Model Checking in CLP

Giorgio Delzanno and Andreas Podelski

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

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. Our contributions are: (1) a semantics-preserving translation of *concurrent systems* into CLP programs, (2) a method for verifying safety and liveness properties on 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 here linear constraints as opposed to Presburger arithmetic as in previous solutions.

1 Introduction

Automated verification methods can today be applied to practical systems [McM93]. One reason for this success is that implicit representations of finite sets of states through Boolean formulas can be handled efficiently via BDD's [BCM+90]. The finiteness is an inherent restriction here. Many systems, however, operate on data values from an infinite domain and are intrinsically infinite-state; i.e., one cannot produce a finite-state model without abstracting away crucial properties. There has been much recent effort in verifying such systems (see e.g. [AČJT96, BW98, BGP97, CJ98, HHWT97, HPR97, LPY97, SKR98]). One important research goal is to find appropriate data structures for implicit representations of infinite sets of states, and design model checking algorithms that perform well on practical examples.

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

We present a translation from *concurrent systems* with infinite state spaces to CLP programs that preserves the semantics in terms of transition sequences. The formalism of 'concurrent systems' is a widely-used guarded-command specification language with shared variables promoted by Shankar [Sha93]. Using this translation, we exhibit the connection between states and *ground atoms*, between sets of states and *constrained facts*, between the pre-condition operator and the *logical consequence operator* of CLP programs, and, finally, between CTL properties (safety, liveness) and model-theoretic or denotational program semantics. This connection suggests a natural approach to model checking for infinite-state systems using CLP. 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 compute fixpoints 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.

We have obtained experimental results for several examples of infinite-state programs; these examples are quickly becoming benchmarks in the community (see e.g. [BGP97, BGP98, SKR98, SUM96, LS97]). Our experiments allow us to see that a CLP-based tool can solve the considered verification problems at acceptable time cost. Moreover, as CLP combines mathematical and logical reasoning, the CLP-based setting helps to find optimizations that are natural, directly implementable and provably correct. This is important since verification is a hard problem (undecidable in the general infinite-state case) and often requires a fine-tuning of the method.

Finally, the experiments show that, perhaps surprisingly, the powerful (triple-exponential time) decision procedure for Presburger Arithmetic used in other approaches [BGP98, SKR98, BW94] 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.

2 Translating Concurrent Systems into CLP

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

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

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

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{repeat}} \\
\quad think: & turn_1 := turn_2 + 1; & \\
\quad wait: & \textbf{when}\ \ turn_1 < turn_2\ \ \textbf{or}\ \ turn_2 = 0\ \ \textbf{do} & \\
\quad use: & \qquad \left[\begin{array}{l}\textbf{critical section};\\ turn_1 := 0\end{array}\right. & \\
\multicolumn{3}{l}{\textbf{forever}}
\end{array}$$

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

The concurrent program above can be directly encoded as the concurrent system $\mathcal{S}$ in Figure 1 following the scheme in [Sha93]. Each process is associated with a *control variable* ranging over the control locations (i.e. program labels). The *data variables* correspond to the program variables. The states of $\mathcal{S}$ are tuples of control and data values, e.g. $\langle think, think, 0, 3\rangle$. The primed version of a variable in an action stands for its successor value. We omit conjuncts like $p_2' = p_2$ expressing that the value remains unchanged.

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

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

Fig. 1. Concurrent system $\mathcal{S}$ specifying the bakery algorithm

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

$$
\begin{array}{lll}
init \leftarrow & Turn_1 = 0, Turn_2 = 0, & p(think, think, Turn_1, Turn_2) \\
p(think, P_2, Turn_1, Turn_2) \leftarrow & Turn_1' = Turn_2+1, & p(wait, P_2, Turn_1', Turn_2) \\
p(wait, P_2, Turn_1, Turn_2) \leftarrow & Turn_1 < Turn_2, & p(use, P_2, Turn_1, Turn_2) \\
p(wait, P_2, Turn_1, Turn_2) \leftarrow & Turn_2 = 0, & p(use, P_2, Turn_1, Turn_2) \\
p(use, P_2, Turn_1, Turn_2) \leftarrow & Turn_1' = 0, & p(think, P_2, Turn_1', Turn_2) \\
\quad \ldots \text{ symmetrically for Process 2} & &
\end{array}
$$

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

If the reader is not familiar with CLP, the following is all one needs to know for this paper.[1] A CLP program is a logical formula, namely a universally quantified

[1] If the reader is familiar with CLP, note that we are proposing a *paradigm shift*:

conjunction of implications (as in Figure 2; it is common to call the implications *clauses* and to write their conjunction as a set). Its first reading is the usual first-order logic semantics. We give it a second reading as a non-deterministic sequential program. The program states are *atoms*, i.e., applications of the predicate p to values such as $p(think, think, 0, 3)$. The successor state of a state s is any atom s' such that the atom s is a direct logical consequence of the atom s' under the program formula. This again is the case if and only if the implication $s \leftarrow s'$ is an *instance* of one of the implications.

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

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

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

We observe that the transition sequences of the concurrent system $\mathcal{S}$ in Figure 1 are exactly the ground derivations of the CLP program $P_{\mathcal{S}}$ in Figure 2. Moreover, the set of all predecessor states of a set of states in $\mathcal{S}$ is the set of its direct logical consequences under the CLP program $P_{\mathcal{S}}$. We will show that these facts are generally true and use them to characterize CTL properties in terms of the denotational (fixpoint) semantics associated with CLP programs.

We will now formalize the connection between concurrent systems and CLP programs. We assume that for each variable x there exists another variable x', the primed version of x. We write $\mathbf{x}$ for the tuple of variables $\langle x_1, \ldots, x_n\rangle$ and $\mathbf{d}$ for the tuple of values $\langle d_1, \ldots, d_n\rangle$. We denote validity of a first-order formula ψ wrt. to a structure $\mathcal{D}$ and an assignment α by $\mathcal{D}, \alpha \models \psi$. As usual, $\alpha[\mathbf{x} \mapsto \mathbf{d}]$ denotes an assignment in which the variables in $\mathbf{x}$ are mapped to the values in $\mathbf{d}$. In the examples of Section 5 formulas will be interpreted over the domains of integers and reals. Note however that the following presentation is given for any structure $\mathcal{D}$.

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

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

instead of looking at the *synthesis* of operational behavior from programs viewed as executable specifications, we are interested in the *analysis* of operational behavior through CLP programs obtained by a translation. The classical correspondence between denotational semantics and operational semantics (for ground derivations) is central again.

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

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

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

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

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

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

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

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

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

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

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

[2] Note that e.g. $p(\mathit{think}, P_2, \mathit{Turn}_1, \mathit{Turn}_2) \leftarrow \ldots$ in the notation used in examples is equivalent to $p(P_1, P_2, \mathit{Turn}_1, \mathit{Turn}_2) \leftarrow P_1 = \mathit{think} \wedge \ldots$ in the notation used in formal statements.

Proof. The clause $p(\mathbf{x}) \leftarrow \psi \wedge \phi \wedge p(\mathbf{x}')$ of $P_{\mathcal{S}}$ corresponds to the event $\langle \psi, \phi \rangle$. Its instances are of the form $p(\mathbf{d}) \leftarrow p(\mathbf{d}')$ where $\mathcal{D}, \alpha[\mathbf{x} \mapsto \mathbf{d}, \mathbf{x}' \mapsto \mathbf{d}'] \models \psi \wedge \phi$. Thus, they correspond directly to the pairs $\langle \mathbf{d}, \mathbf{d'} \rangle$ of the transition relation τ restricted to the event $\langle \psi, \phi \rangle$. This fact can be used to show (i) by induction on the length of a sequence of transitions or derivations and (ii) directly by definition. $\square$

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

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

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

3 Expressing CTL Properties in CLP

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

In the following, the notion of *constrained facts* will be important. A constrained fact is a clause $p(\mathbf{x}) \leftarrow c$ whose body contains only a constraint c. Note that an instance of a constrained fact is (equivalent to) a clause of the form $p(\mathbf{d}) \leftarrow true$ which is the same as the atom $p(\mathbf{d})$, i.e. it is a state. Given a set of constrained facts F, we write $[F]_{\mathcal{D}}$ for the set of instances of clauses in F (also called the 'meaning of F' or the 'set of states represented by F'). For example, the meaning of

$$F_{mut} = \{p(P_1, P_2, Turn_1, Turn_2) \ \leftarrow \ P_1 = use, \, P_2 = use\}$$

is the set of states $[F_{mut}]_{\mathcal{D}} = \{p(use, use, 0, 0), p(use, use, 1, 0), \ldots\}$.

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

As an example of a safety property, consider *mutual exclusion* for the concurrent system $\mathcal{S}$ in Figure 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 program $P_{\mathcal{S}} \oplus F_{mut}$ that we obtain from the union of the CLP Program $P_{\mathcal{S}}$ in Figure 2 and the singleton set of constrained facts F_{mut}. We can compute this fixpoint and show that it does not contain the initial state (i.e. the atom *init*).

As an example of a liveness property, *starvation freedom* for Process 1 ("each time Process 1 waits, it will finally enter the critical section") is expressed by $AG(p_1 = wait \rightarrow AF(p_1 = use))$. Its complement is the set of states

$$
\begin{array}{lll}
init & \leftarrow & Turn_1 = 0, Turn_2 = 0, \;\; p(think, think, Turn_1, Turn_2)
\end{array}
$$

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

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

$EF(p_1 = wait \wedge EG(\neg\, p_1 = use))$. The set of states $EG(\neg\, p_1 = use)$ is equal to the greatest fixpoint for the CLP program $P_{\mathcal{S}} \oslash F_{starv}$ in Figure 3. We obtain $P_{\mathcal{S}} \oslash F_{starv}$ from the CLP Program $P_{\mathcal{S}}$ by a transformation wrt. to the following set of two constrained facts:

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

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

To give an idea about the model checking method that we will describe in the next section: in an intermediate step, the method computes a set F' of constrained facts such that the set of states $[F']_{\mathcal{D}}$ is equal to the greatest fixpoint for the CLP program $P_{\mathcal{S}} \oslash F$. The method uses the set F' to form a third CLP program $P_{\mathcal{S}} \oplus F'$. The least fixpoint for that program is equal to $EF(p_1 = wait \wedge EG(\neg\, p_1 = use))$. For more details, see Corollary 2.1 below.

We will now formalize the general setting.

Definition 2 *Given a CLP program P and a set of constrained facts F, we define the CLP programs $P \oplus F$ and $P \oslash F$ as follows.*

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

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

Theorem 2 (CTL properties and CLP program semantics)
Given a concurrent system $\mathcal{S}$ and its translation to the CLP program $\mathcal{P}_{\mathcal{S}}$, the following properties hold for all sets of constrained facts F.

$$
\begin{aligned}
EF(F) &= \mathit{lfp}(T_{P \oplus F}) \\
EG(F) &= \mathit{gfp}(T_{P \oslash F})
\end{aligned}
$$

Proof. Follows from the fixpoint characterizations of CTL properties (see [Eme90]) and Theorem 1. □

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

Corollary 2.1 (Safety and Liveness)

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

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

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

4 Defining a Model Checking Method

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

A set F of constrained facts is an *implicit representation* of the (possibly infinite) set of states S if $S = [F]_{\mathcal{D}}$. From now on, we always assume that F itself is finite. We will replace the operator T_P over sets of atoms (i.e. states) by the operator S_P over sets 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 $p(x'_1, \ldots, x'_n) \leftarrow c[x'_1/x_1, \ldots, x'_n/x_n] \in F$).

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

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

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

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

Now, a (possibly non-terminating) *model checker* can be defined in a straightforward way. It consists of the manipulation of constrained facts as implicit representations of (in general, infinite) sets of states. It is based on standard fixpoint iteration of S_P operators for the specific programs P according to the fixpoint definition of the CTL properties to be computed (see e.g. Corollary 2.1). 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 $F_0, F_1, F_2, \ldots$ 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 $[F_i]_{\mathcal{D}} = [F_{i+1}]_{\mathcal{D}}$.

The fixpoint of the S_P operator is taken wrt. the *subsumption* ordering between sets of constrained facts. 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 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.

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 CLP setting, with its combination of mathematical and logical reasoning, allows one to find these optimizations naturally.

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

Elimination of redundant facts. We call a set of constrained facts F *irredundant* if no element subsumes another one. We keep all sets of constrained facts $F_1, F_2, \ldots$ during the least fixpoint iteration irredundant by checking whether a new constrained fact in F_{i+1} that is not locally subsumed by F_i itself subsumes (and thus makes redundant) a constrained fact in F_i. This technique is standard in CLP fixpoint computations [MR89].

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

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

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

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

Abstraction. We define an approximation $S_P^{\#}$ of the S_P operator in the style of the abstract interpretation framework, whose results guarantee that we obtain conservative approximations of the fixpoints and, hence, of the CTL properties. This approximation turns our method into a (possibly non-terminating) semi-test for AF and AG properties, in the following direction: only a positive answer is a definite answer.

We introduce a new *widening* operator $\Uparrow$ (in the style of [CH78], but without a termination guarantee) and then define $S_P^{\#}(F) = F \Uparrow S_P(F)$ (so that $[S_P(F)]_{\mathcal{D}} \subseteq [S_P^{\#}(F)]_{\mathcal{D}}$). The operator $\Uparrow$ is defined in terms of constrained facts. For example, if

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

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

In contrast with the 'standard' widening operators in [CH78] and the refined versions in [HPR97, BGP98], the operator $\Uparrow$ can be directly implemented using the entailment test between constraints; furthermore, it is applied fact-by-fact, i.e., without requiring a preliminary computation of the convex hull of union of polyhedra. Besides being computationally expensive, the convex hull approximation may be an important factor wrt. loss of precision. Let us consider e.g. the two sets of constrained atoms

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

When applied to F and F', each of the widening operators in [BGP98, CH78, HPR97] returns the (polyhedra denoted by the) fact $p(\ell, X) \leftarrow true$. In contrast, our widening is precise here, i.e., it returns F'. Note that the use of constrained facts automatically induces a partitioning over the state space wrt. the set of control locations; such a partitioning has shown to be useful to increase the precision of the widening operator (essentially, by reducing its applicability; see e.g. [HPR97, BGP98]).

5 Experimentation in CLP

We have implemented the model checking procedure described above in SICStus Prolog 3.7.1 using the CLP(Q,R) library [Hol95] and the Boolean constraint solvers (which are implemented with BDDs). We made extensive use of the runtime database facilities for storing and retrieving constrained facts, and of the meta-programming facilities (e.g., the interchangeability between uninterpreted and interpreted constraints expressions).

We have applied the implementation to several infinite-state verification problems that are becoming benchmarks in the community (see e.g. [BGP97, BGP98, SKR98, SUM96, LS97]). This allowed us to evaluate the performance of our implementation, to experiment with evaluation strategies and abstractions through widenings, and to compare our solution with previous solutions.

We implement the solving of constraints over integers, which is needed for model checking integer-valued concurrent systems, through a constraint solver over reals. We thus trade the theoretical and practical gain in efficiency with an extra abstraction. This abstraction yields yields a *conservative* approximation of CTL properties (by standard fixpoint theory). In our experiments, we did not incur a loss of precision. It would be interesting to generally characterize the

Programs	C	ET	EN	ERT	ERN	AT	AN	ART	ARN
bakery	8	0.1	18	0.1	16	-	-	-	-
bakery3	21	6.3	157	6.1	109	-	-	-	-
bakery4	53	335.4	1698	253.2	963	-	-	-	-
ticket	6	↑	↑	↑	↑	1.0	15	1.1	13
mut-ast	20	0.0	20	0.0	20	-	-	-	-
network	16	↑	↑	↑	↑	0.7	3	0.6	3
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

Fig. 4. Benchmarks for the verification of safety properties; C: number of clauses, E: exact, A: approximation with widening, R: elimination of redundant facts, T: execution time (in seconds), N: number of produced facts, $-$: not needed, $\uparrow$: non-termination.

integer-valued concurrent systems for which the abstraction of integer constraints to the reals is always precise.
We will now briefly comment on the experimental results listed in Fig. 4. All the verification problems have been tested on a Sun Sparc Station 4, OS 5.5.1.

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

The *ticket* algorithm (see [BGP97]) is based on similar ideas as the bakery algorithm. Here, priorities are maintained through two global variables and two local variables. As in [BGP97], we needed to apply widening to prove safety. In a second experiment we applied the magic set transformation instead and obtained a proof in 0.6s. We proved starvation freedom in 3.0s applying widening for the outer least fixpoint (the inner one for the greatest fixpoint terminates without abstraction).

The algorithm *mut-ast* (see [LS97]) is also designed to ensure mutual exclusion. We have translated the description of a network of an arbitrary, non-fixed number of *mut-ast*-processes in [LS97] into a CLP-program and proved safety using abstraction (*network*).

The other examples are producer-consumer algorithms. The algorithm *bbuffer* (see [BGP98]) coordinates a system of two producers and two consumers connected by a buffer of bounded size. We proved two invariants: the difference between produced and consumed items is always equal to the number of items currently present in the buffer (*bbuffer*(1)), and the number of free slots always ranges between zero and the maximum size of the buffer (*bbuffer*(2)). The algorithm *ubuffer* (see [BGP98]) coordinates a system with one producer and one consumer connected by two unbounded buffers. We have proved the invariant that the number of consumed items is always less than that of produced ones.

A prototypical version of our model checker (SICStus source code, together with the code of the verification problems considered in this section and the outcomes of the fixpoint computations) is available at the URL address `www.mpi-sb.mpg.de/~delzanno/clp.html`.

6 Related Work

There have been other attempts to connect logic programming and verification, none of which has our generality with respect to the applicable concurrent systems and temporal properties. In [FR96], Fribourg and Richardson use CLP programs over *gap-order integer constraints* [Rev93] in order to compute the set of reachable states for a 'decidable' class of infinite-state systems. Constraints of the form $x = y + 1$ (as needed in our examples) are not gap-order constraints. In [FO97], Fribourg and Olsen study reachability for system with integer counters. These approaches are restricted to safety properties.

In [Rau94], Rauzy describes a CLP-style extension of the propositional μ-calculus to finite-domain constraints, which can be used for model checking for *finite-state* systems. In [Urb96], Urbina singles out a class of $CLP(\mathcal{R})$ programs that he baptizes 'hybrid systems' without, however, investigating their formal connection with hybrid system specifications; note that liveness properties of timed or hybrid automata can *not* be directly expressed through fixpoints of the S_P operator (because the clauses translating time transitions may loop). In [GP97], Gupta and Pontelli describe runs of timed automata using the top-down operational semantics of CLP-programs (and not the fixpoint semantics). In [CP98], Charatonik and Podelski show that set-based program analysis can be used as an always terminating algorithm for the approximation of CTL properties for (traditional) logic programs specifying extensions of pushdown processes. In [RRR$^+$97], a logic programming language based on *tabling* called XSB is used to implement an efficient local model checker for finite-state systems specified in a CCS-like value-passing language. The integration of tabling with constraints is possible in principle and has a promising potential.

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

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

7 Conclusion and Future Work

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

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

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

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

Acknowledgements. The authors would like to thank Stephan Melzer for pointing out the paper [BGP97], Christian Holzbaur for his help with the

OFAI-CLP($\mathcal{R}$) library [Hol95], and Tevfik Bultan, Richard Gerber, Supratik Mukhophadyay and C.R. Ramakrishnan for fruitful discussions and encouragements.

References

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

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In John C. Mitchell, editor, *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS'90)*, pages 428–439. IEEE Society Press, 1990.

[BGL98] T. Bultan, R. Gerber, and C. League. Verifying Systems with Integer Constraints and Boolean Predicates: a Composite Approach. In *Proceedings of the 1998 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'98)*, pages 113–123. ACM Press, 1998.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic Model Checking of Infinite-state Systems using Presburger Arithmetics. In Orna Grumberg, editor, *Proceedings of the Ninth Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 400–411. Springer-Verlag, 1997.

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

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

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

[CABN97] W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining Constraint Solving and Symbolic Model Checking for a Class of Systems with Non-linear Constraints. In Orna Grumberg, editor, *Proceedings of the Ninth Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 316–327. Springer-Verlag, 1997.

[CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of the Fifth Annual Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96. ACM Press, 1978.

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

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

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

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

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

[GDL95] M. Gabbrielli, M. G. Dore, and G. Levi. Observable Semantics for Constraint Logic Programs. *Journal of Logic and Computation*, 2(5):133–171, 1995.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Rev93] P. Z. Revesz. A Closed-form Evaluation for Datalog Queries with Integer (Gap)-order Constraints. *Theoretical Computer Science*, 116(1):117–149, 1993.

[RRR+97] Y. S. Ramakrishnan, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient Model Checking using Tabled Resolution. In Orna Grumberg, editor, *Proceedings of the Ninth Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 143–154. Springer-Verlag, 1997.

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

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

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

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

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

[Urb96] L. Urbina. Analysis of Hybrid Systems in CLP(R). In Eugene C. Freuder, editor, *Proceedings of Principles and Practice of Constraint Programming (CP'96)*, volume 1118 of *LNCS*, pages 451–467. Springer-Verlag, 1996.

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