Int J STTT (2001) 3: 250–270 / Digital Object Identifier (DOI) 10.1007/s100090100049

Constraint-based deductive model checking

Giorgio Delzanno, Andreas Podelski

Max-Planck-Institut für Informatik, Am Stadtwald, 66123 Saarbrücken, Germany; E-mail: {delzanno,podelski}@mpi-sb.mpg.de

Published online: 18 July 2001 –

Abstract. We show that constraint logic programming (CLP) can serve as a conceptual basis and as a practical implementation platform for the model checking of infinite-state systems. CLP programs are logical formulas (built up from constraints) that have both a logical interpretation and an operational semantics. Our contributions are: (1) a translation of *concurrent systems* (imperative programs) into CLP programs with the same operational semantics; and (2) a deductive method for verifying safety and liveness properties of the systems which is based on the logical interpretation of the CLP programs produced by the translation. We have implemented the method in a CLP system and verified well-known examples of infinite-state programs over integers, using linear constraints here as opposed to Presburger arithmetic as in previous solutions.

Keywords: Constraints – Logic programming – Verification – Model checking

1 Introduction

Automated verification methods can currently be applied to practical systems [47]. One reason for this success is that implicit representations of finite sets of states through Boolean formulas can be handled efficiently via BDDs [2]. 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 (e.g., see [1, 5, 11, 17, 36, 38, 45, 58]). 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., [50, 61]) can be used for the verification of infinite-state systems. The work reported in this paper investigates constraint logic programming (see [39]) 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 [57]. 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 *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* to the fragment of Horn clauses. This operator is made effective by using constrained facts to implicitly represent sets of ground atoms. Constrained facts can be manipulated symbolically via the operations defined on constraints (e.g., variable elimination, satisfiability, 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 [40, 53, 55] 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, e.g., see [5, 6, 9–12, 17, 28, 58]. The problem is undecidable for most classes of practical importance. Following [10, 11], 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-integer, 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 accommodated in our CLP implementation.

These abstractions allow us to solve the problems taken into consideration at acceptable cost. Furthermore, the experiments show that, perhaps surprisingly, the powerful (triple-exponential time in the worst-case) decision-procedure for Presburger arithmetic used in other approaches [6, 58] 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.

The paper is organized in the following way: in Sect. 2, we will introduce the formal setting of CLP, and the translation of concurrent systems to CLP programs. In Sect. 3, we will show how to express CTL properties in terms of CLP program semantics. In Sect. 4, we will turn the theory into practice, discussing a model-checking method. In Sect. 5, we will show how the method can be implemented using the features of existing CLP systems. In Sect. 6, we will present the techniques that can be used to analyze integer infinite-state systems using efficient constraint solving. In Sect. 8, we will present a number of case-studies. In Sect. 9, we will discuss related work. Finally, in Sect. 9, we will conclude the paper with the future perspectives of our work.

2 Translating concurrent systems into CLP

We take the bakery algorithm as an example of a concurrent program (for example, see [5, 48]):

begin $turn_1 := 0;\ \ turn_2 := 0;\ \ P_1 \mid\mid P_2$ **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:

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

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 Fig. 1 following the scheme in [57]. 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 Fig. 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 is all that one needs to understand this paper. A CLP program is simply a logical formula, namely a universally quantified conjunction of implications (like the one in Fig. 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

Control variables $p_1, p_2 : \{think, wait, use\}$
Data variables $turn_1, turn_2 : int$
Initial 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. 1. Concurrent system $\mathcal{S}$ specifying the bakery algorithm

$init \leftarrow Turn_1 = 0, Turn_2 = 0, p(think, think, Turn_1, Turn_2),$
$p(think, P_2, Turn_1, Turn_2) \leftarrow Turn_1' = Turn_2 + 1,\ p(wait, P_2, Turn_1', Turn_2),$
$p(wait, P_2, Turn_1, Turn_2) \leftarrow Turn_1 < Turn_2,\quad p(use, P_2, Turn_1, Turn_2),$
$p(wait, P_2, Turn_1, Turn_2) \leftarrow Turn_2 = 0,\quad p(use, P_2, Turn_1, Turn_2),$
$p(use, P_2, Turn_1, Turn_2) \leftarrow Turn_1' = 0,\quad p(think, P_2, Turn_1', Turn_2),$
... symmetrically for Process 2

Fig. 2. CLP program $P_{\mathcal{S}}$ for the concurrent system $\mathcal{S}$ in Fig. 1

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 Fig. 1 are exactly the ground derivations of the CLP program $P_{\mathcal{S}}$ in Fig. 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 $\tilde{x}$ for the tuple of variables $\langle x_1, \ldots, x_n\rangle$ and $\tilde{d}$ for the tuple of values $\langle d_1, \ldots, d_n\rangle$. We denote validity of a first-order formula ψ with respect to a structure $\mathcal{D}$ and an assignment α by $\mathcal{D}, \alpha \models \psi$. As usual, $\alpha[\tilde{x} \mapsto \tilde{d}]$ denotes an assignment in which the variables in $\tilde{x}$ are mapped to the values in $\tilde{d}$. In the examples of Sect. 8, 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 [57]) is a triple $\langle V, \Theta, e\rangle$ such that

- V is the tuple $\mathbf{x}$ of control and data variables.
- Θ is a formula over V called the *initial condition*.
- 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

$$\tau = \{\langle \tilde{d}, \tilde{d}'\rangle \mid \exists\, \alpha \text{ s.t. } \mathcal{D}, \alpha[\tilde{x} \mapsto \tilde{d}] \models \psi, \mathcal{D}, \alpha[\tilde{x} \mapsto \tilde{d}, \tilde{x}' \mapsto \tilde{d}'] \models \phi, \langle \psi, \phi\rangle \in e\}.$$

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 [39]). We introduce p

for a dummy predicate symbol with arity n, and *init* for a predicate with arity 0[1]

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 $\mathcal{S} = \langle V, \Theta, e \rangle$ and V is the tuple of variables $\mathbf{x}$.

$$P_{\mathcal{S}} = \{p(\mathbf{x}) \leftarrow \psi \wedge \phi \wedge p(\mathbf{x}') \mid \langle \psi, \phi \rangle \in e\} \cup \{init \leftarrow \Theta \wedge p(\mathbf{x})\}$$

The *direct consequence operator* T_P associated with a CLP program P (see [39]) 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 $p(\tilde{x}) \leftarrow \psi \wedge \phi \wedge p(\tilde{x}')$ of $P_{\mathcal{S}}$ corresponds to the event $\langle \psi, \phi \rangle$. Its instances are of the form $p(\tilde{d}) \leftarrow p(\tilde{d}')$ where $\mathcal{D}, \alpha[\tilde{x} \mapsto \tilde{d}, \tilde{x}' \mapsto \tilde{d}'] \models \psi \wedge \phi$. Thus, they correspond directly to the pairs $\langle \tilde{d}, \tilde{d}' \rangle$ 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.

- *Base.* The thesis immediately follows by noting that the initial states are instances of the body of the *init*-clause.
- *Inductive step.* Let us assume now that the thesis holds for a state-derivation $\tilde{d}_0 \tilde{d}_1 \ldots \tilde{d}_i$. If $\langle \tilde{d}_i, \tilde{d}_{i+1} \rangle \in \tau$ there exists a clause in the translation such that its set of instances contains $p(\tilde{d}_i) \leftarrow p(\tilde{d}_{i+1})$. Furthermore, by inductive hypothesis, $p(\tilde{d}_0)p(\tilde{d}_1) \ldots p(\tilde{d}_i)$ is a ground derivation. Thus, by applying a resolution step we obtain the new sequence $p(\tilde{d}_0)p(\tilde{d}_1) \ldots p(\tilde{d}_i)p(\tilde{d}_{i+1})$.

The converse can be proved by induction on the length of a derivation.

[1] Note that, for example, $p(think, P_2, Turn_1, Turn_2) \leftarrow \ldots$ 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.

Point (ii) follows directly from the definition of pre_S and T_{P_S}. □

As in Shankar's model, if a state $\tilde{d}$ has no predecessors, then the operator $T_{P_{\mathcal{S}}}$ returns the empty set when applied to the singleton consisting of the atom $p(\tilde{d})$. In fact, if $\tilde{d}$ has no predecessors, there are no clauses of $P_{\mathcal{S}}$ whose antecedent can be instantiated with $\tilde{d}$. Thus, the set of logical consequences of the program $P_{\mathcal{S}}$ union the fact $p(\tilde{d})$ is the empty set. (Remember that B is a logical consequence of P union A if and only if every model for P and A is a model for B.)

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

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

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 [24], 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. An instance of a constrained fact $p(\mathbf{x}) \leftarrow c$ is of the form $p(\mathbf{d}) \leftarrow true$ where the assignment $\alpha[\mathbf{x} \mapsto \tilde{d}]$ satisfies c. Furthermore, note that $p(\mathbf{d}) \leftarrow true$ is equivalent to 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, for example, $EF(p_1 = p_2 = use)$ instead of $EF(F_{mut})$.

As an example of a safety property, consider *mutual exclusion* for the concurrent system $\mathcal{S}$ in Fig. 1 ("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 pro-

gram $P_{\mathcal{S}} \oplus F_{mut}$ that we obtain from the union of the CLP Program $P_{\mathcal{S}}$ in Fig. 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 Fig. 3. We obtain $P_{\mathcal{S}} \oslash F_{starv}$ from the CLP program $P_{\mathcal{S}}$ by a transformation with respect to the following set of two constrained facts:

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

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).

We now 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:

$$P \oplus \mathcal{F} = \ P \cup \mathcal{F}$$

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

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 hold for all sets of constrained facts $\mathcal{F}$:

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

Proof. From the fixpoint characterizations of CTL properties (see [24]) we know that

$$EF(\mathcal{F}) = \mu Z.\mathcal{F} \cup EX(Z)$$

$$EG(\mathcal{F}) = \nu Z.\mathcal{F} \cap EX(Z),$$

where $EX(Z) \equiv pre_S(Z)$. From Theorem 1, using the operators of Def. 2, the CLP programs $P_{\mathcal{S}} \oplus \mathcal{F}$ and $P_{\mathcal{S}} \oslash \mathcal{F}$ are such that

$$T_{P_{\mathcal{S}} \oplus \mathcal{F}}(Z) = pre_S(Z) \cup \mathcal{F},$$

$$T_{P_{\mathcal{S}} \oslash \mathcal{F}}(Z) = pre_S(Z) \cap \mathcal{F}.$$

As a consequence, we have that $EF(\mathcal{F}) = lfp(T_{P_{\mathcal{S}} \oplus \mathcal{F}})$ and $EG(\mathcal{F}) = gfp(T_{P_{\mathcal{S}} \oslash \mathcal{F}})$. □

By duality, we have that $AF(\neg\, \mathcal{F})$ is the complement of $gfp(T_{P \oslash \mathcal{F}})$ and $AG(\neg\, \mathcal{F})$ is the complement of $lfp(T_{P \oplus \mathcal{F}})$. 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 $AG(\neg\, \mathcal{F})$ if and only if the atom '*init*' is not in the least fixpoint for the CLP program $P_{\mathcal{S}} \oplus \mathcal{F}$.

(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 $P_{\mathcal{S}} \oplus (F_1 \wedge \mathcal{F}')$, where F' is a set of constrained facts denoting the greatest fixpoint for the CLP program $P_{\mathcal{S}} \oslash F_2$.

$init$	$\leftarrow\ Turn_1 = 0, Turn_2 = 0,$	$p(think, think, Turn_1, Turn_2),$
$p(think, P_2, Turn_1, Turn_2)$	$\leftarrow\ Turn'_1 = Turn_2 + 1,$	$p(wait, P_2, Turn'_1, Turn_2),$
$p(wait, P_2, Turn_1, Turn_2)$	$\leftarrow\ Turn_1 < Turn_2,$	$p(use, P_2, Turn_1, Turn_2),$
$p(wait, P_2, Turn_1, Turn_2)$	$\leftarrow\ Turn_2 = 0,$	$p(use, P_2, Turn_1, Turn_2),$
$p(wait, think, Turn_1, Turn_2)$	$\leftarrow\ Turn'_2 = Turn_1 + 1,$	$p(wait, wait, Turn_1, Turn'_2),$
$p(wait, wait, Turn_1, Turn_2)$	$\leftarrow\ Turn_2 < Turn_1,$	$p(wait, use, Turn_1, Turn_2),$
$p(wait, wait, Turn_1, Turn_2)$	$\leftarrow\ Turn_1 = 0,$	$p(wait, use, Turn_1, Turn_2),$
$p(wait, use, Turn_1, Turn_2)$	$\leftarrow\ Turn'_2 = 0,$	$p(wait, think, Turn_1, Turn'_2),$
$p(think, think, Turn_1, Turn_2)$	$\leftarrow\ Turn'_2 = Turn_1 + 1,$	$p(think, wait, Turn_1, Turn'_2),$
$p(think, wait, Turn_1, Turn_2)$	$\leftarrow\ Turn_2 < Turn_1,$	$p(think, use, Turn_1, Turn_2),$
$p(think, wait, Turn_1, Turn_2)$	$\leftarrow\ Turn_1 = 0,$	$p(think, use, Turn_1, Turn_2),$
$p(think, use, Turn_1, Turn_2)$	$\leftarrow\ Turn'_2 = 0,$	$p(think, think, Turn_1, Turn'_2)$

Fig. 3. The CLP program $P_{\mathcal{S}} \oslash F_{starv}$ for the concurrent system $\mathcal{S}$ in Fig. 1

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 $F \wedge F' = \{p(\mathbf{x}) \leftarrow c_1 \wedge c_2 \mid p(\mathbf{x}) \leftarrow c_1 \in F,\ p(\mathbf{x}) \leftarrow c_2 \in F',\ c_1 \wedge c_2 \text{ is satisfiable in } \mathcal{D}\}$.

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 [11]: "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 [11]).

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 of constrained facts, whose application $S_P(F)$ is effectively computable. 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 $p(x'_1, \ldots, x'_n) \leftarrow c[x'_1/x_1, \ldots, x'_n/x_n] \in F$.)

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

If P also contains 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 [31], where its connection to the ground semantics is also 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 $gfp(T_P) = [gfp(S_P)]_{\mathcal{D}}$. Thus, Theorem 2 leads to the characterization of CTL properties through the S_P operator via:

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

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 a standard fixpoint iteration of S_P operators for the specific programs P according to the fixpoint definition of the CTL properties to be computed (e.g., see 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.

4.1 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 [59]) and local subsumption is polynomial [59]. An abstract study of the complexity of local versus full subsumption based on the CLP techniques can be found in [46]; he shows that (full) subsumption is co-NP-hard unless it is equivalent to local subsumption. Sufficient conditions that guarantee the termination of backward reachability algorithm using local subsumption are given in [1]. Similar ideas are used in other model checkers such as HyTech [36] as well as in constraint databases [59], since they perform well on practical examples.

4.2 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 [49].

5 Implementation in a CLP system

In this section, we present a possible implementation of the previously described model-checking procedures using as platform the CLP system of Sicstus Prolog [37]. The implementation is based on the main features of CLP: constraint solving, unification of terms, meta-programming, and dynamic manipulation of the program database.

Up to now, we have used CLP programs as a mathematical model for transition systems. Existing CLP systems, however, adopt incomplete strategies to find a successful derivation for a goal. Specifically, CLP programs are executed following the *left-to-right* selection order for the literals of a goal. Thus, 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 (from top to bottom). Finally, when a subgoal fails the interpreter tries to find another possible derivation by selecting in backtracking other possible choices for the subgoals executed so far. In this section, we will use the syntax A*:-*$B_1, \ldots, B_n$ to denote clauses of CLP programs executed in accord with the previous strategies. We will use this syntax to describe the CLP programs used to implement the model-checking procedures.

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.

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. More precisely, each clause $p(\mathbf{t}) \leftarrow c, p(\mathbf{t}')$ of a program is stored as the fact $rule(p(\mathbf{t}), p(\mathbf{t}'), c)$, whereas each state $p(\mathbf{t}) \leftarrow c$ is stored as the fact $fact(p(\mathbf{t}), c, i)$, where i is a number denoting the iteration step in which the fact has been added to the database. This way, we can use the built-in efficient database operations to retrieve the data during the computation. In fact, in Sicstus Prolog the internal database is implemented as a hashing table indexed on the functor name and on the first argument of the head of a clause.

Let us first consider the computation of a safety property. The CLP implementation of the least fixpoint computation needed to compute $EF(F)$ is shown in Fig. 4.

The predicate *lfp* of Fig. 4 loads the transition system P and the initial set of constrained facts F in the program database (using CLP built-in primitives). Then, it starts a loop (defined through the recursive predicate *least_fixpoint*) used to compute the set of states that are backward reachable from F.

In each iteration of the loop we need to apply S_P exhaustively to the set of *already visited* states, say Φ, stored as constrained facts in the database. To achieve this effect, we proceed as follows:

(1) We first define a predicate that selects non-deterministically a rule and a fact from the database, applies S_P to them and, finally, checks whether or not the resulting constrained fact has to be added to the database. We call this predicate *apply_and_add_to_lfp*. We will discuss its implementation in a few lines.
(2) We apply the built-in predicate *bag_of* (that generates all derivations of the goal passed as its argument) to the predicate *apply_and_add_to_lfp*. This way, we add to the database *all* predecessors of the current set Φ of constrained facts.

The first clause of the predicate *least_fixpoint* of Fig. 4 implements point (2). The built-in predicate '!' (called cut) has the following effect: the second clause of *least_fixpoint* is selected if and only if the predicate *apply_and_add_to_lfp* fails, i.e., the current set of constrained fact has no *new* predecessors.

Let us now go back to the the implementation of the predicate *apply_and_add_to_lfp*. Its code together with the code of the auxiliary predicates *apply_Sp* and *locally_subsumed* is shown in Fig. 5. The predicate *apply_and_add_to_lfp* makes use of the the predicate *apply_Sp* to generate one of the possible predecessor facts, say $fact(A, C)$. At this point, using the predicate *locally_subsumed*, it checks that $fact(A, C)$ is not subsumed by any of the constrained facts in the current database. The predicate *fact_subsumed* in the definition of *locally_subsumed* (see Fig. 5) can be implemented

```
least_fixpoint(Current):-
  bag_of(apply_and_add_to_lfp(Current)),!,
  Next is Current+1,
  least_fixpoint(Next).

least_fixpoint(_):-
  write('Least fixpoint reached').

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

Fig. 4. Main loop for least fixpoint

```
locally_subsumed(fact(A,C),Current):-
  fact(A,D,Current),
  fact_entail(fact(A,C),fact(A,D)),!.

apply_Sp(Current,fact(H,C2)):-
  rule(H,B,C),
  fact(B,D,Current),
  satisfiable(C ∧ D,C1),
  variable_elimination(C1,H,C2).

apply_and_add_to_lfp(Current):-
  apply_Sp(Current,fact(A,C)),
  not(locally_subsumed(fact(A,C),_)),
  Next is Current+1,
  assert(fact(A,C,Next)).
```

Fig. 5. Application of the S_P operator

using the predicate *entail* defined over constraints. If $fact(A, C)$ passes this test, it is added to the database. Note that, with this definition we implement the *local* subsumption test we mentioned in the previous section.

The predicate *apply_Sp* in Fig. 5 implements an application of the S_P operator to a clause and a fact selected from the database.

After checking satisfiability of the constraint obtained conjoining the constraint of the selected clause with that of the selected fact, the solved form is projected over the variables contained in the head of the clause. Note that constraints are treated 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 parameter *Current* of the predicate *apply_Sp* denotes the index of the current fixpoint iteration. It is passed to the predicate *apply_Sp* simply because at step i we need only to compute the predecessors of constrained facts with the same index i. In addition, note that in the predicate *apply_Sp* we select only facts whose head *unifies* with the body of the selected clause.

Example 1. Let us consider a transition system with the following three clauses:

$init \leftarrow p(x, y), x = 0, y = 0.$
$p(x, y) \leftarrow p(x', y), x' = x + 1.$
$p(x, y) \leftarrow p(x, y'), y \geq 1, y' = y - 1.$

In order to compute $EF(x \geq 1 \wedge y \geq 0)$ we invoke the predicate *lfp*; *lfp* initializes the database as follows: (Note, we use Sicstus Prolog notation for constraints, e.g., $\{x \geq 0, y \geq 0\}$ represents the conjunction $x \geq 0 \wedge y \geq 0$.)

$rule(init, p(x, y), \{x = 0, y = 0\}).$
$rule(p(x, y), p(x_1, y), \{x \geq 0, y \geq 0, x_1 = x + 1\}).$
$rule(p(x, y), p(x, y_1), \{x \geq 0, y \geq 1, y_1 = y - 1\}).$

$fact(p(x, y), \{x \geq 1, y \geq 0\}, 0).$

In the first iteration *apply_Sp* has two possible derivations. The first derivation is obtained selecting the second *rule* and the single constrained *fact* in the initial database. After some calculations, it will give us the new fact $fact(p(x, y), \{x \geq 0, y \geq 0\})$ that is not subsumed by the fact stored in the database. Thus, the newly computed fact can be added to the database. The second possible derivation is obtained using the third rule, and it will give us the new fact $fact(p(x, y), \{x \geq 1, y \geq 1\})$. Clearly, this fact is subsumed by the initial fact, thus its information is redundant and we do not need to add it to the database. Note that the first rule cannot be applied to the initial fact. In fact, the conjunction of their constraints, namely $x \geq 1, y \geq 0, x = 0, y = 0$, is not satisfiable. Based on this observation (and from the code of Fig. 5 and Fig. 4), the invocation $bag\_of(apply\_and\_add\_to\_lfp(0))$ yields the following new database configuration:

$fact(p(x, y), \{x \geq 1, y \geq 0\}, 0).$
$fact(p(x, y), \{x \geq 0, y \geq 0\}, 1).$

In the next step *apply_Sp* may generate three new constrained facts: $fact(init, true)$ (using the first rule and the second fact); $fact(p(x, y), \{x \geq 0, y \geq 0\})$ (using the second rule and the second fact); and $fact(p(x, y), \{x \geq 0, y \geq 1\})$ (using the third rule and the second fact). However, the last two facts are subsumed by the second fact in the current database. Thus, we add only $fact(init, true)$, and we get the following configuration of the database:

$fact(p(x, y), \{x \geq 1, y \geq 0\}, 0).$
$fact(p(x, y), \{x \geq 0, y \geq 0\}, 1).$
$fact(init, true, 2).$

At this point, *apply_Sp* will generate redundant facts only. This implies that *apply_and_add_to_lfp* fails and that the main loop terminates. The set of constrained facts stored in the database represents the set of states that satisfy the property $EF(x \geq 1 \wedge y \geq 0)$. □

We use similar techniques to implement the computation of the greatest fixpoint of S_P used, e.g., for EG-properties. The Prolog code is shown in Fig. 6.

The top level predicate *gfp* in Fig. 6 asserts (via the predicate *assert_program_for_gfp*) the input transition system together with the fact $p(\mathbf{x}) \leftarrow true$ (representing all possible states) in the database. Then, it invokes a loop that, at each step, computes all derivations for *apply_and_add_to_gfp*. This predicate differs from *apply_and_add_to_lfp* in that newly inferred facts are compared only with facts computed at the same iteration step. In fact, by monotonicity of S_P, the denotations of the set of facts computed at step $i + 1$ are always contained in the denotations of those computed at step i. This also implies that we can use the local subsumption test (indicated *subsumed(Current,Next)* in Fig. 6) between the set of facts with index i and the set of the facts with index $i + 1$ as termination test for the greatest fixpoint computation.

```
apply_and_add_to_gfp(Current):-
  apply_Sp(Current,fact(A,C)),
  Next is Current+1,
  not(locally_subsumed(fact(A,C),Next)),
  assert(fact(A,C,Next)).


greatest_fixpoint(Current):-
  bag_of(apply_and_add_to_gfp(Current)),!,
  Next is Current+1,
  not(subsumed(Current,Next)),
  greatest_fixpoint(Next).

greatest_fixpoint(_):-
  write('Greatest fixpoint reached').

gfp(P):-
  assert_program_for_gfp(P),
  greatest_fixpoint(0).
```

Fig. 6. Main loop for greatest fixpoint

A full CTL model checker can be obtained by combining this procedure 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 sketched above in SICStus Prolog 3.7.1 using the arithmetic constraint solver CLP(Q,R) [37] and the Boolean constraint solver (which are implemented with BDDs). In the following section we will report on experimental results obtained by analyzing different types of infinite-state systems.

6 Infinite-state integer systems

The verification problem for systems with (unbounded) integer values is receiving increasing attention; e.g., see [5, 6, 9–12, 17, 28, 58]. The problem is undecidable for most classes of practical importance. So what can you do? There are basically two answers: (1) give a possibly nonterminating algorithm that terminates for useful examples. This is the approach followed, for example, by [7, 11]; (2) give a semi-test that yields the definite answer for useful examples (the other answer being 'don't know'), e.g., see [5, 16, 19, 33–35, 38, 43].

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 lose information with respect to 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 frustrating. 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 [14], 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 accommodated 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 integer-real.

6.1 Relaxation

In this section we investigate the integer-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). [2]

The relaxation from integers to reals stems from linear programming (e.g., see [56]). 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 versus NP-hard for the satisfiability test) as well as practically (e.g., the variable

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

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 [51]), there exist many highly optimized constraint systems over reals, general-purpose such as CLP(R) and special purpose such as Uppaal [8] or Hytech [36], which one would like to exploit for model checking (simple) concurrent systems over integers.

In the rest of the section, we will define the class of CLP programs for which we can prove that the relaxation integer-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*[3]) is built up according to the following grammar (where c is an integer constant, and x and y are (primed or unprimed) variables ranging over positive integers):

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

Note that constraints like $x + y \geq c$ are not simple. *Simple* CLP programs result 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 $x \leq y + c \ \wedge \ x \geq y + c$.

The reachability problem is undecidable for the whole class of systems that can be described using simple constraints. In fact, two counter machines are part of this class. On the other hand, vector addition systems [41] (a.k.a. Petri Nets), and integral relational automata [12] are two examples of systems whose translation in CLP gives rise to simple CLP programs. The reachability problem is decidable for these classes (e.g., see [12, 42]). Other examples are multi-clocks automata [17] and gap-order automata [28].

The above-mentioned decidability results are related to the general results for verification problems of infinite-state systems given in [1, 30]. The communication protocols considered in [5, 6, 58], such as the *bakery* algorithm of Sect. 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 subsets of both the domains $\mathbb{N}$ and $\mathbb{R}_+$ of integers and positive reals, respectively. More precisely, given a set F of constrained facts, $[F]_{\mathbb{N}}$ denotes the set of instances of the facts in F whenever the corresponding constraints are interpreted over $\mathbb{N}$; $[F]_{\mathbb{R}_+}$ denotes the set of instances of F whenever the constraints are interpreted over the *positive reals*. For instance, if $F = \{p(x, y) \leftarrow x \leq y\}$ then $[F]_{\mathbb{N}} = \{p(0,0), p(0,1), p(0,2), \dots\}$, whereas $[F]_{\mathbb{R}_+} = \{p(0,0), p(0.1,1), p(0.2,2), \dots\}$. Clearly, it always holds that $[F]_{\mathbb{N}} \subseteq [F]_{\mathbb{R}_+}$. Furthermore, given a CLP program P, the operator $S_{P,\mathbb{N}}$ is obtained from the general definition of S_P of Sect. 5, by taking $\mathbb{N}$ as domain for the constraints. Formally, $S_{P,\mathbb{N}}$ is defined as follows:

$$\begin{aligned} S_{P,\mathbb{N}}(F) = \{p(\mathbf{x}) \leftarrow c_1 \wedge c_2 \mid\ & p(\mathbf{x}) \leftarrow c_1 \wedge p(\mathbf{x}') \ \in P, \\ & p(\mathbf{x}') \leftarrow c_2 \ \in F, \\ & c_1 \wedge c_2 \ \text{ is satisfiable in } \mathbb{N}\} \end{aligned}$$

Similarly, the operator $S_{P,\mathbb{R}_+}$ is obtained taking $\mathbb{R}_+$ as domain. Formally, $S_{P,\mathbb{R}_+}$ is defined as follows:

$$\begin{aligned} S_{P,\mathbb{R}_+}(F) = \{p(\mathbf{x}) \leftarrow c_1 \wedge c_2 \mid\ & p(\mathbf{x}) \leftarrow c_1 \wedge p(\mathbf{x}') \ \in P, \\ & p(\mathbf{x}') \leftarrow c_2 \ \in F, \\ & c_1 \wedge c_2 \ \text{ is satisfiable in } \mathbb{R}_+\} \end{aligned}$$

Note that in the previous definitions the primed variables are implicitly existentially quantified in the body of the facts computed via S_P. Thus, every constrained atom $p(\mathbf{x}) \leftarrow c \in S_{P,\mathbb{N}}(F)$ for some F has the same denotations as $p(\mathbf{x}) \leftarrow \exists x' \in \mathbb{N}.c$ whenever the variable x' do not occur in $\mathbf{x}$. Similarly, $p(\mathbf{x}) \leftarrow c \in S_{P,\mathbb{N}}(F)$ for some F is equivalent to $p(\mathbf{x}) \leftarrow \exists x' \in \mathbb{R}_+.c$ whenever the variable x' does not occur in $\mathbf{x}$.

We show next that simple constraints enjoy special properties when the satisfiability and entailment test, and variable elimination are *relaxed* from integers-to reals, i.e., we execute all operations over the domain of positive reals instead that over the domain of integers.

Proposition 1 (Relaxation of constraint operators). The tests of satisfiability and entailment and of variable elimination over *simple* constraints yield the same results when interpreted over $\mathbb{N}$ and over $\mathbb{R}_+$, i.e., the relaxation integer-reals of these operations is accurate.

Proof. We consider the three operations separately.
Satisfiability. We first show that the satisfiability test is invariant under the relaxation integer-real. There are different ways to prove this result.

Given $r \in \mathbb{R}_+$, let $\mathit{floor}(r)$ be the greatest positive integer k such that $k \leq r$. Given $r_1, r_2 \in \mathbb{R}_+$ and an integer c, we note that if the inequality $r_1 - r_2 \leq c$ holds, then $k_1 - k_2 \leq c$ holds for $k_i = \mathit{floor}(r_i)$, $i : 1, 2$. Thus, if a simple constraint has the solution that maps the variable x_i to the positive real number r_i for $i : 1, \dots, n$, then it has the solution that maps x_i to $\mathit{floor}(r_i)$. In other words, a simple constraint has a solution over $\mathbb{N}$ if and only if it has a solution over $\mathbb{R}_+$. Another possible proof is based on the observation that a simple constraint φ is also *totally unimodular* (see [56] for a definition of this notion). This property implies the thesis.
Elimination. The proof that variable elimination over

[3] In [3], they are called *counter regions* formulas.

$\mathbb{R}_+$ for simple constraints gives accurate results has been given by Bérard and Fribourg in [3].

Entailment. To show that the entailment test is invariant under the relaxation, we simply note that the negation of an *atomic* (i.e., without $\wedge$) simple constraint is still a simple constraint. Now, the thesis follows by noting that the test $entail(C, D_1 \wedge D_2)$ can be reduced to the tests $not(satisfiable(C \wedge \neg D_1))$ and $not(satisfiable(C \wedge \neg D_2))$, i.e., to two satisfiability tests for simple constraints. □

We can now use the previous result to show that, whenever P is a simple CLP program, the use of $S_{P,\mathbb{N}}$ and $S_{P,\mathbb{R}_+}$ has the same effect.

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 yields 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. Follows from definition of $S_{P,\mathbb{N}}$ and $S_{P,\mathbb{R}_+}$ and from 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 *subsumption* 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.

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 [3], Bérard and Fribourg apply the relaxation integer-real to Petri Nets and apply HyTech for checking invariants of their examples. Finally, note that Proposition 2 holds for any CLP program defined over *simple constraints* (i.e., for CLP programs with more than one literal in the body).

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.

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. Basically, our idea is to give a number of sufficient conditions under which accelerations operators inspired to the widening operator proposed by Cousot and Halbwachs [15] yield accurate results.

More formally, our techniques are based on the following ideas: 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 $P \oplus F$. In many cases, however, it is possible to use stronger inference rules that allow us to *saturate* the set of logical consequences of $P \oplus F$ in one step. For instance, 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,

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

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 fixpoint of S_P (i.e., they will be applied (if possible) at each iteration). The correctness of the method follows by proving the soundness of the resulting deduction system. The inference rules are shown in Fig. 7. In Fig. 7, P is be a CLP program and F is a set of facts. Furthermore, given two first-order formulas F_1 and F_2 $F_1 \models F_2$ indicates that F_2 is a logical consequence of F_2, i.e., a model for F_1 is a model for F_2. Intuitively, in all rules we guess the 'direction of growth' of a given constraint during the computation of the fixpoint. Rule 1 and 2 are devised for constraints of the form $x \leq y + c$, and $x \leq c$, respectively, whereas Rule 3 is devised for constraints having form $x = c$. Similar rules can be defined for the symmetric cases (e.g., taking $x \geq c$ and $c_x < 0$ in Rule 2 in the hypothesis of Rule 2). Note that, in the first condition of every rule, the selected clause is not required to be part of the original program P; however, it must be a logical consequence of P. The rules in Fig. 7 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. In addition, note that 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 constraint it

$$\frac{\begin{array}{c}P \models p(\mathbf{x}) \leftarrow p(\mathbf{x}') \wedge C \\ F \models p(\mathbf{x}) \leftarrow x \leq y + c \wedge D \\ D \text{ entails } \exists \mathbf{x}'.D[\mathbf{x}'/\mathbf{x}] \wedge C \\ C \text{ entails } x'{=}x + c_x \wedge y'{=}y + c_y \text{ where } c_y - c_x > 0\end{array}}{P \oplus F \models p(\mathbf{x}) \leftarrow D} \text{ Rule 1} \qquad \frac{\begin{array}{c}P \models p(\mathbf{x}) \leftarrow p(\mathbf{x}') \wedge C \\ F \models p(\mathbf{x}) \leftarrow x \leq c \wedge D \\ D \text{ entails } \exists \mathbf{x}'.D[\mathbf{x}'/\mathbf{x}] \wedge C \\ C \text{ entails } x'{=}x + c_x \text{ where } c_x > 0\end{array}}{P \oplus F \models p(\mathbf{x}) \leftarrow D} \text{ Rule 3}$$

$$\frac{\begin{array}{c}P \models p(\mathbf{x}) \leftarrow p(\mathbf{x}') \wedge C \\ F \models p(\mathbf{x}) \leftarrow x = c \wedge D \\ D \text{ entails } \exists \mathbf{x}'.D[\mathbf{x}'/\mathbf{x}] \wedge C \\ C \text{ entails } x'{=}x - 1\end{array}}{P \oplus F \models p(\mathbf{x}) \leftarrow D \wedge x \geq c} \text{ Rule 2}$$

Fig. 7. Deductive systems for accelerating least fixpoint computation

would be necessary to work with a mixed integer-real constraint solver. This idea goes beyond the scope of this paper.

The acceleration rules of Fig. 7 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 still be accurate.

Proposition 3 (Soundness of accelerations). Given a program P and a set of facts F, if we can derive $P \oplus F \models A$ using one of the rules in Fig. 7, then every instance of A is a logical consequence of $P \oplus F$.

Proof. By cases on the rules.
Rule 1. Given a program P and a set of constrained facts F, let $P \models p(\mathbf{x}) \leftarrow p(\mathbf{x}') \wedge C$ and $F \models p(\mathbf{x}) \leftarrow x \leq y + c \wedge D$ such that D entails $\exists \mathbf{x}'.D[\mathbf{x}'/\mathbf{x}] \wedge C$, and $C \models x'{=}x + c_x \wedge y'{=}y + c_y$ where $c_y - c_x > 0$.

Now, let us call P' the program containing the single clause $p(\mathbf{x}) \leftarrow p(\mathbf{x}') \wedge C$. Let F_n be the singleton whose element is the constrained fact $p(\mathbf{x}) \leftarrow x \leq y + c + n(c_y - c_x) \wedge D$. According to this definition, F_0 is the singleton containing $p(\mathbf{x}) \leftarrow x \leq y + c \wedge D$. Furthermore, by hypothesis, it follows that $P' \oplus F_0$ is a logical consequence of $P \oplus F$.

In order to show that Rule 1 is sound, we have to show that the set of instances of the constrained fact $p(\mathbf{x}) \leftarrow D$ are contained in the least fixpoint of $P' \oplus F_0$. Let us recall that, for any P, the T_P operator is monotonic and continuous over the lattice of sets of ground facts ordered by set inclusion [31, 39]. From Tarski's theorem, it follows that then $lfp(T_P) = \bigcup_{n=0}^{\omega} T_P \uparrow_n$, where $T_P \uparrow_0 = \emptyset$ and $T_P \uparrow_{n+1} = T_P(T_P \uparrow_n)$. Based on this observation, to prove that the instances of $p(\mathbf{x}) \leftarrow D$ are contained in $lfp(T_{P' \oplus F_0})$ we need the following lemma:

$$[F_n]_{\mathbb{N}} \subseteq T_{P' \oplus F_0} \uparrow_n, \text{ for all } n \geq 0.$$

We prove this lemma by induction on n.

– $n = 0$. By definition.

– $n > 0$. Let us assume that $[F_n]_{\mathbb{N}} \subseteq T_{P' \oplus F_0} \uparrow_n$. By monotonicity of the operator $T_{P' \oplus F_0}$, it follows that

$$T_{P' \oplus F_0}([F_n]_{\mathbb{N}}) \subseteq T_{P' \oplus F_0} \uparrow_{n+1} \qquad (1).$$

Now, by the property of $T_{P' \oplus F_0}$ and $S_{P' \oplus F_0}$ mentioned in Sect. 4, it holds that

$$T_{P' \oplus F_0}([F_n]_{\mathbb{N}}) = [S_{P' \oplus F_0}(F_n)]_{\mathbb{N}} \qquad (2).$$

We can now apply the definition of $S_{P' \oplus F_0}$ and we obtain that $p(\mathbf{x}) \leftarrow D[\mathbf{x}'/\mathbf{x}] \wedge C \wedge x \leq y + c + (n+1)(c_y - c_x)$ belongs to $S_{P' \oplus F_0}(F_n)$. If we now apply the hypothesis that D entails $\exists \mathbf{x}'.D[\mathbf{x}'/\mathbf{x}] \wedge C$ we obtain that the solutions of the previous fact contain the solutions of the fact $p(\mathbf{x}) \leftarrow x \leq y + c + (n+1)(c_y - c_x) \wedge D$ (the single fact contained in F_{n+1}). Following from (2), we obtain that the following property holds

$$[F_{n+1}]_{\mathbb{N}} \subseteq T_{P' \oplus F_0}([F_n]_{\mathbb{N}}),$$

and, from (1), we obtain the thesis, i.e.,

$$[F_{n+1}]_{\mathbb{N}} \subseteq T_{P' \oplus F_0} \uparrow_{n+1}.$$

Let us go back to the main proposition. It remains to show that

$$\bigcup_{n=1}^{\omega} [F_n]_{\mathbb{N}} = [p(\mathbf{x}) \leftarrow D]_{\mathbb{N}}.$$

We first note that $\bigcup_{n=1}^{\omega} [F_n]_{\mathbb{N}} = [\bigcup_{n=0}^{\omega} F_n]_{\mathbb{N}}$. Furthermore, $\bigcup_{n=0}^{\omega} F_n$ is equivalent to the 'infinite' formula $p(\mathbf{x}) \leftarrow D \wedge (\bigvee_{n=0}^{\omega} x \leq y + c + (n+1)(c_y - c_x))$. At this point, we note that the previous 'infinite' formula is equivalent to $p(\mathbf{x}) \leftarrow D$ since $c_y - c_x > 0$. This gives us the thesis. □

Rule 2. The proof is similar to the proof for Rule 1.

Rule 3. Under the hypothesis of Rule 3 and proceeding as in the previous cases, iterating the application of $S_{P' \oplus F_0}$ we would obtain the 'infinite' formula $p(\mathbf{x}) \leftarrow D \wedge (\bigvee_{n=1}^{\omega} x = c + n)$. To conclude, we note that, since all variables range over $\mathbb{N}$, the previous formula is equivalent

to $p(\mathbf{x}) \leftarrow D \wedge x \geq c$, i.e., the constrained fact inferred by Rule 3. □

As mentioned before, the previous soundness results allow us to accelerate the fixpoint computation via S_P without loss of precision. Formally, we have the following theorem:

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 Fig. 7 yields (if terminating) a full test of safety properties for concurrent systems over integers or reals.

Proof. By the soundness of Prop. 3. □

6.3 Strategy for acceleration

In this section, we propose a strategy for the selection of the *clause* and *facts* that are needed to apply the acceleration rule of Fig. 7. The method we propose can be viewed as a dynamic generation of *loops* of the original CLP programs. Our model checker implements this strategy during the computation of safety properties.

Specifically, let us first consider Rule 1 in Fig. 7. Assume that F is the current set of constrained facts inferred after a given number of steps in the computation of least fixpoint of S_P. Let $f_1 = p(\mathbf{x}) \leftarrow C$ be a new fact we want to add to the database and let $f_2 = p(\mathbf{x}) \leftarrow D$ be one of the facts in the F. 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 is reachable from f_1 applying in order $c_1, \ldots, c_n$. Then, we 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 $c_i = p(\mathbf{x}) \leftarrow C_i \wedge p(\mathbf{x}')$, and let P_i be the singleton program with the clause c_i for $i : 2, \ldots, n$. We first compute the following fact:

$$p(\mathbf{x}) \leftarrow C = S_{P_n}(\ldots S_{P_2}(p(\mathbf{x}) \leftarrow C_1)\ldots).$$

The unfolded clause is then defined as $p(\mathbf{x}) \leftarrow C \wedge p(\mathbf{x}')$. A similar idea can be used for the other rules of Fig. 7. For rule 3, we need, however, a slightly different strategy. Since this rule is used to detect periodic behavior of a variable, say x, instead of looking for two facts f_1 and f_2 such that f_1 entails f_2, 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 entails C and f_2 is reachable from f_1 via a sequence of clauses (as before).

7 Other techniques

In this section we discuss briefly other techniques that can be useful to obtain more flexible working tools. We first consider *approximation* techniques inspired by the work of Cousot and Halbwachs [15], and that can be used, for example, when the accuracy of the relaxation integer-real cannot be guaranteed by the form of the program and of the property taken into consideration.

7.1 Conservative approximations

In the previous section, we have focused our attention on the class of systems expressible via simple CLP systems. What is the effect of the relaxation integer-reals in the more general case where guards and actions range over arbitrary linear constraints? Clearly, the following relation holds for any CLP program P defined over a variables of integer type:

> the set of *integer* solutions of $lfp(S_{P,\mathbb{N}})$ are contained in the set of *integer* solutions of $lfp(S_{P,\mathbb{R}_+})$.

As a consequence, in the general case of CLP programs with linear arithmetic constraints the relaxation integer-real gives us a conservative approximation of safety and liveness properties: a positive answer computed over the reals corresponds to a positive answer for the integer system, whereas a counter-example computed over the reals corresponds to a *don´t know* answer.

Following ideas developed in abstract interpretation [14], 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 [15], 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

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

Formally, $F \Uparrow F'$ contains each constrained fact that is obtained from some constrained fact $p(\tilde{x}) \leftarrow c_1 \wedge \ldots \wedge c_n$ in F' by removing all conjuncts c_i that are strictly entailed by some conjunct d_j of some 'compatible' constrained atom $p(\tilde{x}) \leftarrow d_1 \wedge \ldots \wedge d_m$ 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 [15] and to the improved versions in [6, 38], 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, for example, the two sets of constrained atoms

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

When applied to F and F', each of the widening operator in [6, 15, 38] 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 with respect to the set of control locations. The partitioning reduces the applicability of the widening for the benefit of precision of the computation (see also [6, 38]).

7.2 Mixed forward-backward Reachability

To conclude the discussion about *alternative* strategies that one could use when other methods fail, we would like to mention the *magic-set templates* algorithm [55] introduced in the context of bottom-up evaluation of (constraint) logic programs [55]. The magic-set template algorithm defines *goal-driven* bottom-up computations. In our context, we can use this technique in order to specialize the least fixpoint computation of a given safety property with respect to the *init* goal.

More precisely, following [55], the application of a kind of magic-set transformation on the CLP program $P = P_{\mathcal{S}} \oplus F$, where the clauses have a restricted form (one or no predicate in the body), yields the following CLP program $\tilde{P}$ (with new predicates $\tilde{p}$ and $\widetilde{init}$):

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

We obtain the soundness of this transformation with respect to the verification of safety properties by standard results [55] which say that $init \in lfp(T_P)$ if and only if $init \in lfp(T_{\tilde{P}})$ (which is, $init \in lfp(S_{\tilde{P}})$). The soundness continues to hold if we replace the constraints c in the clauses $\tilde{p}(\mathbf{x}') \leftarrow c, \tilde{p}(\mathbf{x})$ in $\tilde{P}$ 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.

Though independently developed in the field of logic programming, this technique is similar to mixed backward-forward strategies used in verification (e.g., see [6]).

8 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.

8.1 Communication protocols

8.1.1 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 [5]), 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.

8.1.2 Ticket algorithm

The *ticket* algorithm (see Fig. 8) 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 variable (a for P_1 and b for P_2) used to store the current value of its ticket. Figure 9 shows the simple CLP program resulting from the translation of the algorithm taken into consideration

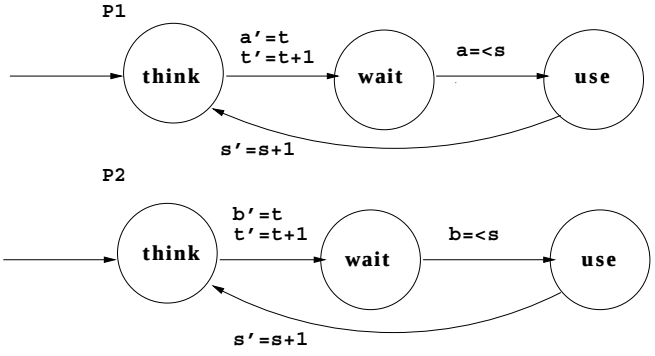

Fig. 8. The ticket algorithm

$$
\begin{array}{lll}
init & \leftarrow T = S, & p(think, think, 0, 0, T, S). \\
p(think, P, A, B, T, S) & \leftarrow T >= 0, T_1 = T + 1, A_1 = T, & p(wait, P, A_1, B, T_1, S). \\
p(P, think, A, B, T, S) & \leftarrow T >= 0, T_1 = T + 1, B_1 = T, & p(P, wait, A, B_1, T_1, S). \\
p(wait, P, A, B, T, S) & \leftarrow A =< S, & p(use, P, A, B, T, S). \\
p(P, wait, A, B, T, S) & \leftarrow B =< S, & p(P, use, A, B, T, S). \\
p(use, P, A, B, T, S) & \leftarrow S >= 0, S_1 = S + 1, & p(think, P, A, B, T, S_1). \\
p(P, use, A, B, T, S) & \leftarrow S >= 0, S_1 = S + 1, & p(P, think, A, B, T, S_1).
\end{array}
$$

Fig. 9. The CLP-program for the ticket algorithm

(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. 7) during the fixpoint iterations. In a second experiment, we applied the magic set transformation instead and obtained a proof in 0.6 s. We proved starvation freedom, i.e., $AG(p_1 = wait \rightarrow AF(p_1 = use))$, in 1.5 s 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 [6] in the following section.

8.1.3 Bounded buffer

The first protocol we consider models the communication of producers and consumers connected via a buffer of size s. Figure 10 shows the behavior of a producer and of a consumer; here the variable a denotes the number of *empty* cells in the buffer, p the number of produced items, and c the number of consumed items. The CLP-program that describes a system with two producers, two consumers and one buffer is given in Fig. 11.

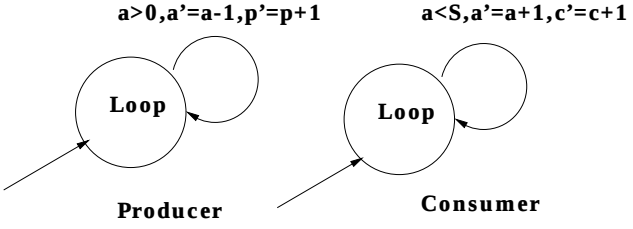

Fig. 10. Producer-consumer with bounded buffer

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 integer-real will give us a conservative approximation of the property. Before applying our model checker, we transform the previous property $AG(\neg(C_1 \vee C_2))$, where $C_1 \equiv p_1 + p_2 - (c_1 + c_2) + a \leq s - 1$ and $C_2 \equiv p_1 + p_2 - (c_1 + c_2) + a \geq s + 1$. Our model checker proves the property in 0.2 s 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.

8.1.4 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. Figure 12 shows the behavior of 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. 13 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., $AG(\neg(C_1 \vee C_2))$ where $C_1 \equiv p \leq c + q_1 + q_2 - 1$ and $C_2 \equiv p \geq c + q_1 + q_2 + 1$ (a non-simple constraint).

$$
\begin{array}{ll}
init & \leftarrow A = S, P_1 = P_2 = C_1 = C_2 = 0, S \geq 1, p(A, P_1, P_2, C_1, C_2, S). \\
p(A, P_1, P_2, C_1, C_2, S) & \leftarrow A \geq 1, A' = A - 1, P_1' = P_1 + 1, p(A', P_1', P_2, C_1, C_2, S). \\
p(A, P_1, P_2, C_1, C_2, S) & \leftarrow A \geq 1, A' = A - 1, P_2' = P_2 + 1, p(A', P_1, P_2', C_1, C_2, S). \\
p(A, P_1, P_2, C_1, C_2, S) & \leftarrow A \leq S - 1, A' = A + 1, C_1' = C_1 + 1, p(A', P_1, P_2, C_1', C_2, S). \\
p(A, P_1, P_2, C_1, C_2, S) & \leftarrow A \leq S - 1, A' = A + 1, C_2' = C_2 + 1, p(A', P_1, P_2, C_1, C_2', S).
\end{array}
$$

Fig. 11. CLP for producers and consumers connected via a bounded buffer

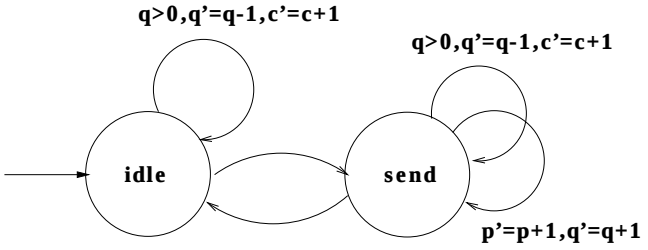

Fig. 12. Producer-consumer with unbounded buffer

We prove the property in three steps by applying the widening operator of Sect. 7.1.

8.2 Array bounds checking

In this section we will discuss an application of our model checker for checking array bounds of imperative programs [15]. As a first step, we extract the information involving manipulation of indexes of arrays out of the flow graph of a program. 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 algorithm.

8.2.1 Insertion sorting

The procedure written in C of Fig. 14 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 cannot access the array A outside the interval $[0,n-1]$ (in C array indexing starts from 0). As anticipated before, the first step consists of extracting all information involving array indexes. The 'simple' CLP of Fig. 15a 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, and the dual condition $\neq (i \geq 0 \wedge A[i] > x)$ is relaxed into *true*. In other words, a property of the abstract graph is a conservative approximation for a property of the original program.

```
        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. 14. Insertion sorting where we add labels for the program locations

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. 15b. Since both the program and the properties are expressed using simple constraints, the analysis over the reals will give accurate results. Our model checker proves the procedure safe from array bounds errors without need of accelerations.

8.3 Parameterized systems

We conclude the section dedicated to the examples presenting the analysis of a special class of parameterized systems called *broadcast protocols*. Broadcast protocols [25] 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).

As an example, we consider the cache coherence protocol presented by Emerson and Namjoshi in [25]. Several processors interact with the main memory through a single line caches connected on a shared bus. Write commands issued by a processor on its cache must invalidate the data of all other caches. We model this system as a collection of identical processes each described as in Fig. 16. The label *a!!* of a transition of Fig. 16 indicates a broadcast sent to all caches; *a??* indicates the reception of the broadcast. The other labels indicate internal actions of the cache. Each cache can be in one of four states:

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

Fig. 13. CLP-program for producers and consumers connected via an unbounded buffer

$$
\begin{array}{lll}
init & \leftarrow p(init,K,N,I). & \\
p(init,K,N,I) & \leftarrow p(for,K1,N,I), & K1=1. \\
p(for,K,N,I) & \leftarrow p(entryA1,K,N,I), & K \leq N-1. \\
p(entryA1,K,N,I) & \leftarrow p(while,K,N,I1), & I1=K-1. \\
p(while,K,N,I) & \leftarrow p(entryA2,K,N,I), & I \geq 0. \\
p(entryA2,K,N,I) & \leftarrow p(while,K,N,I1), & I1=I-1. \\
p(while,K,N,I) & \leftarrow p(entryA3,K,N,I), & true. \\
p(entryA3,K,N,I) & \leftarrow p(for,K1,N,I), & K1=K+1. \\
p(for,K,N,I) & \leftarrow p(end,K,N,I), & K \geq N.
\end{array}
$$

$$
\begin{array}{l}
p(entryA1,K,N,I) \leftarrow K \geq N. \\
p(entryA1,K,N,I) \leftarrow K \leq -1. \\
p(entryA3,K,N,I) \leftarrow I \geq N-1. \\
p(entryA3,K,N,I) \leftarrow I \leq -2. \\
p(entryA2,K,N,I) \leftarrow I \leq -1. \\
p(entryA2,K,N,I) \leftarrow I \leq -2. \\
p(entryA2,K,N,I) \leftarrow I \geq N-1. \\
p(entryA2,K,N,I) \leftarrow I \geq N.
\end{array}
$$

Fig. 15. a CLP program for insertion sorting; **b** potential violations of array bounds

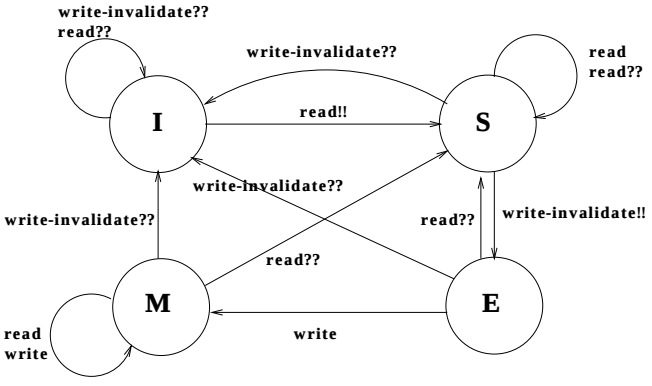

Fig. 16. A cache protocol

I=the cache has an *invalid* copy of data; *S*=the cache has a potentially *shared* copy of data; *E*=the cache has an *exclusive* copy of data; *M*=the cache has a *modified* copy of data. On a write request, a cache in state I broadcasts the *write-invalidate* message to the other caches. A cache that receives the broadcast immediately goes to the state *I*. Similarly, on a read request, the cache sends the *read* broadcast that forces all caches to go to the state *S*. On write requests in the exclusive state *E*, the cache moves to the modified state *M*. Following [25], we can model this parameterized system as a Petri Net with transfer arcs, where we use four variables I, S, E,, and M to *count* the *number* of processes in each state. Each action corresponds to a re-allocation of the counters. The resulting CLP program is shown in Fig. 17. Specifically, the second rule of Fig. 17 corresponds to the *read!!* broadcast, the third rule corresponds to the *write-invalidate!!* broadcast, and the fourth rule to a *write* for a cache in state *E*. The last two rules model *read* and *write* commands in state *M* and *S*. Note that, in the initial configuration, the number of processes in state I is left unspecified ($I \geq 1$).

The safety properties proved in [25] are: *readers and writes cannot access simultaneously the cache*, expressed via the temporal property $AG(\neg(S \geq 1 \wedge M \geq 1))$; *there cannot be more than one process in the exclusive states*, expressed via the temporal property $AG(\neg(M+E \geq 2))$, i.e., $AG(\neg((M \geq 1 \wedge E \geq 1) \vee (E \geq 2) \vee (M \geq 2)))$. These properties can be expressed as *upwards closed sets*. The reachability problem is decidable in this case [1, 26]. In addition, note that the CLP program in Fig. 17 is not *simple*. Our model checker automatically proves both properties after few iterations (without use of accelerations). The state space for the verification problem of broadcast protocols 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 [21].

8.3.1 Performance

The execution times obtained for all examples described in this section are listed in Fig. 18. In Fig. 18, we also list the execution times for other examples: *selection*, and *matrix multiplication* are programs extracted from C-programs that implement the selection sorting, and row per column matrix multiplication, respectively. Finally, the example *csm* is a parameterized system describing the central server model of [21]. All the verification problems have been tested on a Sun Sparc Station 4, OS 5.5.1.

9 Related work

There have been other attempts to connect logic programming and verification, none of which has our gener-

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

Fig. 17. Integer-system for the protocol of Fig. 16

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	-	-	-	-
mesi	4	0.0	2	0.0	2	-	-	-	-
csm	9	38.2	27	58	25	-	-	-	-

Fig. 18. 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

ality with respect to the applicable concurrent systems and temporal properties. In [29], 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 [28], Fribourg and Richardson use CLP programs over *gap-order integer constraints* [53] 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 [27], Fribourg and Olsen study reachability for a 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 [52], 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 [60], 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 *cannot* be directly expressed through fixpoints of the S_P operator (because the clauses translating time transitions may loop). In [32], Gupta and Pontelli describe runs of timed automata using the top-down operational semantics of CLP-programs (and not the fixpoint semantics). In [18], 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 [18] are not suitable for translating general concurrent systems. In [54], 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 [23]). The integration of tabling with constraints is possible in principle and has a promising potential.

As described in [44], constraints as symbolic representations of states are used in UPPAAL, a verification tool for timed systems [45]. 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, for example, starvation freedom.

We will next discuss work on other verification procedures for integer-valued systems. In [5, 6], Bultan, Gerber, and Pugh use the Omega library [51] 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 [4], their method is extended to a composite approach (using BDDs), whose adaptation to the CLP setting may be an interesting task. In [13], Chan, Anderson, Beame, and Notkin incorporate an efficient representation of arithmetic constraints (linear and non-linear) into the BDDs of SMV [47]. 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 [3], Bérard and Fribourg show that the relaxation integer-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: it 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 acceleration rules are related to Boigelot and Wolper's loop-first technique [10] 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 integer-reals constraint solvers might be useful (if not necessary) for the manipulation of this type of constraint. In [21], Delzanno, Esparza, and Podelski discuss in detail the theoretical complexity of the analysis of broadcast protocols over integer arithmetics. Finally, the reader may refer to [20] for more details on verification of parameterized cache coherence protocols.

This paper is an extension of the paper we presented at TACAS'99 [22].

10 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 have defined 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 [39]; 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 also useful 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, Christian Holzbaur, Tevfik Bultan, Richard Gerber, Supratik Mukhophadyay, and C.R. Ramakrishnan for fruitful discussions and encouragement, and the anonymous referees for their help in improving the paper.

References

1. Abdulla, P.A., Čer-ans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: 11th Annual Symposium on Logic in Computer Science (LICS'96), pp. 313–321. IEEE Computer Society, New York, 1996
2. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Mitchell, J.C. (ed.): Proc. 5th Annual Symposium on Logic in Computer Science (LICS'90), pp. 428–439. IEEE Society, New York, 1990
3. Bérard, B., Fribourg, L.: Reachability analysis of (timed) petri nets using real arithmetic. In: Baeten, J., Mauw, S. (eds.): Proc. 10th Int. Conf. on Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands. LNCS 1664. Berlin, Heidelberg, New York: Springer-Verlag, 1999, pp. 178–193
4. Bultan, T., Gerber, R., League, C.: Verifying systems with integer constraints and Boolean predicates: a composite approach. In: 1998 ACM/SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA'98), pp. 113–123. ACM, 1998
5. Bultan, T., Gerber, R., Pugh, W.: Symbolic model checking of infinite-state systems using presburger arithmetics. In: Orna Grumberg, (ed.), Proc. 9th Conference on Computer Aided Verification (CAV'97). LNCS 1254. Berlin, Heidelberg, New York: Springer-Verlag, 1997, pp. 400–411
6. Bultan, T., Gerber, R., Pugh, W.: 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, Md., USA, 1998
7. Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The power of QDD's. In: Proc. of Int. Static Analysis Symposium, Paris. LNCS 1302. Berlin, Heidelberg, New York: Springer-Verlag, 1997, pp. 172–186
8. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W., Weise, C.: New generation of UPPAAL. In: Int. Workshop on Software Tools for Technology Transfer. Aalborg, Denmark, 12–13 July, 1998
9. Bultan, T.: Automated symbolic analysis of reactive systems. Ph.D. Thesis, Department of Computer Science, University of Maryland, College Park, Md., USA, August, 1998
10. Boigelot, B., Wolper, P.: Symbolic verification with periodic sets. In: David Dill, (ed.), Proc. 6th Int. Conf. on Computer Aided Verification (CAV'94). LNCS 818. Berlin, Heidelberg, New York: Springer-Verlag, 1994, pp. 55–67
11. Boigelot, B., Wolper, P.: Verifying systems with infinite but regular state space. In: Hu, A.J., Vardi, M.Y. (eds.), Proc. 10th Conference on Computer Aided Verification (CAV'98).

LNCS 1427. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 88–97
12. Čer-ans, K.: Deciding properties of integral relational automata. In: Abiteboul, S., Shamir, E.(eds.), Proc. 21st Int. Conf. on Automata, Languages and Programming (ICALP '94). LNCS 820. Berlin, Heidelberg, New York: Springer-Verlag, 1994, pp. 35–46
13. Chan, W., Anderson, R., Beame, P., Notkin, D.: Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In: Orna Grumberg, (ed.), Proc. 9th Conference on Computer Aided Verification (CAV'97). LNCS 1254. Berlin, Heidelberg, New York: Springer-Verlag, 1997, pp. 316–327
14. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. 4th the ACM Symposium on Principles of Programmming Languages (POPL'77), pp. 238–252. ACM, 1977
15. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: 5th Annual ACM Symposium on Principles of Programming Languages (POPL'78). ACM, 1978, pp.84–96
16. Clarke, E.M., Grümberg, O., Long, D.E.: Model checking and abstraction. In: 19th Annual Symposium on Principles of Programming Languages (POPL'92). ACM, 1992, pp. 343–354
17. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis, and presburger arithmetics. In: Hu, A.J., Vardi, M.Y. (eds.): Proc. 10th Conference on Computer Aided Verification (CAV'98). LNCS 1427. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 268–279
18. Charatonik, W., Podelski, A.: Set-based analysis of reactive infinite-state systems. In: Steffen, B. (ed.): Proc. 1st Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98). LNCS 1384. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 358–375
19. Dams, D.: Abstract Interpretation and partition refinement for model checking. Ph.D. Thesis, Eindhoven University of Technology, 1996
20. Delzanno, G.: Automatic verification of cache coherence protocols. In: Emerson, E.A., Sistla, A.P. (eds.): Proc. 11th Conference on Computer Aided Verification (CAV 2000). LNCS 1855. Berlin, Heidelberg, New York: Springer-Verlag, 2000, pp. 53–68
21. Delzanno, G., Esparza, J., Podelski, A.: Constraint-based analysis of broadcast protocols. In: Flum, J., Rodriguez-Artalejo, M. (eds.): Proc. Annual Conference of the European Association for Computer Science Logic (CSL 99). LNCS 1683. Berlin, Heidelberg, New York: Springer-Verlag, 1999, pp. 50–66
22. Delzanno, G., Podelski, A.: Model checking in CLP. In: Rance Cleaveland, W. (ed.): Proc. 5th Int. Conf. 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). LNCS 1579. Berlin, Heidelberg, New York: Springer-Verlag, 1999, pp. 223–239
23. Dong, Y., Du, X., Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Sokolsky, O., Stark, E.W., Warren, D.S.: Fighting livelock in the i-protocol: a comparative study of verification tools. In: Cleaveland, W.R. (ed.): Proc. 5th Int. Conf. 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). LNCS 1579. Berlin, Heidelberg, New York: Springer-Verlag, 1999, pp. 74–88
24. Emerson, E.A.: Temporal and modal logic. In: Jan van Leeuwen, (ed.), Handbook of Theoretical Computer Science, vol. B, pp. 995–1072. Elsevier Science, 1990
25. Emerson, E.A., Namjoshi, K.S.: On model checking for nondeterministic infinite-state systems. In: 13th Annual Symposium on Logic in Computer Science (LICS'98), IEEE Computer Society, New York, 1998
26. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: 14th IEEE Int. Symposium on Logic in Computer Science (LICS'99). IEEE Computer Society, New York, 1999
27. Fribourg, L., Olsen, H.: A decompositional approach for computing least fixed point of datalog programs with z-counters. J. Constraints 2(3-4): 305–336, 1997
28. Fribourg, L., Richardson, J.: Symbolic verification with gap-order constraints. Technical Report LIENS-93-3, Laboratoire d'Informatique, Ecole Normale Superieure, Paris, 1996
29. Fribourg, L., Veloso-Peixoto, M.: Automates concurrents à contraintes. Technique et Science Informatiques 13(6): 837-866, 1994
30. Finkel, A., Schnoebelen, Ph.: Well-structured transition systems everywhere! Technical Report LSV-98-4, Laboratoire Spécification et Vérification, Ecole Normale Supérieure de Cachan. April 1998. In: Theor. Comput. Sci., 2001, (to appear)
31. Gabbrielli, M., Dore, M.G., Levi, G.: Observable semantics for constraint logic programs. J. Logic Comput. 2(5): 133–171, 1995
32. Gupta, G., Pontelli, E.: A constraint based approach for specification and verification of real-time systems. In: 18th IEEE Real Time Systems Symposium (RTSS'97). IEEE Computer Society, New York 1997
33. Graf, S.: Verification of a distributed cache memory by using abstractions. In: Dill, D.L. (ed.), Proc. Computer-aided Verification (CAV'94). LNCS 818. Berlin, Heidelberg, New York: Springer-Verlag 1994, pp. 207–219
34. Halbwachs, N.: Delay analysis in synchronous programs. In: 5th Conference on Computer-Aided Verification (CAV'93), Elounda (Greece). LNCS 697. Berlin, Heidelberg, New York: Springer-Verlag, 1993, pp. 333–346
35. Henzinger, T.A., Ho, P.-H.: A note on abstract interpretation strategies for hybrid automata. In: Antsaklis, P., Nerode, A., Kohn, W., Sastry, S. (eds.): Proc. Hybrid Systems II. LNCS 999. Berlin, Heidelberg, New York: Springer-Verlag, 1995, pp. 252–264
36. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HyTech: a model checker for hybrid systems. In: Orna Grumberg, (ed.), Proc. 9th Conference on Computer Aided Verification (CAV'97). LNCS 1254. Berlin, Heidelberg, New York: Springer-Verlag, 1997, pp. 460–463
37. Holzbaur, C.: OFAI CLP(Q,R), Manual, Edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995
38. Halbwachs, N., Y-Proy, E., Roumanoff, P.: Verification of real-time systems using linear relation analysis. Formal Methods in System Design 11(2): 157–185, 1997
39. Jaffar, J., Maher, M.J.: Constraint logic programming: a survey. J. Logic Program. 19-20: 503–582, 1994
40. Kanellakis, P.C., Kuper, G.M., Revesz, P.Z.: Constraint query languages. J. Comput. Syst. Sci. 51: 6–52, 1995
41. Karp, R.M., Miller, R.E.: Parallel program schemata. J. Comput. Syst.m Sci. 3: 147-195, 1969
42. Lambert, J.: A structure to decide reachability in Petri nets. Theor. Comput. Sci. 99(1): 79–1044, 1992
43. Loiseaux, C., Graf, S., Sifakis, J., Bouajiani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design. Kluwer Academic, Boston, Mass., USA, pp. 1–35, 1994
44. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: 18th IEEE Real Time Systems Symposium (RTSS'97), pp. 14–24. IEEE Computer Society, New York, 1997
45. Larsen, K.G., Pettersson, P., Yi, W.: uppaal in a Nutshell. Int. J. Software Tools Technol. Transfer 1(1-2): 134–152, 1997
46. Maher, M.J.: Constrained dependencies. In: Ugo Montanari, (ed.), Proc. 1st Int. Conf. on Principles and Practice of Constraint Programming (CP'95), Cassis, France, 19–22 September, Lecture Notes in Computer Science. Berlin, Heidelberg, New York: Springer-Verlag, 1995, pp. 170–185
47. McMillan, K.L.: Symbolic model checking: an approach to the state explosion problem. Kluwer Academic, Boston, Mass., USA, 1993
48. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: safety. Berlin, Heidelberg, New York: Springer-Verlag, 1995

49. Maher, M.J., Ramakrishnan, R.: Déjà vu in fixpoints of logic programs. In: Lusk, R.A., Overbeek, E.L. (eds.): Proc. North American Conference on Logic Programming (NACLP'89). MIT, Cambridge, Mass., USA, 1989, pp. 963–980
50. Podelski, A.(ed.): Constraint programming: basics and trends. In: LNCS 910. Berlin, Heidelberg, New York: Springer-Verlag, 1994
51. Pugh, W.: The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In: MacCallum, A.C. (ed.): Proc. 4th Annual Conference on Supercomputing, pp. 4–13, Albuquerque, N.M., USA, November 18-22, 1991. IEEE Computer Society, New York
52. Rauzy, A.: Toupie: a constraint language for model checking. In: Podelski [50], pp. 193–208
53. Revesz, P.Z.: A closed-form evaluation for datalog queries with integer (gap)-order constraints. Theor. Comput. Sci. 116(1): 117–149, 1993
54. Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Swift, T., Warren, D.S.: Efficient model checking using tabled resolution. In: Grumberg, O. (ed.): Proc. 9th Conference on Computer Aided Verification (CAV'97). LNCS 1254. Berlin, Heidelberg, New York: Springer-Verlag, 1997, pp. 143–154
55. Ramakrishnan, R., Srivastava, D., Sudarshan, S.: Efficient bottom-up evaluation of logic programs. In: De Wilde, P., Vandewalle, J. (eds.): Computer Systems and Software Engineering: State-of-the-Art, ch. 11. Kluwer Academic, Boston, Mass., USA, 1992
56. Schrijver, A.: Theory of linear and integer programming. Wiley, New York, 1986 (reprinted 1994)
57. Shankar, U.A.: An introduction to assertional reasoning for concurrent systems. ACM Comput. Surv. 25(3): 225–262, 1993
58. Shiple, T.R., Kukula, J.H., Ranjan, R.K.: A comparison of presburger engines for efsm reachability. In: Hu, A.J., Vardi, M.Y. (eds.): Proc. 10th Conference on Computer Aided Verification (CAV'98). LNCS 1427. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 280–292
59. Srivastava, D.: Subsumption and indexing in constraint query languages with linear arithmetic constraints. Ann. Math. Artif. Intell. 8(3-4): 315–343, 1993
60. Urbina, L.: Analysis of hybrid systems in CLP(R). In: Freuder, E.C. (ed.): Proc. Principles and Practice of Constraint Programming (CP'96). LNCS 1118. Berlin, Heidelberg, New York: Springer-Verlag, 1996, pp. 451–467
61. Wallace, M.: Practical applications of constraint programming. J. Constraints 1(1-2): 139–168, 1996