CLP program $P_{\mathcal{S}}$ for the concurrent system $\mathcal{S}$ in Figure 3.

The *direct consequence operator* T_P associated with a CLP program P (see [JM94]) is a function defined as follows: applied to a set S of atoms, it yields the set of all atoms that are direct logical consequences of atoms in S under the formula P. Formally,

$$T_P(S) = \{p(\mathbf{d}) \mid p(\mathbf{d}) \leftarrow p(\mathbf{d}') \text{ is an instance of a clause in } P,\ p(\mathbf{d}') \in S\}.$$

We obtain a (ground) instance by replacing all variables with values. In the next statement we make implicit use of our convention of identifying states $\mathbf{d}$ and atoms $p(\mathbf{d})$.

Theorem 1 (Adequacy of the translation $\mathcal{S} \mapsto P_{\mathcal{S}}$).

(i) The state sequences of the transition system defined by the concurrent system $\mathcal{S}$ are exactly the ground derivations of the CLP program $P_{\mathcal{S}}$.
(ii) The pre-condition operator of $\mathcal{S}$ is the logical consequence operator associated with $P_{\mathcal{S}}$, formally: $pre_{\mathcal{S}} = T_{P_{\mathcal{S}}}$.

Proof. The clause $p(\mathbf{x}) \leftarrow \psi \wedge \phi \wedge p(\mathbf{x}')$ of $P_{\mathcal{S}}$ corresponds to the event $\langle \psi, \phi \rangle$. Its instances are of the form $p(\mathbf{d}) \leftarrow p(\mathbf{d}')$ where $\mathcal{D}, \alpha[\mathbf{x} \mapsto \mathbf{d}, \mathbf{x}' \mapsto \mathbf{d}'] \models \psi \wedge \phi$. Thus, they correspond directly to the pairs $\langle \mathbf{d}, \mathbf{d}' \rangle$ of the transition relation τ restricted to the event $\langle \psi, \phi \rangle$. To prove (i), we first show that state sequences correspond to ground derivation by induction of the length of a sequence. The base case follows by noting that the intial states are instances of the body of the *init*-clause. Let us assume now that the thesis holds for a state-derivation $\mathbf{d}_0\mathbf{d}_1 \ldots \mathbf{d}_i$. If $\langle \mathbf{d}_i, \mathbf{d}_{i+1} \rangle \in \tau$ there exists a clause in the translation such that its set of instances contains $p(\mathbf{d}_i) \leftarrow p(\mathbf{d}_{i+1})$. Furthermore, by inductive hypothesis, $p(\mathbf{d}_0)p(\mathbf{d}_1) \ldots p(\mathbf{d}_i)$ is a ground derivation. Thus, by applying a resolution step we obtain the new sequence $p(\mathbf{d}_0)p(\mathbf{d}_1) \ldots p(\mathbf{d}_i)p(\mathbf{d}_{i+1})$. The converse can be proved by induction on the length of a derivation.

Point (ii) follows directly by definition of $pre_{\mathcal{S}}$ and $T_{P_{\mathcal{S}}}$. □

As an aside, if we translate $\mathcal{S}$ into the CLP program $P_{\mathcal{S}}^{post}$ where

$$P_{\mathcal{S}}^{post} = \{p(\mathbf{x}) \wedge \psi \wedge \phi \rightarrow p(\mathbf{x}') \mid \langle \psi, \phi \rangle \in \mathcal{E}\} \cup \{\Theta \rightarrow p(\mathbf{x})\}$$

then the post-condition operator is the logical consequence operator associated with $P_{\mathcal{S}}$, formally: $post_{\mathcal{S}} = T_{P_{\mathcal{S}}^{post}}$ We thus obtain the characterization of the set of reachable states as the least fixpoint of $T_{P_{\mathcal{S}}^{post}}$.

3 Expressing CTL Properties in CLP

We will use the temporal connectives: EF (exists finally), EG (exists globally), AF (always finally), AG (always globally) of CTL (Computation Tree Logic) to express *safety* and *liveness* properties of transition systems. Following [Eme90], we identify a temporal property with the set of states satisfying it.

In the following, the notion of *constrained facts* will be important. A constrained fact is a clause $p(\mathbf{x}) \leftarrow c$ whose body contains only a constraint c. Note that an instance of a constrained fact is of the form $p(\mathbf{d}) \leftarrow true$ which is the same as the atom $p(\mathbf{d})$, i.e. it is a state. Given a set of constrained

facts F, we write $[F]_{\mathcal{D}}$ for the set of instances of clauses in F (also called the 'meaning of F' or the 'set of states represented by F'). For example, the meaning of the singleton F_{mut} consisting of the fact $p(P_1, P_2, Turn_1, Turn_2) \leftarrow P_1{=}use, P_2{=}use$ is the set of states $[F_{mut}]_{\mathcal{D}} = \{p(use, use, 0, 0), p(use, use, 1, 0), \ldots\}$.

The application of a CTL operator on a set of constrained facts F is defined in terms of the meaning of F. For example, $EF(F)$ is the set of all states from which a state in $[F]_{\mathcal{D}}$ is reachable. In our examples, we will use a more intuitive notation and write e.g. $EF(p_1 = p_2 = use)$ instead of $EF(F_{mut})$.

As an example of a safety property, consider *mutual exclusion* for the concurrent system $\mathcal{S}$ in Figure 3 ("the two processes are never in the critical section at the same time"), expressed by $AG(\neg\,(p_1 = p_2 = use))$. Its complement is the set of states $EF(p_1 = p_2 = use)$. As we can prove, this set is equal to the least fixpoint for the program $P_{\mathcal{S}} \oplus F_{mut}$ that we obtain from the union of the CLP Program $P_{\mathcal{S}}$ in Figure 2 and the singleton set of constrained facts F_{mut}. We can compute this fixpoint and show that it does not contain the initial state (i.e. the atom $init$).

As an example of a liveness property, *starvation freedom* for Process 1 ("each time Process 1 waits, it will finally enter the critical section") is expressed by $AG(p_1 = wait \rightarrow AF(p_1 = use))$. Its complement is the set of states $EF(p_1 = wait \wedge EG(\neg\, p_1 = use))$. The set of states $EG(\neg\, p_1 = use)$ is equal to the greatest fixpoint for the CLP program $P_{\mathcal{S}} \oslash F_{starv}$ in Figure 5. We obtain $P_{\mathcal{S}} \oslash F_{starv}$ from the CLP Program $P_{\mathcal{S}}$ by a transformation wrt. to the following set of two constrained facts:

$$F_{starv} = \{\ p(P_1, P_2, Turn_1, Turn_2) \leftarrow P_1 = think, \quad p(P_1, P_2, Turn_1, Turn_2) \leftarrow P_1 = wait\ \ \}.$$

The transformation amounts to 'constrain' all clauses $p(label_1, _, _, _) \leftarrow \ldots$ in $P_{\mathcal{S}}$ such that $label_1$ is either $wait$ or $think$ (i.e., clauses of the form $p(use, _, _, _) \leftarrow \ldots$ are removed).

To give an idea about the model checking method that we will describe in the next section: in an intermediate step, the method computes a set F' of constrained facts such that the set of states $[F']_{\mathcal{D}}$ is equal to the greatest fixpoint for the CLP program $P_{\mathcal{S}} \oslash F$. The method uses the set F' to form a third CLP program $P_{\mathcal{S}} \oplus F'$. The least fixpoint for that program is equal to $EF(p_1 = wait \wedge EG(\neg\, p_1 = use))$. For more details, see Corollary 1 below.

We will now formalize the general setting.

Definition 2. The CLP programs $P \oplus F$ and $P \oslash F$ are the following formulas, for a given CLP program P and a set of constrained facts F.

$$P \oplus F \;=\; P \cup F$$

$$P \oslash F \;=\{\ p(\mathbf{x}) \leftarrow c_1 \wedge c_2 \wedge p(\mathbf{x}') \mid p(\mathbf{x}) \leftarrow c_2\ \in F,\ p(\mathbf{x}) \leftarrow c_1 \wedge p(\mathbf{x}')\ \in P\}.$$

Theorem 2 (CTL properties and CLP program semantics). Let $\mathcal{S}$ be a concurrent system and let $\mathcal{P}_{\mathcal{S}}$ be the CLP program which results from applying the translation of Def. 1 to $\mathcal{S}$, then the following properties holds for all sets of constrained facts F.

$$EF(F) = lfp(T_{P_{\mathcal{S}} \oplus F})$$

$$EG(F) = gfp(T_{P_{\mathcal{S}} \oslash F})$$

Proof. From the fixpoint characterizations of CTL properties (see [Eme90]) we know that $EF(F) = \mu Z.F \cup EX(Z)$ and $EG(F) = \nu Z.F \cap EX(Z)$ where $EX(Z) \equiv pre_{S}(Z)$. From Theorem 1, using the operators of Def. 2, the CLP programs $P_{\mathcal{S}} \oplus F$ and $P_{\mathcal{S}} \oslash F$ are such that $T_{P_{\mathcal{S}} \oplus F}(Z) = pre_{S}(Z) \cup F$ and $T_{P_{\mathcal{S}} \oslash F}(Z) = pre_{S}(Z) \cap F$. As a consequence, we have that $EF(F) = lfp(T_{P_{\mathcal{S}} \oplus F})$ and $EG(F) = gfp(T_{P_{\mathcal{S}} \oslash F})$. □

By duality, we have that $AF(\neg\, F)$ is the complement of $gfp(T_{P \oslash F})$ and $AG(\neg\, F)$ is the complement of $lfp(T_{P \oplus F})$. We next single out two important CTL properties that we have used in the examples in order to express mutual exclusion and absence of individual starvation, respectively.

Corollary 1 (Safety and Liveness).

(i) The concurrent system $\mathcal{S}$ satisfies the safety property $AG(\neg\, F)$ if and only if the atom '$init$' is not in the least fixpoint for the CLP program $P_{\mathcal{S}} \oplus F$.

(ii) $\mathcal{S}$ satisfies the liveness property $AG(F_1 \rightarrow AF(\neg\, F_2))$ if and only '$init$' is not in the least fixpoint for the CLP program $P_{\mathcal{S}} \oplus (F_1 \wedge F')$, where F' is a set of constrained facts denoting the greatest fixpoint for the CLP program $P_{\mathcal{S}} \oslash F_2$.

For the constraints considered in the examples, the sets of constrained facts are effectively closed under negation (denoting complement). Conjunction (denoting intersection) can always be implemented as $F \wedge F' = \{p(\mathbf{x}) \leftarrow c_1 \wedge c_2 \mid\ p(\mathbf{x}) \leftarrow c_1 \in F,\ \ p(\mathbf{x}) \leftarrow c_2 \in F',\ \ c_1 \wedge c_2$ is satisfiable in $\mathcal{D}\}$.

4 Defining a Model Checking Method

It is important to note that temporal properties are undecidable for the general class of concurrent systems that we consider. Thus, the best we can hope for are 'good' semi-algorithms, in the sense of Wolper in [BW98]: "the determining factor will be how often they succeed on the instances for which verification is indeed needed" (which is, in fact, similar to the situation for most decidable verification problems [BW98]).

A set F of constrained facts is an *implicit representation* of the (possibly infinite) set of states S if $S = [F]_{\mathcal{D}}$. From now on, we always assume that F itself is finite. We will replace the operator T_P over sets of atoms (i.e., states) by the operator S_P over sets

$$
\begin{array}{lll}
init \;\leftarrow\; Turn_1 = 0, Turn_2 = 0, \; p(think, think, Turn_1, Turn_2), & & \\
p(think, P_2, Turn_1, Turn_2) & \leftarrow Turn_1' = Turn_2 + 1, & p(wait, P_2, Turn_1', Turn_2), \\
p(wait, P_2, Turn_1, Turn_2) & \leftarrow Turn_1 < Turn_2, & p(use, P_2, Turn_1, Turn_2), \\
p(wait, P_2, Turn_1, Turn_2) & \leftarrow Turn_2 = 0, & p(use, P_2, Turn_1, Turn_2), \\
p(wait, think, Turn_1, Turn_2) & \leftarrow Turn_2' = Turn_1 + 1, & p(wait, wait, Turn_1, Turn_2'), \\
p(wait, wait, Turn_1, Turn_2) & \leftarrow Turn_2 < Turn_1, & p(wait, use, Turn_1, Turn_2), \\
p(wait, wait, Turn_1, Turn_2) & \leftarrow Turn_1 = 0, & p(wait, use, Turn_1, Turn_2), \\
p(wait, use, Turn_1, Turn_2) & \leftarrow Turn_2' = 0, & p(wait, think, Turn_1, Turn_2'), \\
p(think, think, Turn_1, Turn_2) & \leftarrow Turn_2' = Turn_1 + 1, & p(think, wait, Turn_1, Turn_2'), \\
p(think, wait, Turn_1, Turn_2) & \leftarrow Turn_2 < Turn_1, & p(think, use, Turn_1, Turn_2), \\
p(think, wait, Turn_1, Turn_2) & \leftarrow Turn_1 = 0, & p(think, use, Turn_1, Turn_2), \\
p(think, use, Turn_1, Turn_2) & \leftarrow Turn_2' = 0, & p(think, think, Turn_1, Turn_2')
\end{array}
$$

Fig. 5. The CLP program $P_{\mathcal{S}} \oslash F_{starv}$ for the concurrent system $\mathcal{S}$ in Figure 3.

of constrained facts, whose application $S_P(F)$ is effectively computable (see Appendix A). If the CLP programs P is an encoding of a concurrent system, we can define S_P as follows (note that F is closed under renaming of variables since clauses are implicitly universally quantified; i.e., if $p(x_1, \ldots, x_n) \leftarrow c \in F$ then also $p(x_1', \ldots, x_n') \leftarrow c[x_1'/x_1, \ldots, x_n'/x_n] \in F$.)

$$
\begin{aligned}
S_P(F) \;=\; \{p(\mathbf{x}) \leftarrow c_1 \wedge c_2 \mid \; & p(\mathbf{x}) \leftarrow c_1 \wedge p(\mathbf{x}') \;\in P, \\
& p(\mathbf{x}') \leftarrow c_2 \;\in F, \\
& c_1 \wedge c_2 \;\text{ is satisfiable in } \mathcal{D}\}
\end{aligned}
$$

If P contains also constrained facts $p(\mathbf{x}) \leftarrow c$, then these are always contained in $S_P(F)$.

The S_P operator has been introduced to study the non-ground semantics of CLP programs in [GDL95], where also its connection to the ground semantics is investigated: the set of ground instances of a fixpoint of the S_P operator is the corresponding fixpoint of the T_P operator, formally $lfp(T_P) = [lfp(S_P)]_{\mathcal{D}}$ and $gfp(T_P) = [gfp(S_P)]_{\mathcal{D}}$ (see Appendix A). Thus, Theorem 2 leads to the characterization of CTL properties through the S_P operator via:

$$
\begin{aligned}
EF(F) &= [lfp(S_{P \oplus F})]_{\mathcal{D}}, \\
EG(F) &= [gfp(S_{P \oslash F})]_{\mathcal{D}}.
\end{aligned}
$$

Now, a (possibly non-terminating) *model checker* can be defined in a straightforward way. It consists of the manipulation of constrained facts as implicit representations of (in general, infinite) sets of states. It is based on standard fixpoint iteration of S_P operators for the specific programs P according to the fixpoint definition of the CTL properties to be computed (see e.g. Corollary 1). An iteration starts either with $F = \emptyset$ representing the empty set of states, or with $F = \{p(\mathbf{x}) \leftarrow true\}$ representing the set of all states. The computation of the application of the S_P operator on a set of constrained facts F consists in scanning all pairs of clauses in P and constrained facts in F and checking the satisfiability of constraints; it produces a new (finite) set of constrained facts.

The iteration yields a (possibly infinite) sequence F_0, F_1, F_2, ... of sets of constrained facts. The iteration stops at i if the sets of states represented by F_i and F_{i+1} are equal, formally $[F_i]_{\mathcal{D}} = [F_{i+1}]_{\mathcal{D}}$.

We interleave the least fixpoint iteration with the test of membership of the state *init* in the intermediate results; this yields a semi-algorithm for safety properties.

The fixpoint test is based on the test of *subsumption* between two sets of constrained facts F and F'. We say that F is subsumed by F' if the set of states represented by F is contained in the set of states represented by F', formally $[F]_{\mathcal{D}} \subseteq [F']_{\mathcal{D}}$. Testing subsumption amounts to testing entailment of disjunctions of constraints by constraints.

We next describe some *optimizations* that have shown to be useful in our experiments (described in the next section). Our point here is to demonstrate that the combination of mathematical and logical reasoning allows one to find these optimizations naturally.

Local subsumption. For practical reasons, one may consider replacing subsumption by *local subsumption* as the fixpoint test. We say that F is locally subsumed by F' if every constrained fact in F is subsumed by some constrained fact in F'. Testing local subsumption amounts to testing entailment between quadratically many combinations of constraints. Generally, the fixpoint test may become strictly weaker but is more efficient, practically (an optimized entailment test for constraints is available in all modern CLP systems) and theoretically. For linear arithmetic constraints, for example, subsumption is prohibitively hard (co-NP [Sri93]) and local subsumption is polynomial [Sri93]. An abstract study of the complexity of local vs. full subsumption based on the CLP techniques can be found in [Mah95]; he shows that (full) subsumption is co-NP-hard unless it is equivalent to local subsumption.

Elimination of redundant facts. We call a set of constrained facts F *irredundant* if no element subsumes another one. We keep all sets of constrained facts $F_1, F_2, \ldots$ during the least fixpoint iteration irredundant by checking whether a new constrained fact in F_{i+1} that is not

locally subsumed by F_i itself subsumes (and thus makes redundant) a constrained fact in F_i. This technique is standard in CLP fixpoint computations [MR89].

4.1 Strategies

We obtain different fixpoint evaluation strategies (essentially, mixed forms of backward and forward analysis) by applying transformations such as the *magic-sets templates* algorithm [RSS92] to the CLP programs $P_S \oplus F$. Such transformations are natural in the context of CLP programs which may also be viewed as constraint data bases (see [RSS92, Rev93]).

The application of a kind of magic-set transformation on the CLP program $P = P_S \oplus F$, where the clauses have a restricted form (one or no predicate in the body), yields the following CLP program $\widetilde{P}$ (with new predicates $\widetilde{p}$ and $\widetilde{init}$).

$$\begin{aligned}\widetilde{P} \;=\; & \{p(\mathbf{x}) \leftarrow body, \widetilde{p}(\mathbf{x}') \mid p(\mathbf{x}) \leftarrow body \;\in P\} \;\cup \\ & \{\widetilde{p}(\mathbf{x}') \leftarrow c, \widetilde{p}(\mathbf{x}) \mid p(\mathbf{x}) \leftarrow c, p(\mathbf{x}') \;\in P\} \;\cup \\ & \{\widetilde{init} \leftarrow true\}\end{aligned}$$

We obtain the soundness of this transformation wrt. the verification of safety properties by standard results [RSS92] which say that $init \in lfp(T_P)$ if and only if $init \in lfp(T_{\widetilde{P}})$ (which is, $init \in lfp(S_{\widetilde{P}})$). The soundness continues to hold if we replace the constraints c in the clauses $\widetilde{p}(\mathbf{x}') \leftarrow c, \widetilde{p}(\mathbf{x})$ in $\widetilde{P}$ by constraints $c^{\#}$ that are entailed by c. We thus obtain a whole spectrum of transformations through the different possibilities to weaken constraints. In our example, if we weaken the arithmetical constraints by $true$, then the first iterations amount to eliminating constrained facts $p(label_1, label_2, _, _) \leftarrow \ldots$ whose *locations* $\langle label_1, label_2\rangle$ are "definitely" not reachable from the initial state.

5 Implementation in a CLP system

In this section, we describe the main procedures of our prototype. The rule-based nature of a CLP program allows us to incorporated naturally different optimizations for the basic model checking procedures (based on fixpoint computations). In the implementation we make an extensive use of the main features of CLP: unification of terms, meta-programming, dynamic manipulation of the program database, and constraint solving.

So far we used CLP programs as mathematical model for transition systems. Existing CLP systems, however, adopt incomplete strategies to compute a derivation for a goal and a program. First of all, CLP programs are executed following the *left-to-right* selection order for the literals of a goal (i.e., the body of a clause can be read as a sequence of subgoals). Furthermore, clauses are selected from the program following the order in which they are written (i.e., from top to bottom). Finally, when a subgoal fails (i.e., it has no successful derivations) the interpreter tries to find another possible derivation by selecting (in backtracking) other possible choices for the subgoals executed so far. We use the syntax A:-$B_1, \ldots, B_n$ to denote (clauses of) CLP programs used to code our model checker (i.e. their execution takes into account all the previous assumptions). This way, we distinguish them from CLP programs viewed as mathematical object, whose clauses will be still written as $p(\mathbf{x}) \leftarrow c, p(\mathbf{x'})$.

The algorithms of this section can be used with any constraint domain. All we need to know about the constraint solver is that it provides the following operations:

- *satisfiable(C,C')*: checks satisfiability for the constraint C and returns its solved form *C'*.
- *variable_elimination(C,T,C')*: *C'* is obtained from the constraint C projecting away the variables contained in the term T.
- *entail(C,D)*: checks if the constraint C entails the constraint D.

We will also use $C \wedge D$ to denote the conjunction of the constraints C and D.

In our implementation, we store the CLP program resulting from the translation of Def. 1, and the states computed during the exploration of its state space in the internal database provided by CLP systems. Each clause $p(\mathbf{t}) \leftarrow p(\mathbf{t'}) \wedge c$ of a program is stored as the fact $r(p(\mathbf{t}), p(\mathbf{t'}), c)$, whereas each state $p(\mathbf{t}) \leftarrow c$ is stored as the fact $s(I, p(\mathbf{t}), c)$ where I is a number denoting the iteration in which the fact has been added to the database.

Let us first consider the computation of EF. The predicate *apply_Sp* in Fig. 6 implements an application of the S_P operator to a given clause and a given fact. Both the clause and the fact are selected from the database (non-deterministically, in principle, following the order they are stored in the database, in practice) using the predicates *select_clause* and *select_fact*. The parameter I of the predicate *apply_Sp* denotes the index of the current fixpoint iteration. Note that, by monotonicity of S_P, at step $I+1$ we don't need to select facts with index less than I, i.e., each fact is selected only once during the whole fixpoint computation. The predicate *unify* (occurring in the body of *apply_Sp*) finds the most general unifier for the variables of the atoms A and B (i.e., after its execution the constraints C and D will range over the same set of variables). After checking for satisfiability of the conjunction $C \wedge D$, the resulting solved constraint *C'* is projected over the variables contained in the head of the clause *H*. Note that, here, constraints are considered as uninterpreted (when manipulated as facts) as well as interpreted terms (when passed to the constraint solver, e.g., via the predicates *satisfiable* and *variable_elimination*). The last step consists of adding the newly constructed fact ($H \leftarrow C''$) to the current

```
handle_new_fact(I,A,C):-
  select_fact(J,B,D),
  entail(A,C,B,D),!.

handle_new_fact(I,A,C,B,D):-
  assert_fact(I+1,A,C).

apply_Sp(I):-
  select_clause(H,B,C),
  select_fact(I,A,D),
  unify(A,B),
  satisfiable(C ∧ D,C'),
  variable_elimination(C',H,C"),
  handle_new_fact(I,H,C").
```

Fig. 6. Application of the S_P operator.

```
ef(P,F):-
  assert_program(P),
  assert_facts(F),
  least_fixpoint(0).

least_fixpoint(I):-
  all_derivations(apply_Sp(I)),
  initial_state_not_reached(I),!,
  least_fixpoint(I+1).

least_fixpoint(I):-
  write('Fixpoint reached or initial state detected').
```

Fig. 7. Main loop for least fixpoint.

database. This tasked is carried out by the predicate *handle_new_fact*. The new fact is added only if it does not entail an existing fact (i.e. the denotations of the new fact are not contained in the denotations of an existing fact, checked using the predicate *entail*). This way, we implement the *local* subsumpion test we mentioned before.

To enforce the exhaustive exploration of the database, we use a special built-in predicate we call *all_derivations* (in CLP systems this predicate is usually called *bag_of* or *find_all*). The invocation *all_derivations(G)* explores all possible derivations for the goal G. This idea is used to implement the core of the algorithm for computing EF as given in Fig. 7. The predicate *ef* loads the transition system P and the initial set of facts F in the program database (using CLP built-in primitives). Then, it starts the loop used to compute the backwards reachable states. The first clause for *least_fixpoint* computes all possible derivations for the predicate *apply_Sp* (by invoking *all_derivations*), and then tests for the presence of the initial state in the resulting database (predicate *initial_state_not_reached*). The CLP built-in predicate '!' (called cut) is used to make the predicate *least_fixpoint* deterministic after the test for the initial state. This way, the second clause

```
apply_Sp(I):-
  select_clause(H,B,C),
  select_fact(I,A,D),
  make_copy(A,D,CopyA,CopyD),
  unify(A,B),
  satisfiable(C ∧ D,C'),
  variable_elimination(C',H,C"),
  not_entail(H,C",CopyA,CopyD),
  handle_new_fact(I,H,C").
```

Fig. 8. Heuristic for the application of the S_P operator.

```
gfp(P):-
  assert_program(P),
  assert_template,
  greatest_fixpoint(0).

greatest_fixpoint(I):-
  all_derivations(apply_Sp_for_gfp(I)),!,
  not_contained(I,I+1),
  greatest_fixpoint(I+1).

greatest_fixpoint(I):-
  write('Greatest fixpoint reached').
```

Fig. 9. Main loop for greatest fixpoint.

for *least_fixpoint* is selected only when the predicate *apply_Sp* has no derivations (i.e., no news facts can be derived using S_P) or the initial state occurs in the database.

In Fig. 8, we modify the specification of *apply_Sp* applying the following heuristic: before testing for subsumption, we check if the newly computed fact is subsumed by the fact that produced it. To implement it, we simply make a copy with fresh new varibales of the selected fact, $A \leftarrow D$ in Fig. 8, and then we check that it is not entailed by the newly produced one. (The reason we need to a copy of the original fact is that the invocation of predicate *unify* may turn the fact $A \leftarrow D$ in an instance of the original selected fact; as an example, consider $unify(p(X), p(3))$). This heuristic may drastically cut the exploration of the state space needed to test subsumption. Note that the predicate *apply_Sp* can also be extended so as to incorporate the elimination of redundant facts we mentioned in the previous section. We will present other heuristics in Section 6.

We use similar techniques to implement the computation of the greatest fixpoint of S_P (e.g. used in EG). The main loop used to compute the greatest fixpoint is given in Fig. 9. The predicate *assert_template* is used to initialize the database with a set of facts representing the Herbrand base. The predicate *not_contained(I,I+1)* is used to test that the set of facts computed at step I are not subsumed by the facts computed at step $I + 1$ (i.e., the fixpoint has not been reached, yet). In its implementation, we used again a local subsumption test.

Similarly, we obtain *apply_Sp_for_gfp* from *apply_Sp* by substituting *not_subsumed(H,C”)* with the predicate invocation *not_subsumed(I+1,H,C”)*, i.e., we check that the fact $H \leftarrow C''$ is not subsumed by facts computed at step $I+1$ (in the greatest fixpoint computation the denotations of F_{i+1} coincide with the intersection of the denotations of $F_0, \ldots, F_{i+1}$).

A full CTL model checker can be obtained by combining this procedures and by using other set operations like intersection and complementation. Complementation is domain-specific: unless the constraint solver provides a built-in procedure to compute the complement of a constraint, the user has to define its own procedure.

We have implemented the model checking procedure described above in SICStus Prolog 3.7.1 using the arithmetic constraint solver CLP(Q,R) [Hol95] and the Boolean constraint solvers (which are implemented with BDDs). In the following section we will report on experimental results obtained by analysing several type of infinite-state systems.

6 Infinite-State Integer Systems

The verification problem for systems with (unbounded) integer values is receiving increasing attention; see e.g. [BW94, BW98, Bul98, BGP97, BGP98, Čer94, CJ98, FR96, SKR98]. The problem is undecidable for most classes of practical importance. So what can you do? There are basically two answers. (1) Give a possibly non-terminating algorithm that terminates for useful examples. This is the approach followed e.g. by [BW98, BGW+97]. (2) Give a semi-test that yields the definite answer for useful examples (the other answer being ‘don’t know’); see e.g. [BGP97, CGL92, LGS+94, Gra94, Dam96, Hal93, HPR97, HH95].

One obtains a semi-test by introducing abstractions that yield a conservative approximation of the original property. In this section, we consider automated, application-independent abstractions that do not enforce termination; instead, their approximation is accurate, i.e. does not loose information wrt. the original property. This way, we carry over the practical advantage of the second approach, namely the acceleration of the model-checking fixpoint computation, to the first approach while still implementing a full test, i.e. maintaining the definiteness of all answers.

To know the accuracy of an abstraction is important conceptually and pragmatically. Note that there seems to be no other way to predict its effect (“too rough?”) for a particular application. Obviously, the accuracy is useful for debugging (or finding typos); ‘don’t know’ answers are quite frustating. Finally, it allows us to determine the ‘correct’ parameters in initial-state specifications.

We have considered abstractions of different nature.

We show that the symbolic model checking procedure (based on the CLP semantic operators) over reals obtained by relaxation from the one over integers yields a full test of temporal properties for a specific class of CLP program; the class of integer systems that can be translated to this type of CLP programs contains many examples considered in the literature. The purpose of this abstraction is to accelerate each single fixpoint iteration. The number of iterations does not decrease. In order to show that it does not increase, we prove that the relaxation of the fixpoint test is accurate as well.

Applying history-dependent acceleration techniques as already foreseen in the abstract interpretation scheme [CC77], we show that a set of acceleration rules of the model-checking fixpoint operator yields an accurate model checking algorithm (i.e. a full test if terminating). The acceleration rules are formulated via a deductive system, i.e., they are specialized rules to compute logical consequences of CLP programs with linear constraints. The correctness and accuracy of the method is ensured by the soundness of the inference rules. Given the rule-based nature of CLP programs, the acceleration rules can be naturally accomodated in our CLP implementation of the model checking procedure.

We also consider approximations that may return *don’t know* answers. They can be applied when the accurate approximations fail from returning an answer or when the form of constraints involved in the systems does not guarantee the accuracy of the relaxation int-real.

6.1 Relaxation

In this section, we investigate the int-real relaxation of the symbolic model checking procedures (based on S_P) defined for a large class of CLP programs (concurrent systems) with unbounded positive integer values (which we call ‘simple’ for the lack of a better name). [3]

The relaxation from integers to reals stems from linear programming (see e.g. [Sch86]). The motivation there is the same as here: the manipulation of linear arithmetic constraints is less costly over reals than over integers, theoretically (e.g. polynomial vs. NP-hard for the satisfiability test) as well as practically (e.g., the variable elimination is less involved). Even if the complexity for integers is the same as for reals for a particular application (as discussed in details in the description of the Omega library, a solver for Presburger arithmetics [Pug91]), there exist many highly optimized constraint systems over reals, general-purpose such as CLP(R) and special purpose such as Uppaal [BLL+98] or Hytech [HHWT97], which one would like to exploit for model checking (simple) concurrent systems over integers.

[3] Note that this abstraction is *not* an embedding of the verification problem for a system over integers into one for a system over reals.

In the rest of the section we will define the class of CLP programs for which we can prove that the relaxation int-real of the constraint operators used in S_P is accurate.

Definition 3 (Simple CLP programs). A *simple* CLP program consists of clauses $p(\mathbf{x}) \leftarrow p(\mathbf{x}') \wedge \phi$ where ϕ (called *simple constraint*[4]) is built up according to the following grammar (where c is an integral constant, and x and y are (primed or unprimed) variables ranging over positive integers).

$$\phi \ ::= \ x \leq y + c \mid c \leq x \mid x \leq c \mid \text{true} \mid \text{false} \mid \phi_1 \wedge \phi_2.$$

Simple CLP programs results from the translation of systems that contain comparisons between variables, assignments between variables, increments and decrements. Note that the expression $x < y + c$ can be translated to $x \leq y + c - 1$, without loss of precision (by hypothesis, simple programs are interpreted over $\mathbb{N}$). Furthermore, $x = y + c$ can be translated to $x \leq y + c \ \wedge \ x \geq y + c$. Vector Addition Systems [KM69] (a.k.a. Petri Nets), and Integral Relational Automata [Čer94] are two examples of systems whose translation in CLP gives rise to simple CLP programs. The reachability problem is decidable for these classes (see e.g [Čer94,Lam92]). Other examples are multi-clocks automata [CJ98] and gap-order automata [FR96].

The above-mentioned decidability results are related to the general results for verification problems of infinite-state systems in [AČJT96,FS98]. The communication protocols considered in [BGP97,BGP98,SKR98], such as the *bakery* algorithm of Section 2, are examples of systems that can be translated to simple CLP programs but that do not seem to belong to a known decidable subclass.

We will interpret simple constraints over positive subset of both, the domains $\mathbb{N}$ and $\mathbb{R}$ of integers and reals, respectively. In the following, given a CLP program P, we will use $S_{P,\mathcal{D}}$ to fix the domain $\mathcal{D}$ of interpretation of the operators *satisfiable*, *entail*, and *variable_elimination* used for its definition (see Section 5).

Proposition 1 (Relaxation of constraint operators). The relaxation of the tests of satisfiability and entailment and of the variable elimination is accurate; i.e., the predicates *satisfiable*, *entail*, and *variable_elimination* over simple constraints yields the same results for $\mathcal{D} = \mathbb{N}$ and for $\mathcal{D} = \mathbb{R}$.

Proof. To show that the satisfiability test is invariant under the relaxation int-real, we note that a simple constraint $x - y \leq c$ is satisfiable in $\mathbb{R}$ if and only if $\mathit{floor}(x - y) \leq \mathit{floor}(c)$ is satisfiable in $\mathbb{N}$; the property extends to conjunctions of simple constraints. Furthermore, it is easy to see that the considered class of constraints is closed under application of Fourier-Motzkin's variable elimination (see also [BF99] for the special case of Petri Nets). Finally, to show that the entailment test is invariant under the relaxation, we simply note that the negation of an *atomic* simple constraint is still a simple costraint. Now, the thesis follows by noting that the test $\mathit{entail}(C, D_1 \wedge D_2)$ can be reduced to the tests $\mathit{not}(\mathit{satisfiable}(C \wedge \neg D_1))$ and $\mathit{not}(\mathit{satisfiable}(C \wedge \neg D_2))$. □

[4] In [BF99], they are called *counter regions* formulas.

Proposition 2 (Relaxation of S_P). Let P be a simple CLP program. The application of the operator S_P over integers, namely $S_{P,\mathbb{N}}$, and its real relaxation, namely $S_{P,\mathbb{R}}$, to a set of simple constrained facts I yield two sets of constrained facts denoting the same relation over integers. Formally,

$$[S_{P,\mathbb{R}}(I)]_{\mathbb{N}} \ = \ [S_{P,\mathbb{N}}(I)]_{\mathbb{N}} \ = \ T_P(I).$$

Proof. By definition of S_P and Prop. 1. □

The iteration of Prop. 2 yields that for all $k \geq 0$, $[S^k_{P,\mathbb{R}}(I)]_{\mathbb{N}} = [S^k_{P,\mathbb{N}}(I)]_{\mathbb{N}}$.

This means that the relaxations of the model checking procedures from integers to reals 'compute' (if terminating) the same set of states of simple systems. Moreover, since the *subsmuption* test is invariant under the relaxation, we obtain the following result.

Theorem 3 (Relaxation). The relaxation of the symbolic model checking procedures for safety and liveness properties of Corollary 1 defined for simple integer programs is accurate.

Note that, though our formal setting is that of CLP, the previous results can be generalized to any symbolic model checking procedure based on real-arithmetics, by simply substituting S_P with the (symbolic) predecessor operator used in that context. For instance, in [BF99], Berard and Fribourg apply the relaxation int-real to Petri Nets, in order to use HyTech for invariant checking. Finally, note that Proposition 2 holds for any CLP program with *simple constraints* in the body of clauses, i.e., the body of clause may contain more than a single atomic literal. We have restricted our formulation to unary programs in order to simplify the presentation (in this paper we are interested only in CLP programs obtained via the translation of integer systems).

We have applied our prototype implementation in CLP($\mathbb{R}$) to prove mutual exclusion and starvation freedom for the *bakery* algorithm (see Sect. 2 and Sect. 3). The computation terminates in both cases proving the algorithm correct. The resulting fixpoints are accurate by the results proved in this section. We will turn back to this and other examples after introducing acceleration methods for our model checking procedures.

$$\frac{\begin{array}{c} P \models p(\mathbf{x}) \leftarrow p(\mathbf{x'}) \wedge C, \\ F \models p(\mathbf{x}) \leftarrow x \leq y + c \wedge D, \\ D \text{ entails } \exists \mathbf{x'}.D[\mathbf{x'}/\mathbf{x}] \wedge C, \\ C \models x'{=}x + c_x \wedge y'{=}y + c_y \text{ where } c_y - c_x > 0 \end{array}}{P \oplus F \models p(\mathbf{x}) \leftarrow D} \text{ rule 1}$$

$$\frac{\begin{array}{c} P \models p(\mathbf{x}) \leftarrow p(\mathbf{x'}) \wedge C, \\ F \models p(\mathbf{x}) \leftarrow x = c \wedge D \\ D \text{ is equivalent to } \exists \mathbf{x'}.D[\mathbf{x'}/\mathbf{x}] \wedge C \\ C \models x'{=}x + 1 \end{array}}{P \oplus F \models p(\mathbf{x}) \leftarrow D \wedge x \geq c} \text{ rule 2}$$

$$\frac{\begin{array}{c} P \models p(\mathbf{x}) \leftarrow p(\mathbf{x'}) \wedge C, \\ F \models p(\mathbf{x}) \leftarrow x \leq c \wedge D \\ D \text{ entails } \exists \mathbf{x'}.D[\mathbf{x'}/\mathbf{x}] \wedge C \\ C \models x'{=}x + c_x \text{ where } c_x > 0 \end{array}}{P \oplus F \models p(\mathbf{x}) \leftarrow D} \text{ rule 3}$$

$$\frac{\begin{array}{c} P \models p(\mathbf{x}) \leftarrow p(\mathbf{x'}) \wedge C, \\ F \models p(\mathbf{x}) \leftarrow x \geq c \wedge D \\ D \text{ entails } \exists \mathbf{x'}.D[\mathbf{x'}/\mathbf{x}] \wedge C \\ C \models x'{=}x + c_x \text{ where } c_x < 0 \end{array}}{P \oplus F \models p(\mathbf{x}) \leftarrow D} \text{ rule 4}$$

Fig. 10. Deductive systems for accelerating least fixpoint computation.

6.2 Accurate Abstractions

In this section, we consider how one can achieve (or just speed up) the termination of the symbolic model checking algorithm for safety properties, without loss of precision.

Our method is based on the following intuition. Given a CLP program P and a set of facts F, $S_P(F)$ gives us the set of *direct* logical consequences (i.e., computed in one step) of $P \oplus F$. Basically, they are obtained applying the *modus ponens* rule of first order logic. In many cases, however, it is possible to use stronger inference rules that allow to *saturate* the set of logical consequences of $P \oplus F$ in one step.

Consider the program $p(x, y) \leftarrow y' = y + 1 \wedge p(x, y')$ and the fact $p(x, y) \leftarrow x \leq y$. The computation of $lfp(S_{P \oplus F})$ generates an infinite sequence of strictly increasing sets of facts,

$$\begin{array}{l} F_0 = \{p(x, y) \leftarrow x \leq y\}, \\ F_1 = F_0 \cup \{p(x, y) \leftarrow x \leq y + 1\}, \\ F_2 = F_1 \cup \{p(x, y) \leftarrow x \leq y + 2\}, \\ \ldots \end{array}$$

whose infinite union is equivalent to the fact $p(x, y) \leftarrow true$. However, it is easy to prove that $p(x, y) \leftarrow true$ is a logical consequence of $P \oplus F$ without having to go through the iterations of S_P.

The kind of deductive rules we will present can be viewed as a generalization of this simple idea. The resulting deductive system will be used to accelerate the computation of the least fipoint of S_P (i.e., they will be applied (if possible) at each iteration). The correctness of the method will follow by proving the soundness of the resulting deduction system.

Let P be a CLP program (not necessarily a simple program), F be a set of facts, and let $F \models G$ denote that G is a logical consequence of F. The inference rules are shown in Fig. 10. Rule 1, 3 and 4 are used to guess the direction of growth of the constraints generated during the iterations of S_P (i.e., a sort of widening operator formulated in logical terms). Rule 2 is used to detect a periodic behaviour in the modification of the value of variable x. Note that, for instance in rule 1, the first condition, namely $P \models p(\mathbf{x}) \leftarrow p(\mathbf{x'}) \wedge C$, means that the clause $p(\mathbf{x}) \leftarrow p(\mathbf{x'}) \wedge C$ can be obtained as a logical consequence of the clauses of the program P, i.e., it is not necessarily a clause of P.

Clearly, the rules can be extended so as to consider more involved constraints (e.g., inequalities with more than two variables). In this paper, however, we restrict out attention to simple constraints. It is also important to remark that the previous rules represent a sort of limit case for accelerations we can express using real constraints without explicit quantifiers over natural number. For instance, to represent the set of values obtained by repeatedly incrementing the variable x by c, we would need a constraint of the form $\exists n.x = n * c$ where n is a natural number. To handle this type of constraints it would be necessary to work with a mixed int-real constraint solver.

The acceleration rules of Fig. 10 are sound as we will prove in following proposition. This means that when incorporated in the least fixpoint computation they will not alter the result of the computation, i.e., the resulting fixpoint will be accurate. A note on the notation: in the rest of the section, we will write $S_P(\{f\})$ and $[\{f\}]$ as $S_P(f)$ and $[f]$, respectively (here f is a fact).

Proposition 3. Rules 1-4 of Fig. 10 are sound.

Proof. In this proof we use the properties mentioned in Appendix A. We only prove the soundness of the rule 1. Let Q consists of the clauses $p(\mathbf{x}) \leftarrow p(\mathbf{x'}) \wedge C$ and $p(\mathbf{x}) \leftarrow x \leq y + c \wedge D$ satisfying the hypothesis of rule 1. Let $f(n) = p(\mathbf{x}) \leftarrow D, x \leq y + c + n(c_y - c_x)$. We first prove that $[f(n)]_{\mathcal{D}}$ is a subset of $[S_Q^n(\emptyset)]_{\mathcal{D}}$ for all $n \geq 0$. The proof is by induction on n. Base case: by definition. Inductive step. Let us assume that $[f(n)]_{\mathcal{D}}$ is a subset of $[S_Q^n(\emptyset)]_{\mathcal{D}}$. By monotonicity of T_Q, $T_Q([f(n)]_{\mathcal{D}}) \subseteq$

$T_Q([S_Q^n(\emptyset)]_\mathcal{D})$, thus, by a property of S_Q, $[S_Q(f(n))]_\mathcal{D}) \subseteq [S_Q^{n+1}(\emptyset)]_\mathcal{D}$. Now, the expression on the left-hand side gives us the new fact $f' = p(\mathbf{x}) \leftarrow D[\mathbf{x'}/\mathbf{x}] \wedge C \wedge x \leq y + c + (n+1)(c_y - c_x)$ (i.e., $[f']_\mathcal{D} \subseteq [S_Q^{n+1}(\emptyset)]_\mathcal{D}$. We conclude noting that, by the third hypothesis of rule 1 (namely D entails $\exists \mathbf{x'}.D[\mathbf{x'}/\mathbf{x}] \wedge C$) it follows that $[p(\mathbf{x}) \leftarrow D \wedge x \leq y + c + (n+1)(c_y - c_x)]_\mathcal{D} \subseteq [f']_\mathcal{D}$. Let us go back now to the main proof. To conclude, we simply note that, since $c_y - c_x > 0$, the denotations of $\bigcup_{n=0}^{\omega}\{p(\mathbf{x}) \leftarrow D \wedge x \leq y + c + n(c_y - c_x)\}$ coincide with those of $p(\mathbf{x}) \leftarrow D$, i.e., $p(\mathbf{x}) \leftarrow D$ is a logical consequences of Q, and, by the first and second hypothesis of rule 1, is a logical consequence of $P \oplus F$. Similar arguments apply to the other rules. □

Theorem 4 (Acceleration). The algorithm obtained by abstracting the least fixpoint operator S_P in the symbolic model checking algorithm for safety properties with the acceleration defined in Figure 10 yields (if terminating) a full test of safety properties for concurrent systems over integers or reals.

Proof. By the soundness of the rules in Fig. 10. □

6.3 Strategy for acceleration

The inference rules of Fig. 10 are non-deterministic wrt. the clause, fact and constraint to consider. In this section, we will propose a strategy for the selection of the candidate clause and fact. Our model checker implements this strategy during the computation for safety properties.

Let us first consider rule 1 of Fig. 10. We assume that F is the set of facts computed at a given iteration for computing the least fixpoint of S_P. Now, let $f_1 = p(\mathbf{x}) \leftarrow C$ and $f_2 = p(\mathbf{x}) \leftarrow D$ be two facts in F, obtained, respectively, after i and j ($i < j$) applications of S_P. Furthermore, let us assume that C entails D and that there exists a sequence of clauses $c_1, \ldots, c_n$ in P such that $f_2 = S_{c_n}(\ldots(S_{c_1}(f_1))$ (i.e., f_2 is reachable from f_1).

Under these hypotheses, we will try to apply rule 1 to the fact f_1 (clearly, a logical consequence of F) and to the clause obtained by *unfolding* the clauses $c_1, \ldots, c_n$ (i.e., composing them into a single clause). The unfolding of a list of clauses is defined formally as follows. Let $c_i = p(\mathbf{x}) \leftarrow C_i \wedge p(\mathbf{x'})$ for $i : 1, \ldots, n$. We first compute the following fact:

$$p(\mathbf{x}) \leftarrow C = S_{c_n}(\ldots S_{c_2}(p(\mathbf{x}) \leftarrow C_1) \ldots).$$

The unfolded clause is then defined as $p(\mathbf{x}) \leftarrow C \wedge p(\mathbf{x'})$.

The method we propose can be viewed as a dynamic generation of *loops* of the original CLP programs.

A similar idea can be used for rules 3-4 of Fig. 10. For rule 2, we need a slightly different strategy. Since this rule is used to detect periodic behaviour of a variable, say x, instead of looking for two fact f_1 and f_2 such that $[f_1]_\mathcal{D} \subseteq [f_2]_\mathcal{D}$, we look for two facts $f_1 = p(\mathbf{x}) \leftarrow x = c \wedge D$ and $f_2 = p(\mathbf{x}) \leftarrow x = c + 1 \wedge C$ such that D is *equivalent* to C and f_2 is reachable from f_1 via a sequence of clauses (as before).

```
handle_new_fact(I,A,C):-
  select_fact(B,D),
  compare(I,A,C,B,D),!.

compare(I,A,C,B,D):-
  entail(A,C,B,D),!.

compare(I,A,C,B,D):-
  entail(B,D,A,C),
  compute_unf_clause(B,D,A,C,UnfC),
  accelerate(B,D,UnfC,D'),!,
  assert_fact(I+1,B,D').

handle_new_fact(I,A,C,B,D):-
  assert_fact(I+1,A,C).
```

Fig. 11. Code for acceleration rules.

These ideas can be easily incorporated in the CLP implementation of Section 5. Specifically, all we have to do is to modify the procedure *handle_new_fact* (see Fig. 6 of Section 5) as shown in Fig. 11. Remember that the predicate *handle_new_fact* takes care of inserting newly produced facts at a given iteration of the applications of S_P. In the new version, we first check that new fact is not subsumed by an existing one. Then, we apply our heuristics to derive loops from the programs. The predicate *compute_unf_clause* succeeds if the fact $A \leftarrow C$ can be reached from fact $B \leftarrow D$ using the unfolded clause *UnfC*. If the heuristics succeeds, we try to apply one of the acceleration rules (we assume the predicate *accelerate* to be defined according to the rules of Fig. 10). The predicate *accelerate* succeeds if the conditions of one of the rules of Fig.10 are satisfied returning the new constraint computed for the fact $B \leftarrow D$.

If the acceleration can not be applied we simply add the fact to the database (last rule for *handle_new_fact*).

6.4 Conservative Approximations

In many cases the accuracy of the relaxation int-real can not be guaranteed by the form of the program and of the property taken into consideration.

However, the relaxation int-real still gives us a conservative approximation of safety and liveness properties for this type of systems (i.e., systems that can not be translated to simple CLP programs). In fact, the following relation holds for any CLP program with integer variables and a collection I of (linear) constraint fact:

$$[S_{P,\mathbb{N}}(I)]_\mathbb{N} \subseteq [S_{P,\mathbb{R}}(I)]_\mathbb{N}.$$

In other words, when computig over $\mathbb{R}$, for a generic CLP program with constraint over integers, only negative an-

swers (e.g., the initial state is not in the least model computed over $\mathbb{R}$) are definite answers.

Following ideas developed in abstract interpretation [CC77], it is also possible to apply *acceleration* operators that may return *don't know* answers, i.e., when incorporated in fixpoint computation, they yield a conservative approximation of the property taken into consideration. In this section we will introduce a new *widening* operator $\Uparrow$ (in the style of [CH78], but without a termination guarantee) used to define $S^{\#}_P(F) = F \Uparrow S_P(F)$ (so that $[S_P(F)]_{\mathcal{D}} \subseteq [S^{\#}_P(F)]_{\mathcal{D}}$). The operator $\Uparrow$ may return an upper approximation of the least fixpoint for S_P.

The operator $\Uparrow$ is defined in terms of constrained facts. For example, if

$$\begin{array}{rl} F = \{p(X,Y) \leftarrow X \geq 0, Y \geq 0, X \leq Y\} & \\ F' = \{p(X,Y) \leftarrow X \geq 0, Y \geq 0, X \leq Y+1\} & \text{then} \\ F \Uparrow F' = \{p(X,Y) \leftarrow X \geq 0, Y \geq 0\}. & \end{array}$$

Formally, $F \Uparrow F'$ contains each constrained fact that is obtained from some constrained fact $p(\mathbf{x}) \leftarrow c_1 \wedge \ldots \wedge c_n$ in F' by removing all conjuncts c_i that are strictly entailed by some conjunct d_j of some 'compatible' constrained atom $p(\mathbf{x}) \leftarrow d_1 \wedge \ldots \wedge d_m$ in F, where 'compatible' means that the conjunction $c_1 \wedge \ldots \wedge c_n \wedge d_1 \wedge \ldots \wedge d_m$ is satisfiable. This condition restricts the applications of the widening operator e.g to facts with the same values for the control locations.

Contrary to the 'standard' widening operators in [CH78] and to the improved versions in [HPR97, BGP98], the operator $\Uparrow$ can be directly implemented using the entailment test between constraints; furthermore, it is applied fact-by-fact, i.e., without requiring a preliminary computation of the convex hull of union of polyhedra. Note that the convex hull is computationally very expensive and it might be a source of further loss of precision. Let us consider e.g. the two sets of constrained atoms

$$\begin{array}{l} F = \{p(\ell, X) \leftarrow X \geq 2\} \\ F' = \{p(\ell, X) \leftarrow X \geq 2,\ p(\ell, X) \leftarrow X \leq 0\}. \end{array}$$

When applied to F and F', each of the widening operator in [BGP98, CH78, HPR97] returns the (polyhedra denoted by the) fact $p(\ell, X) \leftarrow true$. In contrast, our widening is precise here, i.e., it returns F'.

Finally, note that the use of constrained facts automatically induces a partitioning over the state space wrt. the set of control locations. The partitioning reduces the applicability of the widening for the benefit of precision of the computation (see also [HPR97, BGP98]).

7 Case-studies

In this section we comment on some experimental results obtained with our model checker implemented in SICStus Prolog and the CLP(Q,R) library. In order to show the generality of the approach we select three different types of integer systems: communication protocols, parameterized systems, and constraint programs used for array-bounds checking of imperative programs. Communication protocols like the bakery algorithm are typical examples of concurrent systems, whereas the remaining examples are interesting for the difficulties their analysis may present. For the sake of simplicity, in the following sections we will bypass Shankar's intermediate syntax, directly translating the examples to CLP programs.

7.1 Communication Protocols

Bakery algorithm. Mutual exclusion and starvation freedom for the *bakery* algorithm (see Sect. 2 and Sect. 3) can be verified without the use of accelerations (execution time for starvation freedom: 0.9s). In versions of the bakery algorithm for 3 and 4 processes (not treated in [BGP97]), a maximum operator (used in assignments of priorities such as $Turn_1 = max(Turn_2, Turn_3) + 1$) is encoded case-by-case in the constraint representation. This makes the program size grow exponentially in the number of processes.

Ticket Algorithm. The *ticket* algorithm (see Fig. 12) is based on similar ideas as the bakery algorithm. Here, priorities are handled using two global variables, namely t and s. The variable t is used to assign new priorities to processes waiting for entering their critical section. The variable s is used to store the value of the ticket of the next process to be served. Each process has a local value used to store the current value of its ticket. Fig. 13 shows the simple CLP program resulting from the translation of the algorithm taken into consideration (for the translation, we follow the same method we followed for the bakery algorithm). The safety property is expressed by $AG(\neg(p_1 = use \wedge p_2 = use))$ (as for the bakery algorithm). Since both the program and the property contain simple constraints only we can predict that the analysis over $\mathbb{R}$ will be accurate.

We prove safety by applying the accurate acceleration (rule 1 of Fig. 10) during the fixpoint iterations. In a second experiment we applied the magic set transformation instead and obtained a proof in 0.6s. We proved starvation freedom, i.e., $AG(p_1 = wait \rightarrow AF(p_1 = use))$, in 1.5s applying the accurate acceleration for the outer least fixpoint (the inner greatest fixpoint terminates without abstraction).

Producer-consumer protocols are other interesting examples of concurrent programs. We will discuss some examples taken from [BGP98] in the following section.

Bounded Buffer. The first protocol we consider models the communication of producers and consumers connected via a buffer of size s. Fig. 14 shows the automata for a producer and a consumer; here the variable a denotes the number of *empty* cells in the buffer, p the

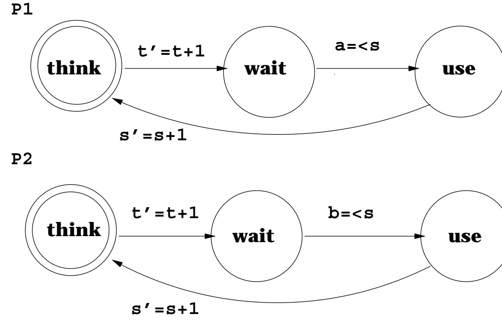
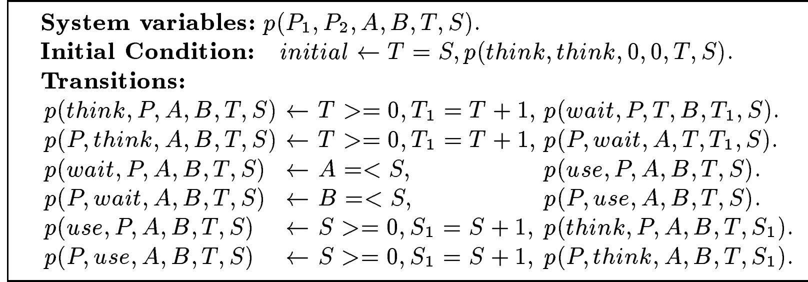


Fig. 12. Automata for the ticket algorithm.



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:
$p(think, P, A, B, T, S) \leftarrow T >= 0, T_1 = T + 1, p(wait, P, T, B, T_1, S)$.
$p(P, think, A, B, T, S) \leftarrow T >= 0, T_1 = T + 1, p(P, wait, A, T, T_1, S)$.
$p(wait, P, A, B, T, S) \leftarrow A =< S, p(use, P, A, B, T, S)$.
$p(P, wait, A, B, T, S) \leftarrow B =< S, p(P, use, A, B, T, S)$.
$p(use, P, A, B, T, S) \leftarrow S >= 0, S_1 = S + 1, p(think, P, A, B, T, S_1)$.
$p(P, use, A, B, T, S) \leftarrow S >= 0, S_1 = S + 1, p(P, think, A, B, T, S_1)$.

Fig. 13. The CLP-program for the ticket algorithm.

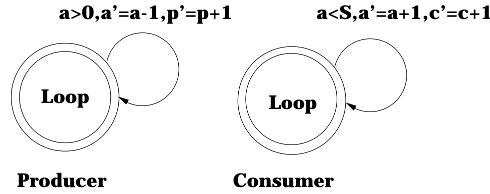


Fig. 14. Producer-consumer with bounded buffer.

number of produced items, and c the number of consumed items. The CLP-program that describes a system with a producer, a consumer and two buffers is given in Figure 15. The first invariant that we want to prove is $Inv_1 = AG(p_1+p_2-(c_1+c_2) = s-a)$. Note that the previous constraint is not simple, i.e., the relaxation int-real will give us a conservative approximation of the property. Before applying our model checker, we transform the previous property in $AG(p_1 + p_2 - (c_1 + c_2) + a \leq s - 1 \wedge p_1 + p_2 - (c_1 + c_2) + a \geq s + 1)$. Our model checker proves the property in 0.2s without need of accelerations. Another safety condition is given by $Inv_2 = AG(0 \leq p_1+p_2-(c_1+c_2) \leq s)$. Using the invariant Inv_1 we can write Inv_2 as the safety property $AG(0 \leq a \leq s)$. Furthermore, it is easy to see that the above program is safe if and only if the following (simple) program is safe:

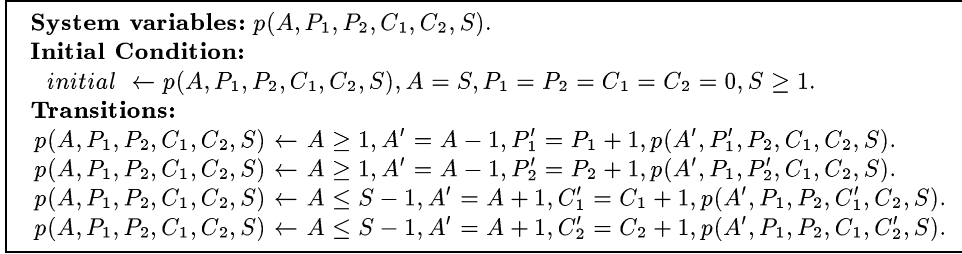
$$p(A, S) \leftarrow p(A', S), A \geq 1, A' = A - 1.$$
$$p(A, S) \leftarrow p(A', S), A \leq S - 1, A' = A + 1.$$

Our model checker proves the safety property Inv_2 for the new program in one step.

Unbounded Buffer. We consider now a protocol for producers and consumers connected via unbounded buffers. 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. Fig. 16 shows the automata for a pair producer-consumer and only one buffer. The variable p keeps track of the number of produced items, the variable c the number of consumed items and q the number of elements in the buffer. The CLP program in Fig. 17 models a system with a producer a consumer and two unbounded buffers. To prove the invariant $AG(p \geq c)$ we prove that $AG(p = c + q_1 + q_2)$, (note that $q_i \geq 0$), i.e., $AG(p \leq c + q_1 + q_2 - 1 \wedge p \geq c + q_1 + q_2 + 1)$ (a non-simple constraint). We prove the property in 3 steps by applying the widening operator of Section 6.4.

7.2 Array Bounds Checking.

In this section, we will discuss an application of our model checker for checking array bounds of imperative programs. The main idea is to extract, from the flow



System variables: $p(A, P_1, P_2, C_1, C_2, S)$.
Initial Condition:
$initial \leftarrow p(A, P_1, P_2, C_1, C_2, S), A = S, P_1 = P_2 = C_1 = C_2 = 0, S \geq 1.$
Transitions:
$p(A, P_1, P_2, C_1, C_2, S) \leftarrow A \geq 1, A' = A - 1, P_1' = P_1 + 1, p(A', P_1', P_2, C_1, C_2, S).$
$p(A, P_1, P_2, C_1, C_2, S) \leftarrow A \geq 1, A' = A - 1, P_2' = P_2 + 1, p(A', P_1, P_2', C_1, C_2, S).$
$p(A, P_1, P_2, C_1, C_2, S) \leftarrow A \leq S - 1, A' = A + 1, C_1' = C_1 + 1, p(A', P_1, P_2, C_1', C_2, S).$
$p(A, P_1, P_2, C_1, C_2, S) \leftarrow A \leq S - 1, A' = A + 1, C_2' = C_2 + 1, p(A', P_1, P_2, C_1, C_2', S).$

Fig. 15. CLP for producers and consumers connected via a bounded buffer.

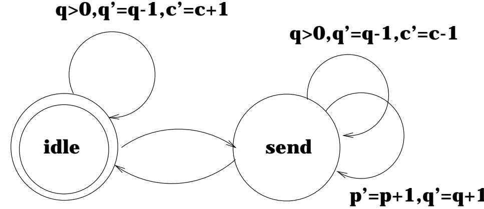
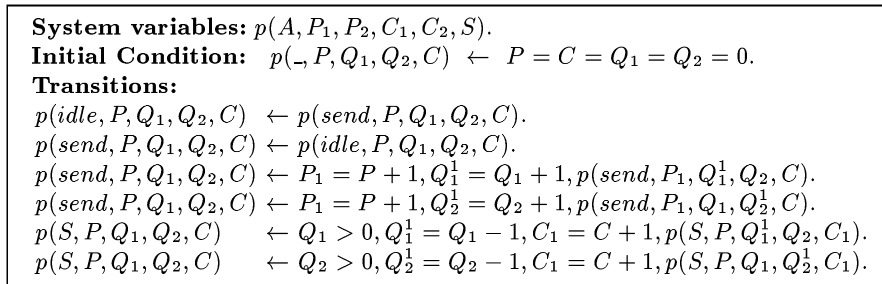


Fig. 16. Producer-consumer with unbounded buffer.



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:
$p(idle, P, Q_1, Q_2, C) \leftarrow p(send, P, Q_1, Q_2, C).$
$p(send, P, Q_1, Q_2, C) \leftarrow p(idle, P, Q_1, Q_2, C).$
$p(send, P, Q_1, Q_2, C) \leftarrow P_1 = P + 1, Q_1^1 = Q_1 + 1, p(send, P_1, Q_1^1, Q_2, C).$
$p(send, P, Q_1, Q_2, C) \leftarrow P_1 = P + 1, Q_2^1 = Q_2 + 1, p(send, P_1, Q_1, Q_2^1, C).$
$p(S, P, Q_1, Q_2, C) \leftarrow Q_1 > 0, Q_1^1 = Q_1 - 1, C_1 = C + 1, p(S, P, Q_1^1, Q_2, C_1).$
$p(S, P, Q_1, Q_2, C) \leftarrow Q_2 > 0, Q_2^1 = Q_2 - 1, C_1 = C + 1, p(S, P, Q_1, Q_2^1, C_1).$

Fig. 17. CLP-program for producers and consumers connected via an unbounded buffer.

graph of a program, all information involving manipulation of indexes of arrays. All remaining information will be abstracted away. In many cases the resulting system can be translated into a simple CLP program, as we will show with the help of a non trivial example: the insertion sorting algorithms

Insertion sorting. The procedure written in C of Fig. 18 implements the *insertion sorting* algorithm. It takes an array A and its right bound n as parameters and sorts the elements of A in increasing order. Our aim is to check that the procedure can not access the array A outside the interval $[0, n-1]$ (in C array indexing starts from 0). As anticipate before, the first step consists of extracting all information involving array indexes. The 'simple' CLP of Fig. 19(right) shows the resulting abstract flow graph. In the abstract flow graph we use the locations *entryA1*, *entryA2* and *entryA3* to keep track of the accesses to the array A in the original code. Note that the abstract flow graph has more possible states than the original program (e.g.,the condition $A[i] > x$ in the guard of the while is abstracted away). In other words, a property of the abstract graph is a conservative approximation for a property of the original program.

The requirement that the program does not violate the array bounds can be formulated as the safety property $AG(\neg(\textit{bounds are violated}))$. The potential violations for *insertion sorting* are given in Fig. 19(right). Since both the program and the properties are expressed using simple constraints, the analysis over the reals will give accurate results. A plain fixpoint computation (needed to check safety) will not terminate. Our model checker, however, proves the procedure correct by using the acceleration rule 2 of Fig. 10. Note that the use of accurate accelerations allow the detection of possible errors in the abstract graph (i.e., errors in the manipulation of array indexes in the original program).

7.3 Parameterized systems

We conclude the section dedicated to the examples presenting the analysis of parameterized systems called *broadcast protocols*.

Broadcast protocols [EN98] are systems composed of a finite but arbitrarily large number of processes that communicate by rendezvous (two processes exchange a message) or by broadcast (a process sends a message to all other processes).

```
                void InsertionSort(int* A, int n) {
init:             int i, k, x;
for:              for(k = 1; k < n; k++) {
entryA1:            x = A[k];
                    i = k−1;
while:              while (i >= 0 && A[i] > x) {
entryA2:              A[i + 1] = A[i];
                      i−−;
                    }
entryA3:            A[i + 1] = x;
                  }
end:            }
```

Fig. 18. Insertion sorting (left: program location).

System variables: $p(Location, K, N, I)$.
Initial Condition: $init \leftarrow p(init, K, N, I)$.
Transitions:

$p(init, K, N, I) \leftarrow p(for, K1, N, I), \quad K1 = 1.$
$p(for, K, N, I) \leftarrow p(entryA1, K, N, I), \quad K \leq N - 1.$
$p(entryA1, K, N, I) \leftarrow p(while, K, N, I1), \quad I1 = K - 1.$
$p(while, K, N, I) \leftarrow p(entryA2, K, N, I), \quad I \geq 0.$
$p(entryA2, K, N, I) \leftarrow p(while, K, N, I1), \quad I1 = I - 1.$
$p(while, K, N, I) \leftarrow p(entryA3, K, N, I), \quad I \leq -1.$
$p(entryA3, K, N, I) \leftarrow p(for, K1, N, I), \quad K1 = K + 1.$
$p(for, K, N, I) \leftarrow p(end, K, N, I), \quad K \geq N.$

$p(entryA1, K, N, I) \leftarrow K \geq N.$
$p(entryA1, K, N, I) \leftarrow K \leq -1.$
$p(entryA3, K, N, I) \leftarrow I \geq N - 1.$
$p(entryA3, K, N, I) \leftarrow I \leq -2.$
$p(entryA2, K, N, I) \leftarrow I \leq -1.$
$p(entryA2, K, N, I) \leftarrow I \leq -2.$
$p(entryA2, K, N, I) \leftarrow I \geq N - 1.$
$p(entryA2, K, N, I) \leftarrow I \geq N.$

Fig. 19. Left: CLP program for insertion sorting. Right: potential violations of array bounds.

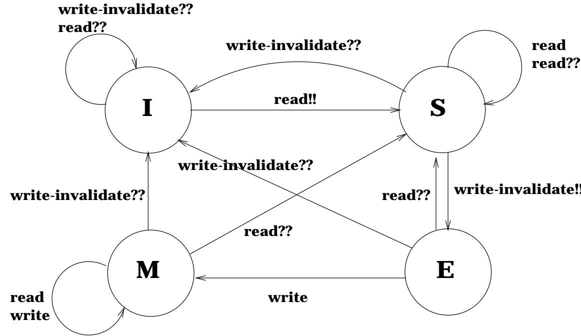


Fig. 20. MESI-protocol.

We consider the cache coherence protocol presented in [EN98]. Several processors interact with the main memory through a one-line cache. When a processor requires to copy the contents of its cache in the main memory all other processors must invalidate the contents of their local cache. We model this system as a collection of identical processes each described as in Fig. 20. In the states **I** and **S**, processes cannot write the content of the cache to the main memory, whereas only one process at a time can be in state **E** and can copy the cache line in main memory (action **write**). When a process in state **S** sends the broadcast **write-invalidate!!** to all other processes, and then moves to state **E**, the other processes react and move to the state **I**. The action **write-invalidate??** denotes the reception of the broadcast. Reading from the cache is preceded by the broadcast message **read!!**, as well. The remaining transitions denote internal actions of processes. The protocol must ensure mutual exclusion between readers and writers.

We model this protocol as the CLP program of Fig. 21. We use four variables I, S, E, and M to *count* the *number* of processes in the corresponding state. Each action corresponds to a re-allocation of the counters. Specifically, the second rule correspond to the **read!!** broadcast action, (i.e. one process sends the broadcast while being in state **I**, the other processes move to **S**), the third rule corresponds to the **write-invalidate!!**

$$
\begin{array}{lll}
init & \leftarrow p(I',S',E',M'), & I \geq 1, S=0, E=0, M=0. \\
p(I,S,E,M) & \leftarrow p(I',S',E',M'), & I \geq 1, S'=S+E+M+1, I'=I-1, E'=0, M'=0. \\
p(I,S,E,M) & \leftarrow p(I',S',E',M') & S \geq 1, S'=0, I'=S+M+E-1, M'=0, E'=0. \\
p(I,S,E,M) & \leftarrow p(I,S,E',M'), & E \geq 1, E'=E-1, M'=M+1. \\
p(I,S,E,M) & \leftarrow p(I,S,E,M), & S \geq 1. \\
p(I,S,E,M) & \leftarrow p(I,S,E,M), & M \geq 1.
\end{array}
$$

Fig. 21. Integer-system for the M.E.S.I. protocol 20.

broadcast, and the remaining rule to the **read** and **write** internal actions (only with **write** there is a reallocation of processes). Note that, in the initial configuration, the number of processes in state I is unspecified ($I \geq 1$).[5] The safety properties proved in [EN98] are: $AG(\neg(S \geq 1 \wedge M \geq 1))$ (readers and writes cannot access simultaneously the cache), $AG(\neg(M+E \geq 1))$ (at most one process can stay in M or E). These properties can be expressed as *upwards closed sets*. The reachability problem is decidable in this case [AČJT96,EFM99]. Upwards closed sets are fully characterized by the following class of constraints:

$$\phi \quad ::= \quad x_1 + \ldots + x_n \geq c \mid \phi \wedge \phi \mid true,$$

where c is a positive integer constant. Note that the constraints arising from the translation of broadcast protocol to CLP programs and the above properties are not simple constraints. However, in this case the relaxation int-real of the S_P operator is still accurate. In fact, it is easy to see that a ϕ-constraint is always satisfiable (both over $\mathbb{N}$ and $\mathbb{R}$). Futhermore, when P is a broadcast protocol, the class of ϕ-constrained facts is closed under application of S_P (see [DEP99]).

On the other hand the termination test (two sets of ϕ-constraints have the same denotations) over the reals is weaker than the termination test over the integers. For instance, $p(x,y) \leftarrow x+y \geq 1$ and $p(x,y) \leftarrow x \geq 1 \wedge y \geq 1$ have the same denotations over $\mathbb{N}$ but not over $\mathbb{R}$. Thus, the detection of a fixpoint (though guaranteed even in case of analisys over $\mathbb{R}$) might be delayed when using a model checker over reals.

Our model checker automatically proves both properties after few iterations (without use of accelerations). The state space for this type of verification problems suffers from a dramatic explosion even for small constants occurring in the initial set of unsafe states. A detailed account of efficient techniques for handling the state-explosion problem is given in [DEP99].

Performance. The execution times obtained for all examples described in this section are listed in Fig. 22. In Fig. 22, we also list the execution times for other examples: *selection*, *matrix multiplication*, and *circular* are programs extracted from C-programs that implement the selection sorting, row*column matrix multiplication and a shifting of elements in a circular array, respectively. Finally, the example *csm* is a parameterized system describing the central server model of [DEP99]. All examples can be found at the address: www.mpi-sb.mpg.de/~delzanno/clp.html. All the verification problems have been tested on a Sun Sparc Station 4, OS 5.5.1.

[5] Broadcast protocols can be viewed as an extension of Petri Nets where tokens can be dynamically re-distributed among the places.

8 Related Work

There have been other attempts to connect logic programming and verification, none of which has our generality with respect to the applicable concurrent systems and temporal properties. In [FV94], Fribourg and Veloso-Pexoto define the notion of *automata with constraints* and study their properties (e.g. language inclusion) through a representation as CLP programs. In [FR96], Fribourg and Richardson use CLP programs over *gap-order integer constraints* [Rev93] in order to compute the set of reachable states for a 'decidable' class of infinite-state systems. Constraints of the form $x = y + 1$ (as needed in our examples) are not gap-order constraints. In [FO97], Fribourg and Olsen study reachability for system with integer counters via a translation to CLP programs with integer constraints. They also propose a number of optimizations (e.g. *fusion* of transitions for Petri Nets) in order to accelerate the fixpoint computation. These approaches are restricted to safety properties.

In [Rau94], Rauzy describes a CLP-style extension of the propositional μ-calculus to finite-domain constraints, which can be used for model checking for *finite-state* systems. In [Urb96], Urbina singles out a class of $CLP(\mathbb{R})$ programs that he baptizes 'hybrid systems' without, however, investigating their formal connection with hybrid system specifications; note that liveness properties of timed or hybrid automata can *not* be directly expressed through fixpoints of the S_P operator (because the clauses translating time transitions may loop). In [GP97], Gupta and Pontelli describe runs of timed automata using the top-down operational semantics of CLP-programs (and not the fixpoint semantics).

Programs	C	ET	EN	ERT	ERN	AT	AN	ART	ARN
bakery	8	0.1	18	0.1	13	-	-	-	-
bakery3	21	6.3	157	6.1	109	-	-	-	-
bakery4	53	335.4	1698	253.2	963	-	-	-	-
ticket	6	↑	↑	↑	↑	0.9	15	1.0	13
bbuffer (1)	4	0.2	2	0.2	2	-	-	-	-
bbuffer (2)	4	0.0	2	0.0	2	-	-	-	-
ubuffer	6	↑	↑	↑	↑	3.0	16	1.7	6
insertion	9	↑	↑	↑	↑	0.5	19	0.6	17
selection	8	0.2	16	0.2	16	-	-	-	-
matrix mul.	29	1.5	80	2.1	78	-	-	-	-
circular	10	0.1	13	0.1	12	-	-	-	-
mesi	4	0.0	2	0.0	2	-	-	-	-
csm	9	38.2	27	58	25	-	-	-	-

Fig. 22. Benchmarks for the verification of safety: C=number of clauses, E=exact, A=approximation with widening, R=elimination of redundant facts, T=execution time, N=number of produced facts, ↑=non-termination, −=not needed. The execution time is given in seconds.

In [CP98], Charatonik and Podelski show that set-based analysis of logic programs can be used as an always terminating algorithm for the approximation of CTL properties for *pushdown processes*; (traditional) logic programs as considered in [CP98] are not suitable for translating general concurrent systems. In [RRR+97], a logic programming language based on *tabling* called XSB is used to implement an efficient local model checker for finite-state systems specified in a CCS-like value-passing language (see also [DDR+99]). The integration of tabling with constraints is possible in principle and has a promising potential.

As described in [LLPY97], constraints as symbolic representations of states are used in UPPAAL, a verification tool for timed systems [LPY97]. It seems that, for reasons of syntax, it is not possible to verify safety for our examples in the current version of UPPAAL (but possibly in an extension). Note that UPPAAL can check *bounded liveness* properties only, which excludes e.g. starvation freedom.

We will next discuss work on other verification procedures for integer-valued systems. In [BGP97,BGP98], Bultan, Gerber and Pugh use the Omega library [Pug91] for Presburger arithmetic as their implementation platform. Their work directly stimulated ours; we took over their examples of verification problems. The execution times (ours are about an order of magnitude shorter than theirs) should probably not be compared since we manipulate formulas over reals instead of integers; we thus add an extra abstraction for which in general a loss of precision is possible. In [BGL98], their method is extended to a composite approach (using BDDs), whose adaptation to the CLP setting may be an interesting task. In [CABN97], Chan, Anderson, Beame and Notkin incorporate an efficient representation of arithmetic constraints (linear and non-linear) into the BDDs of SMV [McM93]. This method uses an external constraint solver to prune states with unsatisfiable constraints. The combination of Boolean and arithmetic constraints for handling the interplay of control and data variables is a promising idea that fits ideally with the CLP paradigm and systems (where BBD-based Boolean constraint solvers are available).

In [BF99], Bérard and Fribourg show that the relaxation int-real for the computation of pre^* and $post^*$ of Petri Nets and Timed Automata with Counters is accurate. They consider *counter regions* formulas that here we called *simple* constraints. Proposition 2 generalizes their result in the following sense: is formulated for a wider class of systems (all systems that can be translated to simple CLP programs); it also states the accuracy of the *termination* test (i.e. the subsumption test between sets of facts) for the model checking procedure.

Our accelrations rules are related to Boigelot and Wolper's loop-first technique [BW94] for deriving 'periodic sets' as representation of infinite sets of integer-valued states for reachability analysis. As a difference, Boigelot and Wolper analyze cycles and nested cycles in the control graph to detect meta-transitions *before* and independently of their (forward) model checking procedure, whereas we construct new loops (which roughly are meta-transitions) *during* our model checking procedure and consider them only if we detect that they possibly lead to an infinite loop. It will be interesting to formulate their 'widening' in our setup and possibly extend it; note that a set is 'periodic' if it can be represented by an equational constraint with existential variables, e.g. $\exists y\, x = 2y$. Mixed int-reals costraint solvers might be useful (if not necessary) for manipulationg this type of constraints.

In [DEP99], Delzanno, Esparza, and Podelski discuss in the details the theoretical complexity of the analysis of broadcast protocols over integer arithmetics. In this paper we show that the relaxation int-real of the predecessor operator for broadcast protocols gives accurate

results, though the number of iterations may increase since the termination test over the reals becomes weaker.

This paper is an extension of the paper we presented at TACAS'99 [DP99].

9 Conclusion and Future Work

We have explored a connection between the two fields of verification and programming languages, more specifically between model checking and CLP. We have given a reformulation of safety and liveness properties in terms of the well-studied CLP semantics, based on a novel translation of concurrent systems to CLP programs. We could define a model checking procedure in a setting where a fixpoint of an operator on infinite sets of states and a fixpoint of the corresponding operator on their *implicit representations* can be formally related via well-established results on program semantics.

We have turned the theoretical insights into a practical tool. Our implementation in a CLP system is direct and natural. One reason for this is that the two key operations used during the fixpoint iteration are testing entailment and conjoining constraints together with a satisfiability test. These operations are central to the CLP paradigm [JM94]; roughly, they take over the role of read and write operations for constraints as first-class data-structures.

We have obtained experimental results for several example infinite-state systems over integers. Our tool, though prototypical, has shown a reasonable performance in these examples, which gives rise to the hope that it is useful also in further experiments. Its edge on other tools may be the fact that its CLP-based setting makes some optimizations for specific examples more direct and transparent, and hence experimentation more flexible. We note that some CLP systems, such as SICStus, provide support for building and integrating ad hoc constraint solvers.

As for future work, we believe that more experience with practical examples is needed in order to estimate the effect of different fixpoint evaluation strategies and different forms of constraint weakening for conservative approximations. We believe that after such experimentation it may be useful to look into more specialized implementations.

Acknowledgements. The authors would like to thank Stephan Melzer for pointing out the paper [BGP97], Christian Holzbaur for his help with the OFAI-CLP($\mathbb{R}$) library [Hol95], and Tevfik Bultan, Richard Gerber, Supratik Mukhophadyay and C.R. Ramakrishnan for fruitful discussions and encouragements.

References

[AČJT96] P. A. Abdulla, K. Čerāns, B. Jonsson, and Y.-K. Tsay. General Decidability Theorems for Infinite-state Systems. In *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science (LICS'96)*, pages 313–321. IEEE Computer Society Press, 1996.

[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 Fifth Annual Symposium on Logic in Computer Science (LICS'90)*, pages 428–439. IEEE Society Press, 1990.

[BF99] B. Bérard, and L. Fribourg. Reachability Analysis of (Timed) Petri Nets Using Real Arithmetic. In J. Baeten and S. Mauw, editors, *Proceedings of the Tenth International Conference on Concurrency Theory (CONCUR'99)*, Eindhoven, The Netherlands, volume 1664 of *LNCS*, pages 178-193. Springer, 1999.

[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)*, pages 113–123. ACM Press, 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 Ninth Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 400–411. Springer, 1997.

[BGP98] T. Bultan, R. Gerber, and W. Pugh. Model Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations and Experimental Results. Technical Report CS-TR-3870, UMIACS-TR-98-07, Department of Computer Science, University of Maryland, College Park, 1998.

[BGW+97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDD's. In *Proc. of Int. Static Analysis Symposium*, LNCS 1302, pages 172–186. Springer, Paris, September 1997.

[BLL+98] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi and C. Weise. New Generation of UPPAAL. In *Proceedings of the International Workshop on Software Tools for Technology Transfer.* Aalborg, Denmark, 12 - 13 July, 1998.

[Bul98] T. Bultan. *Automated Symbolic Analysis of Reactive Systems*, Ph.D. Thesis, Department of Computer Science, University of Maryland, College Park, August 1998.

[BW94] B. Boigelot and P. Wolper. Symbolic Verification with Periodic Sets. In David Dill, editor, *Proceedings of the Sixth International Conference on Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 55–67. Springer, 1994.

[BW98] B. Boigelot and P. Wolper. Verifying Systems with Infinite but Regular State Space. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the Tenth Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 88–97. Springer, 1998.

[Čer94] K. Čerāns. Deciding Properties of Integral Relational Automata. In *Proceedings of ICALP 94*, LNCS 820, pages 35-46. 1994.

[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 Ninth Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 316–327. Springer, 1997.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of Fourth the ACM Symposium on Principles of Programmming Languages (POPL '77)* pages 238-252. Los Angeles, January 1977.

[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.

[CGL92] E.M. Clarke, O. Grümberg and D.E. Long. Model checking and abstraction, In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages (POPL 92)*, pages 343–354. ACM Press, 1992.

[CJ98] H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis, and Presburger Arithmetics. In Alan J. Hu and M. Y. Vardi, editors, *Proceedings of the Tenth Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 268–279. Springer, 1998.

[CP98] W. Charatonik and A. Podelski. Set-based Analysis of Reactive Infinite-state Systems. In Bernhard Steffen, editor, *Proceedings of of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 358–375. Springer, 1998.

[Dam96] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD Thesis, Eindhoven University of Technology, 1996.

[DEP99] G.Delzanno, J.Esparza, and A.Podelski. Constraint-based Analysis of Broadcast Protocols. In J.Flum and M. Rodriguez-Artalejo, editors, *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL 99)*, LNCS, vol. 1683, pag. 50-66. Springer, September1999,

[DP99] G. Delzanno, A. Podelski. Model Checking in CLP. In W.Rance Cleaveland, editor, *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99), held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS '99)*, pages 223-239, Lecture Notes in Computer Science, vol. 1579, Springer, Amsterdam, The Nederlands, March 1999.

[DDR+99] Y. Dong, X. Du, Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S. A. Smolka, O. Sokolsky, E.W. Stark, and D. S.Warren. Fighting Livelock in the i-Protocol: A Comparative Study of Verification Tools . In W.Rance Cleaveland, editor, *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99), held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS '99)*, pages 74-88, Lecture Notes in Computer Science, vol. 1579, Springer, Amsterdam, The Nederlands, March 1999.

[Eme90] E. A. Emerson. Temporal and Modal Logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science, 1990.

[EN98] E. A. Emerson and K. S. Namjoshi. On Model Checking for Non-Deterministic Infinite-State Systems. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS '98)*, IEEE Computer Society Press, 1998.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the Verification of Broadcast Protocols. In *Proceedings of the 14th IEEE International Symposium on Logic in Computer Science (LICS '99).* IEEE Computer Society Press, 1999.

[FO97] L. Fribourg and H. Olsen. A Decompositional Approach for Computing Least Fixed Point of Datalog Programs with Z-counters. *Journal of Constraints*, 2(3-4):305–336, 1997.

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

[FV94] L. Fribourg and M. Veloso-Peixoto. Automates Concurrents Contraintes. Technique et Science Informatiques, 13(6):837-866, 1994.

[FS98] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! Technical Report LSV-98-4, Laboratoire Spécification et Vérification, Ecole Normale Supérieure de Cachan. April 1998. To appear in *Theoretical Computer Science*, 1999.

[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.

[GP97] G. Gupta and E. Pontelli. A Constraint Based Approach for Specification and Verification of Real-time Systems. In *Proceedings of the 18th IEEE Real Time Systems Symposium (RTSS'97).* IEEE Computer Society, 1997.

[Gra94] S. Graf. Verification of a distributed cache memory by using abstractions. In D.L. Dill editor, *Proceedings of Computer-aided Verification (CAV 94)*, LNCS 818, pages 207–219. Springer, 1994.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In *Proceedings of the 5th Conference on Computer-Aided Verification (CAV '93)*, Elounda (Greece), LNCS 697, pages 333-346. Springer, 1993.

[HH95] T. A. Henzinger and P.-H. Ho. A Note on Abstract Interpretation Strategies for Hybrid Au-

tomata. In P. Antsaklis, A. Nerode, W. Kohn, and S. Sastry, editors, *Proceedings of Hybrid Systems II*, volume 999 of *LNCS*, pages 252–264, 1995.

[HHWT97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a Model Checker for Hybrid Systems. In Orna Grumberg, editor, *Proceedings of the Ninth Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 460–463. Springer, 1997.

[Hol95] C. Holzbaur. OFAI CLP(Q,R), Manual, Edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[HPR97] N. Halbwachs, Y-E. Proy, and P. Roumanoff. Verification of Real-time Systems using Linear Relation Analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[JM94] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[KKR95] P. C. Kanellakis and G. M. Kuper and P. Z. Revesz. Constraint Query Languages. Journal of Computer and System Sciences, 51, pages 6–52, 1995.

[KM69] R. M. Karp and R. E. Miller. Parallel Program Schemata. Journal of Computer and System Sciences, 3, pages 147-195, 1969.

[Lam92] J. Lambert. A structure to decide reachability in Petri nets. Theoretical Computer Science, 99(1), pages 79–1044, 1992.

[LGS+94] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajiani, S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. Formal Methods in System Design, pages 1–35. Kluwer Academic, Boston 1994.

[LLPY97] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient Verification of Real-time Systems: Compact Data Structure and State-space Reduction. In *Proceedings of the 18th IEEE Real Time Systems Symposium (RTSS'97)*, pages 14–24. IEEE Computer Society, 1997.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LS97] D. Lesens and H. Saidi. Automatic Verification of Parameterized Networks of Processes by Abstraction. In *Proceedings of the International Workshop on Verification Infinite State Systems (INFINITY'97), available at the URL http://sunshine.cs.uni-dortmund.de/organization/pastE.html*, 1997.

[Mah95] M. J. Maher. Constrained dependencies. In Ugo Montanari, editor, *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP'95)*, Lecture Notes in Computer Science, pages 170–185, Cassis, France, 19–22 September 1995. Springer.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic, 1993.

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

[MR89] M. J. Maher and R. Ramakrishnan. Déjà Vu in Fixpoints of Logic Programs. In Ross A. Lusk and Ewing L. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming (NACLP'89)*, pages 963–980. MIT Press, 1989.

[Pod94] A. Podelski, editor. *Constraint Programming: Basics and Trends*, volume 910 of *LNCS*. Springer, 1994.

[Pug91] W. Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In A. Copeland MacCallum, editor, *Proceedings of the 4th Annual Conference on Supercomputing*, pages 4–13, Albuquerque, NM, USA, November 18-22, 1991. IEEE Computer Society Press.

[Rau94] A. Rauzy. Toupie: A Constraint Language for Model Checking. In Podelski [Pod94], pages 193–208.

[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. Ramakrishnan, 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 Ninth Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 143–154. Springer, 1997.

[RSS92] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Efficient Bottom-up Evaluation of Logic Programs. In P. De Wilde and J. Vandewalle, editors, *Computer Systems and Software Engineering: State-of-the-Art*, chapter 11. Kluwer Academic, 1992.

[Sch86] A. Schrijver, Alexander. Theory of Linear and Integer Programming, Wiley, Chichester;New York;Brisbane, 1986 (repr. 94).

[Sha93] U. A. Shankar. An Introduction to Assertional Reasoning for Concurrent Systems. *ACM Computing Surveys*, 25(3):225–262, 1993.

[SKR98] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A Comparison of Presburger Engines for EFSM Reachability. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the Tenth Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 280–292. Springer, 1998.

[Sri93] D. Srivastava. Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints. *Annals of Mathematics and Artificial Intelligence*, 8(3-4):315–343, 1993.

[SUM96] H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive Model Checking. In R. Alur and T. Henzinger, editors, *Proceedings of the Eighth Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 208–219. Springer, 1996.

[Urb96] L. Urbina. Analysis of Hybrid Systems in CLP(R). In Eugene C. Freuder, editor, *Proceed-*

ings of Principles and Practice of Constraint Programming (CP'96), volume 1118 of *LNCS*, pages 451–467. Springer, 1996.

[Wal96] M. Wallace. Practical Applications of Constraint Programming. *Constraints*, 1(1-2):139–168, 1996.

A Preliminaries on CLP

A CLP program [JM94] is nothing but a logic program where a given set of formulas (called constraints) are interpreted over a fixed domain. This way, specialized constraint solvers can be used to make resolution-based methods (for first order logic) more efficient.

Formally, a CLP program is a first order theory consisting of a universally quantifed conjunction of formulas called *clauses*. A clause has the form $A \leftarrow B_1 \wedge \ldots \wedge B_n$ where $n \geq 0$, and A (the head) and $B_1, \ldots, B_n$'s (that form the body) are atomic formulas. A *constraint* is a *finite* conjunction of atomic formulas (e.g., occurring in the body of a clause) built on a given set of constraint constructors. As anticipated before, constraints will be interpreted over a fixed domain and handled via a specialized inference engine called *constraint solver*. As an example, consider the two clauses $max(X,Y,X) \leftarrow X > Y$ and $max(X,Y,Y) \leftarrow X \leq Y$. In a CLP language defined over the domain of numbers, instead of giving a specification for $X \leq Y$ and $X > Y$, we can use a specialized solver for arithmetics to check their satisfiability.

In the following we will use $\mathcal{D}$ to denote the constraint domain of the CLP language taken into consideration. We say that a constraint c with variable in V is solvable in $\mathcal{D}$, namely $\mathcal{D} \models c\theta$, if there exists a valuation $\theta : V \rightsquigarrow D$, such that $c\theta$ evaluates to true in $\mathcal{D}$. A constraint c *entails* a constraint d if for each valuation θ, $\mathcal{D} \models c\theta \rightarrow d\theta$. Furthermore, given a program P we will indicate by $[P]_{\mathcal{D}}$ the set of formulas obtained by instantiating the variables of P with values from $\mathcal{D}$.[6] In the following we will use $\mathbf{t}$ to denote a list of terms $t_1, \ldots, t_n$.

A.1 Operational semantics

A clause $p(t_1, \ldots, t_n) \leftarrow B$ can be viewed as a *definition* for the predicate p. A goal, i.e., a conjunction $p_1(\mathbf{s_1}) \wedge \ldots p_m(\mathbf{s_m})$ of atomic formulas, can be viewed then as procedure invocations. The invocation of a procedure (resolution step) is solved as follows: i) replace a literal $p(s_1, \ldots, s_n)$ in the current goal with $s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge B$ (= denotes equality); ii) check that the constraint contained in the resulting goal-formula is satisfiable in $\mathcal{D}$ (using the constraint solver). Note that clauses can be selected non-deterministically. A *derivation* is then a sequence of goal formulas obtained via resolution steps. A successful computation is finite and ends with a constraint formula (the answer to the goal), meaning that the goal is a logical consequence of the program whenever the resulting constraint is satisfied. A ground resolution step is defined as follows: replace a (ground) literal A with the body of a ground (instance of a) clause whose head matches A. A *ground* derivation is a derivation obtained via ground resolution steps.

A successful ground computation is then a sequence of ground goals terminated by the empty goal, meaning that the goal is a logical consequence of the program.

[6] For simplicity, we assume that all constants are interpreted over $\mathcal{D}$.

A.2 Fixpoint semantics

The least model of a CLP program, defining its declarative semantics, can be defined as the fixpoint of an operator that computes the direct logical consequences of the program and of a given set of atomic formulas. In the following we will present its definition for the ground and for the non-ground case.

The *ground direct consequence operator* [JM94] is defined over collections of atomic formulas as follows:

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

T_P is monotonic and continuos w.r.t. set inclusion. The least fixpoint of T_P coincides with the least Herbrand model of P [JM94], and characterizes the set of atomic goals for which there exists a successful derivation.

The *non-ground direct consequences operator* S_P is defined over a collections of *facts*, i.e., of clauses of the form $p(\mathbf{x}) \leftarrow c$ where c is a constraint. A fact is an implicit representation of a set of ground atoms. Its definition is as follows:

$$S_P(I) = \{p(\mathbf{x}) \leftarrow c \mid p(\mathbf{x}) \leftarrow c', b_1, \ldots, b_n \ \in P \\ (a_i \leftarrow c_i) \in I, \ i : 1...n, \ n \geq 0 \\ \mathcal{D} \models c \leftrightarrow c' \wedge \textstyle\bigwedge_{i=1}^{n} (c_i \wedge a_i = b_i)\}.$$

The S_P operator finds a practical application in deductive databases where it is used for the *bottom-up* evaluation of queries, as opposite to the above mentioned *top-down* evaluation typical of logic programming systems.

The S_P operator is monotonic and continuous w.r.t. *set inclusion* of collections of facts [GDL95]. In [GDL95, JM94], the following properties are proved, under the assumption that the constraint domain $\mathcal{D}$ is *solution compact*:

- $T_P([I]_{\mathcal{D}}) = [S_P(I)]_{\mathcal{D}}$ for a set of facts I,
- $lfp(T_P) = \bigcup_{i=0}^{\omega} T_P^i(\emptyset)$,
- $lfp(S_P) = \bigcup_{i=0}^{\omega} S_P^i(\emptyset)$,
- $lfp(T_P) = [lfp(S_P)]_{\mathcal{D}}$,
- $\bigcap_{i=0}^{\omega} T_P^i(\mathcal{B}) = gfp(T_P)$, for $\alpha \geq \omega$.

Here, $\mathcal{B}$ (the Herbrand base) is the collection of all ground atomic formulas. In order to obtain a similar property for the greatest fixpoint of S_P, we need two extensions: i) we allow constraints to be *infinite* conjunctions; ii) we order collections of facts wrt. their denotations, i.e., $I \sqsubseteq J$ iff $[I]_\mathcal{D} \subseteq [J]_\mathcal{D}$. The lower bound for two collections of facts I and J is obtained then as follows:

$$I \wedge J = \{\ p(\mathbf{x}) \leftarrow \mathbf{x} = \mathbf{t}\ \wedge\ \mathbf{x} = \mathbf{s}\ \wedge\ c\ \wedge\ d \mid \\ p(\mathbf{t}) \leftarrow c \in I,\ p(\mathbf{s}) \leftarrow d \in J\ \}.$$

Note that $[I \wedge J]_\mathcal{D} = [I]_\mathcal{D} \cap [J]_\mathcal{D}$. The operator S_P is monotonic w.r.t. $\sqsubseteq$. Furthermore, it holds that

- $\mathit{gfp}(S_P) = \bigwedge_{i=0}^{\alpha} S_P^i(\mathcal{B}_S)$ for $\alpha \geq \omega$,
- $[\mathit{gfp}(S_P)]_\mathcal{D} = \mathit{gfp}(T_P)$.

where $\mathcal{B}_S$ is such that $[\mathcal{B}_S]_\mathcal{D} = \mathcal{B}$, e.g, $\mathcal{B}_S$ is the collections of facts $p(\mathbf{x}) \leftarrow \mathit{true}$ with p a predicate and $\mathbf{x}$ a vector of variables. Finally, note that hypothesis i is only necessary when $\mathit{gfp}(S_P)$ can not be computed in a finite number of steps.