Constraint-based Deductive Model Checking

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

Abstract.

Contents

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

1 Introduction

Automated verification methods can today be applied to practical systems [McM93]. One reason for this success is that implicit representations of finite sets of states through Boolean formulas can be handled efficiently via BDD’s [BCM+90]. The finiteness is an inherent restriction here. Many systems, however, operate on data values from an infinite domain and are intrinsically infinite-state; i.e., one cannot produce a finite-state model without abstracting away crucial properties. There has been much recent effort in verifying such systems (see e.g., [ACJT96,BW98,BGP97,CJ98,HHWT97, HPR97,LPY97,SKR98]). One important research goal is to find appropriate data structures for implicit representations of infinite sets of states, and design model checking algorithms that perform well on practical examples.

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

We present a translation from concurrent systems with infinite state spaces to CLP programs that preserves the semantics in terms of transition sequences. The formalism of ‘concurrent systems’ is a widely-used guarded-command specification language with shared variables promoted by Shankar [Sha93]. Using this translation, we exhibit the connection between states and ground atoms, between sets of states and constrained facts, between the pre-condition operator and the logical consequence operator of CLP programs, and, finally, between CTL properties (safety, liveness) and model-theoretic or denotational program semantics. This connection suggests a natural approach to model checking for infinite-state systems using CLP: model checking as 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 constraint facts to implicitly represent set of ground atoms. Constraint facts can be manipulated symbolically via the operations defined on constraints (e.g. variable elimination, 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 [KKR95, 1] If A and A → B hold, then B holds.
Rev93,RSS92] to improve the efficiency of bottom-up query evaluation.

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

As practical examples, we focus on the verification problem for systems with (unbounded) integer values; see e.g. [BW94,BW98,Bu98,BGP97,BGP98,Cert94,CJ98,FR96,SKR98]. The problem is undecidable for most classes of practical importance. Following [BW94,BW98], we apply our possibly non-terminating model checking procedure and we use abstractions to enforce or (simply speed-up) termination on practical examples.

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

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

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

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

2 Translating Concurrent Systems into CLP

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

\begin{verbatim}
begin turn1 := 0; turn2 := 0; P1 || P2 end
\end{verbatim}

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

\begin{verbatim}
repeat
  think : turn1 := turn2 + 1;
  wait : when turn1 < turn2 or turn2 = 0 do
    use : [critical section:
      turn1 := 0
    ]
  forever
\end{verbatim}
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 $\text{turn}_1$ and $\text{turn}_2$ 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 $S$ in Figure 3 following the scheme in [Sha93]. Each process is associated with a control variable ranging over the control locations (i.e., program labels). The data variables correspond to the program variables. The states of $S$ are tuples of control and data values, e.g., $(\text{think}, \text{think}, 0, 3)$. The primed version of a variable in an action stands for its successor value. We omit conjuncts like $p'_2 = p_2$ expressing that the value remains unchanged.

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

If the reader is not familiar with CLP, the following introduction together with Appendix A are all one needs to know for this paper. A CLP program is simply a logical formula, namely a universally quantified conjunction of implications (like the one in Figure 2; the implications are usually called clauses). Its first reading is the usual first-order logic semantics. We give it a second reading as a non-deterministic sequential program. The program states are atoms, i.e., applications of the predicate $p$ to values such as $p(\text{think}, \text{think}, 0, 3)$. 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 $s \propto s'$ is an instance of one of the implications.

For example, the state $p(\text{think}, \text{think}, 0, 3)$ has as a possible successor the state $p(\text{wait}, \text{think}, 4, 3)$, since $p(\text{think}, \text{think}, 0, 3) \propto p(\text{wait}, \text{think}, 4, 3)$ is an instance of the first implication for $p$ (instantiate the variables with $P_2 = \text{think}, \text{Turn}_1 = 0, \text{Turn}_1' = 4$ and $\text{Turn}_2 = 3$).

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 $S$ with the corresponding atom of the CLP program $P_S$; for example, $(\text{think}, \text{think}, 0, 3)$ with $p(\text{think}, \text{think}, 0, 3)$.

We observe that the transition sequences of the concurrent system $S$ in Figure 3 are exactly the ground derivations of the CLP program $P_S$ in Figure 2. Moreover, the set of all predecessor states of a set of states in $S$ is the set of its direct logical consequences under the CLP program $P_S$. 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 $x$ for the tuple of variables $\langle x_1, \ldots, x_n \rangle$ and $d$ for the tuple of values $\langle d_1, \ldots, d_n \rangle$. We denote validity of a first-order formula $\psi$ wrt. a structure $D$ and an assignment $\alpha$ by $D,\alpha \models \psi$. As usual, $\alpha[x \mapsto d]$ denotes an assignment in which the variables in $x$ are mapped to the values in $d$. In the examples of Section 7 formulas will be interpreted over the domains of integers and reals. Note however that the following presentation is given for any structure $D$.

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

- $V$ is the tuple $x$ of control and data variables,
- $\Theta$ is a formula over $V$ called the initial condition,
- $\mathcal{E}$ is a set of pairs $(\psi, \phi)$ called events, where the enabling condition $\psi$ is a formula over $V$ and the action $\phi$ is a formula of the form $x'_1 = e_1 \wedge \ldots \wedge x'_{n} = e_n$ 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 $\psi$ action $\phi$ for the event $(\psi, \phi)$ (omitting conjuncts of the form $x' = x$).

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

$$\tau = \{ (d, d') \mid D, \alpha[x \mapsto d] \models \psi, \quad D, \alpha[x \mapsto d, x' \mapsto d'] \models \phi, \quad (\psi, \phi) \in \mathcal{E} \}.$$  

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

For the translation to CLP programs, we view the formulas for the enabling condition and the action as constraints over the structure $D$ (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 $S$ is encoded as the CLP program $P_S$ given below, if $S = \langle V, \Theta, \mathcal{E} \rangle$ and $V$ is the tuple of variables $x$.

$$P_S = \{ p(x) \leftarrow \psi \wedge \phi \wedge p(x') \mid (\psi, \phi) \in \mathcal{E} \} \cup \{ \text{init} \leftarrow \Theta \wedge p(x) \}$$

Note that e.g. $p(\text{think}, P_2, \text{Turn}_1, \text{Turn}_2) \leftarrow \ldots$ in the notation used in examples is equivalent to $p(P_1, P_2, \text{Turn}_1, \text{Turn}_2) \leftarrow P_1 = \text{think} \wedge \ldots$ in the notation used in formal statements.
Control variables \( p_1, p_2 : \{ \text{think, wait, use} \} \)
Data variables \( \text{turn}_1, \text{turn}_2 : \text{int.} \)
Initial condition \( p_1 = \text{think} \land p_2 = \text{think} \land \text{turn}_1 = \text{turn}_2 = 0 \)

Events \[ \begin{align*}
\text{cond } p_1 = \text{think} & \quad \text{action } p'_1 = \text{wait} \land \text{turn}_1' = \text{turn}_1 + 1 \\
\text{cond } p_1 = \text{wait} \land \text{turn}_1 < \text{turn}_2 & \quad \text{action } p'_1 = \text{use} \\
\text{cond } p_1 = \text{wait} \land \text{turn}_2 = 0 & \quad \text{action } p'_1 = \text{use} \\
\text{cond } p_1 = \text{use} & \quad \text{action } p'_1 = \text{think} \land \text{turn}_1' = 0 \\
\end{align*} \]

\( \ldots \) symmetrically for Process 2

**Fig. 3.** Concurrent system \( S \) specifying the bakery algorithm

\[
\begin{align*}
\text{init } & \leftarrow \text{turn}_1 = 0, \text{turn}_2 = 0, p(\text{think}, \text{think}, \text{Turn}_1, \text{Turn}_2), \\
p(\text{think}, p_2, \text{Turn}_1, \text{Turn}_2) & \leftarrow \text{Turn}_1 = \text{Turn}_1 + 1, p(\text{wait}, p_2, \text{Turn}_1, \text{Turn}_2), \\
p(\text{wait}, p_2, \text{Turn}_1, \text{Turn}_2) & \leftarrow \text{Turn}_1 < \text{Turn}_2, p(\text{use}, p_2, \text{Turn}_1, \text{Turn}_2), \\
p(\text{use}, p_2, \text{Turn}_1, \text{Turn}_2) & \leftarrow \text{Turn}_2 = 0, p(\text{use}, p_2, \text{Turn}_1, \text{Turn}_2), \\
p(\text{use}, p_2, \text{Turn}_1, \text{Turn}_2) & \leftarrow \text{Turn}_1 = 0, p(\text{think}, p_2, \text{Turn}_1, \text{Turn}_2), \\
\end{align*}
\]

\( \ldots \) symmetrically for Process 2

**Fig. 4.** CLP program \( P_S \) for the concurrent system \( S \) in Figure 3.

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

\[
T_P(S) = \{ p(d) \mid p(d) \leftarrow p(d') \text{ is an instance of} \}
\]

\[
\text{a clause in } P, \ p(d') \in S \}.
\]

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 \( d \) and atoms \( p(d) \).

**Theorem 1 (Adequacy of the translation \( S \mapsto P_S \)).**

(i) The state sequences of the transition system defined by the concurrent system \( S \) are exactly the ground derivations of the CLP program \( P_S \).

(ii) The pre-condition operator of \( S \) is the logical consequence operator associated with \( P_S \), formally:

\[
\text{pre}_S = T_{P_S}.
\]

**Proof.** The clause \( p(x) \leftarrow \psi \land \phi \land p(x') \) of \( P_S \) corresponds to the event \( \langle \psi, \phi \rangle \). Its instances are of the form \( p(d) \leftarrow p(d') \) where \( D, a[x \mapsto d, x' \mapsto d'] \models \psi \land \phi \). Thus, they correspond directly to the pairs \( (d, d') \) of the transition relation \( \tau \) restricted to the event \( \langle \psi, \phi \rangle \).

To prove (i), we first show that state sequences correspond to ground derivation by induction of the length of a sequence. The base case follows by noting that the initial states are instances of the body of the \textit{init}-clause. Let us assume now that the thesis holds for a state-derivation \( d_0d_1 \ldots d_i \). If \( (d_i, d_{i+1}) \in \tau \) there exists a clause in the translation such that its set of instances contains \( p(d_i) \leftarrow p(d_{i+1}) \). Furthermore, by inductive hypothesis, \( p(d_0)p(d_1) \ldots p(d_i) \) is a ground derivation. Thus, by applying a resolution step we obtain the new sequence \( p(d_0)p(d_1) \ldots p(d_i)p(d_{i+1}) \). The converse can be proved by induction on the length of a derivation.

Point (ii) follows directly by definition of \( \text{pre}_S \) and \( T_{P_S} \).

As an aside, if we translate \( S \) into the CLP program \( P_{S_{\text{post}}} \) where

\[
P_{S_{\text{post}}} = \{ p(x) \land \psi \land \phi \rightarrow p(x') \mid \langle \psi, \phi \rangle \in \mathcal{E} \} \cup \{ \theta \rightarrow p(x) \}
\]

then the post-condition operator is the logical consequence operator associated with \( P_S \), formally: \( \text{post}_S = T_{P_{S_{\text{post}}}} \). We thus obtain the characterization of the set of reachable states as the least fixpoint of \( T_{P_{S_{\text{post}}}} \).

**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 \( p(x) \leftarrow c \) whose body contains only a constraint \( c \). Note that an instance of a constrained fact is of the form \( p(d) \leftarrow \text{true} \) which is the same as the atom \( p(d) \), i.e. it is a state. Given a set of constrained
facts $F$, we write $[F]_D$ 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 $F_{\text{mut}}$ consisting of the fact $p(P_1,P_2,\text{Turn}_1,\text{Turn}_2) \leftarrow P_1=\text{user},P_2=\text{use}$ is the set of states $[F_{\text{mut}}]_D = \{p(\text{use},\text{use},0,0),p(\text{use},\text{use},1,0)\}$. 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 $[F]_D$ is reachable. In our examples, we will use a more intuitive notation and write e.g. $EF(p_1 = p_2 = \text{use})$ instead of $EF(F_{\text{mut}})$.

As an example of a safety property, consider mutual exclusion for the concurrent system $S$ in Figure 3 ("the two processes are never in the critical section at the same time"), expressed by $AG(\neg (p_1 = p_2 = \text{use}))$. Its complement is the set of states $EF(p_1 = p_2 = \text{use})$.

As we can prove, this set is equal to the least fixpoint for the program $P_S \oplus F_{\text{mut}}$ that we obtain from the union of the CLP Program $P_S$ in Figure 2 and the singleton set of constrained facts $F_{\text{mut}}$. 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 $AG(p_1 = \text{wait} \rightarrow AF(p_1 = \text{use}))$. Its complement is the set of states $EF(p_1 = \text{wait} \land EG(\neg p_1 = \text{use}))$. The set of states $EG(\neg p_1 = \text{use})$ is equal to the greatest fixpoint for the CLP program $P_S \ominus F_{\text{starv}}$ in Figure 5. We obtain $P_S \ominus F_{\text{starv}}$ from the CLP Program $P_S$ by a transformation wrt. to the following set of two constrained facts:

$$F_{\text{starv}} = \{ p(P_1,P_2,\text{Turn}_1,\text{Turn}_2) \leftarrow P_1 = \text{think},
\quad p(P_1,P_2,\text{Turn}_1,\text{Turn}_2) \leftarrow P_1 = \text{wait} \}.$$ 

The transformation amounts to ‘constrain’ all clauses $p(\text{label}_{1,\ldots},\ldots) \leftarrow \ldots$ in $P_S$ such that label$_1$ is either wait or think (i.e., clauses of the form $p(\text{use}_{1,\ldots},\ldots) \leftarrow \ldots$ 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 $[F']_D$ is equal to the greatest fixpoint for the CLP program $P_S \ominus F$. The method uses the set $F'$ to form a third CLP program $P_S \oplus F'$. The least fixpoint for that program is equal to $EF(p_1 = \text{wait} \land EG(\neg p_1 = \text{use}))$. For more details, see Corollary 1 below.

We will now formalize the general setting.

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

$$P \oplus F = P \cup F \quad P \ominus F = \{ p(x) \leftarrow c_1 \land c_2 \land p(x') \mid p(x) \leftarrow c_2 \in F,
\quad p(x) \leftarrow c_1 \land p(x') \in P \}.$$ 

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

$$EF(F) = \text{Inf}(T_{P_S \ominus F})
\quad EG(F) = \text{Inf}(T_{P_S \oplus F})$$

**Proof.** From the fixpoint characterizations of CTL properties (see [Eme90]) we know that $EF(F) = \mu Z. F \cup EX(Z)$ and $EG(F) = \nu Z. F \cap EX(Z)$ where $EX(Z) \equiv \text{pre}_S(Z)$. From Theorem 1, using the operators of Def. 2, the CLP programs $P_S \ominus F$ and $P_S \oplus F$ are such that $T_{P_S \ominus F}(Z) = \text{pre}_S(Z) \cup F$ and $T_{P_S \oplus F}(Z) = \text{pre}_S(Z) \cap F$. As a consequence, we have that $EF(F) = \text{Inf}(T_{P_S \ominus F})$ and $EG(F) = \text{Inf}(T_{P_S \oplus F})$. □

By duality, we have that $AF(\neg F)$ is the complement of $\text{Inf}(T_{P_S \oplus F})$ and $AG(\neg F)$ is the complement of $\text{Inf}(T_{P_S \ominus F})$. 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 $S$ satisfies the safety property $AG(\neg F)$ if and only if the atom ‘init’ is not in the least fixpoint for the CLP program $P_S \oplus F$.

(ii) $S$ 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 $P_S \oplus (F_1 \land F')$, where $F'$ is a set of constrained facts denoting the greatest fixpoint for the CLP program $P_S \ominus F_2$.

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

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 $S = [F]_D$. 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 (see Appendix A). If the CLP programs \( P \) is an encoding of a concurrent system, we can define \( S_P \) as follows (note that \( F \) is closed under renaming of variables since clauses are implicitly universally quantified; i.e., if \( p(x_1, \ldots, x_n) \leftarrow c \in F \) then also \( p(x'_1, \ldots, x'_n) \leftarrow c[x'/x_1, \ldots, x'_n/x_n] \in F \)).

\[
S_P(F) = \{ p(x) \leftarrow c_1 \land c_2 \mid p(x) \leftarrow c_1 \land p(x') \in P, \\
p(x') \leftarrow c_2 \in F, \\
c_1 \land c_2 \text{ is satisfiable in } D \}
\]

If \( P \) contains also constrained facts \( p(x) \leftarrow c \), 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 \( \{ f_p(T_P) \} = \{ f_p(S_P) \} \) and \( g_f(T_P) = [g_f(S_P)]_D \) (see Appendix A). Thus, Theorem 2 leads to the characterization of CTL properties through the \( S_P \) operator via:

\[
EF(F) = [\{ f_p(S_P \circ P) \}]_D, \\
EG(F) = [\{ g_f(S_P \circ P) \}]_D.
\]

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 standard fixpoint iteration of \( S_P \) operators for the specific programs \( P \) according to the fixpoint definition of the CTL properties to be computed (see e.g. Corollary 1). An iteration starts either with \( F = \emptyset \) representing the empty set of states, or with \( F = \{ p(x) \leftarrow true \} \) 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 \( [F_i]_D = [F_{i+1}]_D \).

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 \( [F]_D \subseteq [F']_D \). Testing subsumption amounts to testing entailment of disjunctions of constraints by constraints.

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

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

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

### 4.1 Strategies

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

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

\[
\bar{P} = \{ p(x) \leftarrow \text{body}, \bar{p}(x') \mid p(x) \leftarrow \text{body} \in P \} \cup \\
\{ \bar{p}(x') \leftarrow c, \bar{p}(x) \mid p(x) \leftarrow c, p(x') \in P \} \cup \\
\{ \bar{\text{init}} \leftarrow \text{true} \}
\]

We obtain the soundness of this transformation wrt. the verification of safety properties by standard results [RSS92] which say that \( \text{init} \in \text{fp}(T_F) \) if and only if \( \text{init} \in \text{fp}(T_{\bar{P}}) \) (which is, \( \text{init} \in \text{fp}(S_{\bar{P}}) \)). The soundness continues to hold if we replace the constraints \( c \) in the clauses \( \bar{p}(x') \leftarrow c, \bar{p}(x) \) in \( P \) 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 \( \text{true} \), then the first iterations amount to eliminating constrained facts \( p(\text{label}_1, \text{label}_2, \ldots) \leftarrow \ldots \) whose locations \( \langle \text{label}_1, \text{label}_2 \rangle \) are “definitely” not reachable from the initial state.

### 5 Implementation in a CLP system

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

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

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

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

We will also use \( C \land D \) to denote the conjunction of the constraints \( C \) and \( D \).

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

Let us first consider the computation of \( EF \). The predicate \( \text{apply\_Sp} \) in Fig. 5 implements an application of the \( S_P \) operator to a given clause and a given fact. Both the clause and the fact are selected from the database (non-deterministically, in principle, following the order they are stored in the database, in practice) using the predicates \( \text{select\_clause} \) and \( \text{select\_fact} \). The parameter \( I \) of the predicate \( \text{apply\_Sp} \) denotes the index of the current fixpoint iteration. Note that, by monotonicity of \( S_P \), at step \( I + 1 \) we don’t need to select facts with index less than \( I \), i.e., each fact is selected only once during the whole fixpoint computation. The predicate \( \text{unify} \) (occurring in the body of \( \text{apply\_Sp} \)) finds the most general unifier for the variables of the atoms \( A \) and \( B \) (i.e., after its execution the constraints \( C \) and \( D \) will range over the same set of variables). After checking for satisfiability of the conjunction \( C \land D \), the resulting solved constraint \( C' \) is projected over the variables contained in the head of the clause \( H \). Note that, here, constraints are considered as uninterpreted (when manipulated as facts) as well as interpreted terms (when passed to the constraint solver, e.g., via the predicates \( \text{satisfiable} \) and \( \text{variable\_elimination} \)). The last step consists of adding the newly constructed fact \( (H \leftarrow C'' \) to the current
database. This task is carried out by the predicate `handle_new_fact`. The new fact is added only if it does not entail an existing fact (i.e., the denotations of the new fact are not contained in the denotations of an existing fact, checked using the predicate `entail`). This way, we implement the local subsumption test we mentioned before.

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

The CLP built-in predicate `!` (called cut) is used to make the predicate `least_fixpoint` deterministic after the test for the initial state. This way, the second clause for `least_fixpoint` is selected only when the predicate `apply_Sp` has no derivations (i.e., no new facts can be derived using `S_P`) or the initial state occurs in the database.

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

We will present other heuristics in Section 6.

We use similar techniques to implement the computation of the greatest fixpoint of `S_P` (e.g., used in `EG`). The main loop used to compute the greatest fixpoint is given in Fig. 9. The predicate `assert_template` is used to initialize the database with a set of facts representing the Herbrand base. The predicate `not_contained(I,I+1)` is used to test that the set of facts computed at step I are not subsumed by the facts computed at step I + 1 (i.e., the fixpoint has not been reached, yet). In its implementation, we used again a local subsumption test.
Similarly, we obtain apply_sp for gfp from apply_sp by substituting not_subsumed(H,C) with the predicate invocation not_subsumed(I+1,H,C), i.e., we check that the fact $H \vdash C^I$ is not subsumed by facts computed at step $I+1$ (in the greatest fixpoint computation the denotations of $F_{i+1}$ coincide with the intersection of the denotations of $F_0, \ldots, F_{i+1}$).

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

We have implemented the model checking procedure described above in SICStus Prolog 3.7.1 using the arithmetic constraint solver CLP(Q,R) [Hal95] and the Boolean constraint solvers (which are implemented with BDDs). In the following section we will report on experimental results obtained by analysing several type of infinite-state systems.

### 6 Infinite-State Integer Systems

The verification problem for systems with (unbounded) integer values is receiving increasing attention; see e.g. [BW94,BW98,Bul98,BGP97,BGP98, Cer94,CJ98,FR96,SKR98]. The problem is undecidable for most classes of practical importance. So what can you do? There are basically two answers. (1) Give a possibly non-terminating algorithm that terminates for useful examples. This is the approach followed e.g. by [BW98, BGW97]; (2) Give a semi-test that yields the definite answer for useful examples (the other answer being 'don't know'); see e.g. [BGP97,CLG92,LGS+94,Gra94, Dam96,Hal93,HPR97,HH95].

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

To know the accuracy of an abstraction is important conceptually and pragmatically. Note that there seems to be no other way to predict its effect (“too rough?”) for a particular application. Obviously, the accuracy is useful for debugging (or finding typos); ‘don’t know’ answers are quite 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 [CC77], we show that a set of acceleration rules of the model-checking fixpoint operator yields an accurate model checking algorithm (i.e. a full test if terminating). The acceleration rules are formulated via a deductive system, i.e., they are specialized rules to compute logical consequences of CLP programs with linear constraints. The correctness and accuracy of the method is ensured by the soundness of the inference rules. Given the rule-based nature of CLP programs, the acceleration rules can be naturally accomodated in our CLP implementation of the model checking procedure.

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

#### 6.1 Relaxation

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

The relaxation from integers to reals stems from linear programming (see e.g. [Sch86]). The motivation there is the same as here: the manipulation of linear arithmetic constraints is less costly over reals than over integers, theoretically (e.g. polynomial vs. NP-hard for the satisfiability test) as well as practically (e.g., the variable elimination is less involved). Even if the complexity for integers is the same as for reals for a particular application (as discussed in details in the description of the Omega library, a solver for Presburger arithmetics [Pug91]), there exist many highly optimized constraint systems over reals, general-purpose such as CLP(R) and special purpose such as Uppaal [BLL98] or Hytech [HHWT97], which one would like to exploit for model checking (simple) concurrent systems over integers.

Note that this abstraction is not an embedding of the verification problem for a system over integers into one for a system over reals.
In the rest of the section we will define the class of CLP programs for which we can prove that the relaxation int-real of the constraint operators used in $S_P$ is accurate.

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

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

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 $x \leq y + c \land \land x \geq y + c$. Vector Addition Systems [KM69] (a.k.a. Petri Nets), and Integral Relational Automata [Čer94] are two examples of systems whose translation in CLP gives rise to simple CLP programs. The reachability problem is decidable for these classes (see e.g. [Čer94, Lam92]). Other examples are multi-clocks automata [CJ98] and gap-order automata [FR96].

The above-mentioned decidability results are related to the general results for verification problems of infinite-state systems in [ACJT96, FS98]. The communication protocols considered in [BGP97, BGP98, SKR98], such as the bakery algorithm of Section 2, are examples of systems that can be translated to simple CLP programs but that do not seem to belong to a known decidable subclass.

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

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

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

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


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

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

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.

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

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

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

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

Consider the program \( p(x,y) \leftarrow y = y + 1 \land p(x,y') \) and the fact \( p(x,y) \leftarrow x \leq y \). The computation of \( I_P(S_P \oplus F) \) 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 will follow by proving the soundness of the resulting deduction system.

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

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

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

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

**Proof.** In this proof we use the properties mentioned in Appendix A. We only prove the soundness of the rule 1. Let \( Q \) consists of the clauses \( p(x) \leftarrow p(x') \land C \) and \( p(x) \leftarrow x \leq y + c \land D \) satisfying the hypothesis of rule 1. Let \( f(n) = p(x) \leftarrow D, x \leq y + c + n(c_y - c_x) \). We first prove that \( [f(n)] \) is a subset of \( [S^n_P(\emptyset)] \) for all \( n \geq 0 \). The proof is by induction on \( n \). Base case: by definition. Inductive step. Let us assume that \( [f(n)] \) is a subset of \( [S^n_P(\emptyset)] \). By monotonicity of \( T_Q \), \( T_Q([f(n)]) \subseteq \)
Theorem 4 (Acceleration). The algorithm obtained by abstracting the least fixpoint operator $S_P$ in the symbolic model checking algorithm for safety properties with the acceleration defined in Figure 10 yields (if terminating) a full test of safety properties for concurrent systems over integers or reals.

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

6.3 Strategy for acceleration

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

Let us first consider rule 1 of Fig. 10. We assume that $F$ is the set of facts computed at a given iteration for computing the least fixpoint of $S_P$. Now, let $f_1 = p(x) \leftarrow c$ and $f_2 = p(x) \leftarrow D$ be two facts in $F$, obtained, respectively, after $i$ and $j$ ($i < j$) applications of $S_P$. Furthermore, let us assume that $c$ entails $D$ and that there exists a sequence of clauses $c_1, \ldots, c_n$ in $P$ such that $f_2 = S_n((S_{n-1}((\ldots(S_1(f_1)))))$ (i.e., $f_2$ is reachable from $f_1$).

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

$$p(x) \leftarrow c = S_n, \ldots, S_1(p(x) \leftarrow c_1) \ldots$$

The unfolded clause is then defined as $p(x) \leftarrow c \land p(x')$.

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

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

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

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

6.4 Conservative Approximations

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

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

$$[S_P, N(I)] \subseteq [S_P, N(I)]_N.$$
answers (e.g., the initial state is not in the least model computed over \( \mathbb{R} \)) are definite answers.

Following ideas developed in abstract interpretation [CC77], it is also possible to apply acceleration operators that may return don’t know answers, i.e., when incorpo-
   rated in fixpoint computation, they yield a conservative
   approximation of the property taken into consideration.
In this section we will introduce a new widening operator \( \uparrow \) (in the style of [CH78], but without a termination guarantee) used to define \( S^\uparrow_P(F) = F \uparrow S_P(F) \) (so that \( [S_P(F)]_\mathcal{D} \subseteq [S^\uparrow_P(F)]_\mathcal{D} \)). 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
\[
F = \{ p(X,Y) \leftarrow X \geq 0, Y \geq 0, X \leq Y \} \\
F' = \{ p(X,Y) \leftarrow X \geq 0, Y \geq 0, X \leq Y + 1 \}
\]
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 \( p(x) \leftarrow c_1 \land \ldots \land c_n \)
in \( F \) by removing all conjuncts \( c_i \) that are strictly entailed by some conjunct \( d_j \) of some ‘compatible’ constrained atom \( p(x) \leftarrow d_1 \land \ldots \land d_m \) in \( F \), where ‘compatible’ means that the conjunction \( c_1 \land \ldots \land c_n \land d_1 \land \ldots \land d_m \) is satisfiable. This condition restricts the applications of the widening operator e.g. to facts with the same values for the control locations.

Contrary to the ‘standard’ widening operators in [CH78] and to the improved versions in [HP97, BGP98], the operator \( \uparrow \) can be directly implemented using the entailment test between constraints; furthermore, it is applied fact-by-fact, i.e., without requiring a preliminary computation of the convex hull of union of polyhedra. Note that the convex hull is computationally very expensive and it might be a source of further loss of precision. Let us consider e.g. the two sets of constrained atoms
\[
F = \{ p(\ell,X) \leftarrow X \geq 2 \} \\
F' = \{ p(\ell,X) \leftarrow X \geq 2, p(\ell,X) \leftarrow X \leq 0 \}.
\]
When applied to \( F \) and \( F' \), each of the widening operator in [BGP98, CH78, HP97] returns the (polyhedra denoted by the) fact \( p(\ell,X) \leftarrow \text{true} \). In contrast, our widening is precise here, i.e., it returns \( F' \).

Finally, note that the use of constrained facts automatically induces a partitioning over the state space wrt. the set of control locations. The partitioning reduces the applicability of the widening for the benefit of precision of the computation (see also [HP97, BGP98]).

7 Case-studies

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

7.1 Communication Protocols

Bakery algorithm. Mutual exclusion and starvation freedom for the bakery algorithm (see Sect. 2 and Sect. 3) can be verified without the use of accelerations (execution time for starvation freedom: 0.9s). In versions of the bakery algorithm for 3 and 4 processes (not treated in [BGP97]), a maximum operator (used in assignments of priorities such as \( \text{Turn}_1 = \max(\text{Turn}_2, \text{Turn}_3) + 1 \)) is encoded case-by-case in the constraint representation. This makes the program size grow exponentially in the number of processes.

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

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

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

Bounded Buffer. The first protocol we consider models the communication of producers and consumers connected via a buffer of size \( s \). Fig. 14 shows the automata for a producer and a consumer; here the variable \( a \) denotes the number of empty cells in the buffer, \( p \) the
number of produced items, and \( c \) the number of consumed items. The CLP-program that describes a system with a producer, a consumer and two buffers is given in Figure 15. The first invariant that we want to prove is \( Inv_1 = AG(p_1 + p_2 - (c_1 + c_2) = s - a) \). Note that the previous constraint is not simple, i.e., the relaxation int-real will give us a conservative approximation of the property. Before applying our model checker, we transform the previous property in \( AG(p_1 + p_2 - (c_1 + c_2) + a \leq s - 1 \land p_1 + p_2 - (c_1 + c_2) + a \geq s + 1) \). Our model checker proves the property in 0.28s 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:

\[
\begin{align*}
p(A, S) &\leftarrow p(A', S), A \geq 1, A' = A - 1, \\
p(A, S) &\leftarrow p(A', S), A \leq S - 1, A' = A + 1.
\end{align*}
\]

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

**Unbounded Buffer.** We consider now a protocol for producers and consumers connected via unbounded buffers. The system can be represented by an automaton with two states: *idle*, in which only the consumer is active (to weaken the producer), and *send*, in which both processes are active. Fig. 16 shows the automata for a pair producer-consumer and only one buffer. The variable \( p \) keeps track of the number of produced items, the variable \( c \) the number of consumed items and \( q \) the number of elements in the buffer. The CLP program in Fig. 17 models a system with a producer a consumer and two unbounded buffers. To prove the invariant \( AG(p \geq c) \) we prove that \( AG(p = c + q_1 + q_2) \), (note that \( q_i \geq 0 \)), i.e., \( AG(p \leq c + q_1 + q_2 - 1 \land p \geq c + q_1 + q_2 + 1) \) (a non-simple constraint). We prove the property in 3 steps by applying the widening operator of Section 6.4.

7.2 Array Bounds Checking.

In this section, we will discuss an application of our model checker for checking array bounds of imperative programs. The main idea is to extract, from the flow
graph of a program, all information involving manipulation of indexes of arrays. All remaining information will be abstracted away. In many cases the resulting system can be translated into a simple CLP program, as we will show with the help of a non-trivial example: the insertion sort algorithm.

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

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

7.3 Parameterized systems

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

Broadcast protocols [EN98] are systems composed of a finite but arbitrarily large number of processes that communicate by rendezvous (two processes exchange a message) or by broadcast (a process sends a message to all other processes).
void InsertionSort(int* A, int n) {
    int i, k, x;
    for (k = 1; k < n; k++) {
        entryA1: x = A[k];
        i = k - 1;
        while (i >= 0 && A[i] > x) {
            entryA2: A[i + 1] = A[i];
            i--;
        }
        entryA3: A[i + 1] = x;
    }
}

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

<table>
<thead>
<tr>
<th>System variables:</th>
<th>p(Location, K, N, I).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Condition:</td>
<td>init ← p(init, K, N, I).</td>
</tr>
<tr>
<td>Transitions:</td>
<td>p(init, K, N, I) ← p(for, K1, N, I), K1 = 1.</td>
</tr>
<tr>
<td></td>
<td>p(for, K, N, I) ← p(entryA1, K, N, I), K ≤ N - 1.</td>
</tr>
<tr>
<td></td>
<td>p(entryA1, K, N, I) ← p(while, K, N, I), I1 = K - 1.</td>
</tr>
<tr>
<td></td>
<td>p(while, K, N, I) ← p(entryA2, K, N, I), I ≥ 0.</td>
</tr>
<tr>
<td></td>
<td>p(entryA2, K, N, I) ← p(while, K, N, I), I1 = I - 1.</td>
</tr>
<tr>
<td></td>
<td>p(while, K, N, I) ← p(entryA3, K, N, I), I ≤ -1.</td>
</tr>
<tr>
<td></td>
<td>p(entryA3, K, N, I) ← p(for, K1, N, I), K1 = K + 1.</td>
</tr>
<tr>
<td></td>
<td>p(for, K, N, I) ← p(end, K, N, I), K ≥ N.</td>
</tr>
<tr>
<td></td>
<td>p(entryA1, K, N, I) ← K ≥ N.</td>
</tr>
<tr>
<td></td>
<td>p(entryA3, K, N, I) ← I ≥ N - 1.</td>
</tr>
<tr>
<td></td>
<td>p(entryA2, K, N, I) ← I ≤ -1.</td>
</tr>
<tr>
<td></td>
<td>p(entryA2, K, N, I) ← I ≥ N - 1.</td>
</tr>
<tr>
<td></td>
<td>p(entryA2, K, N, I) ← I ≥ N.</td>
</tr>
</tbody>
</table>

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

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

We model this protocol as the CLP program of Fig. 21. We use four variables I, S, E, and M to count the number of processes in the corresponding state. Each action corresponds to a re-allocation of the counters. Specifically, the second rule correspond to the read!! broadcast action, (i.e. one process sends the broadcast while being in state I, the other processes move to S), the third rule corresponds to the write-invalidate!!
broadcast, and the remaining rule to the read and write internal actions (only with write there is a re-allocation of processes). Note that, in the initial configuration, the number of processes in state $I$ is unspecified ($I \geq 1$). The safety properties proved in [EN98] are: $AG(\neg(S \geq 1 \land M \geq 1))$ (readers and writes cannot access simultaneously the cache), $AG(\neg(M + E \geq 1))$ (at most one process can stay in $M$ or $E$). These properties can be expressed as upwards closed sets. The reachability problem is decidable in this case [ACJT96,EFM99]. Upwards closed sets are fully characterized by the following class of constraints:

$$\phi ::= \ x_1 + \ldots + x_n \geq c \ | \ \phi \land \phi \ | \ \text{true},$$

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

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

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

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

8 Related Work

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

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

---

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

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

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

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

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

In [DEP96], Delzanno, Esparza, and Podelski discuss in the details of the theoretical complexity of the analysis of broadcast protocols over integer arithmetics. In this paper we show that the relaxation int-real of the predecessor operator for broadcast protocols gives accurate


9 Conclusion and Future Work

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

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

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

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

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

References


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


A Preliminaries on CLP

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

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

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

In the following we will use $t$ to denote a list of terms $t_1, \ldots, t_n$. For simplicity, we assume that all constants are interpreted over $\mathcal{D}$.

A.1 Operational semantics

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

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

A.2 Fixpoint semantics

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

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

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

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

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

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

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

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

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

---

**References**


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

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

Note that $[I \land J]_D = [I]_D \cap [J]_D$. The operator $S_P$ is monotonic wrt. $\subseteq$. Furthermore, it holds that

- $\text{gfp}(S_P) = \bigwedge_{\alpha = 0}^{\omega} S_\alpha(B_S)$ for $\alpha \geq \omega$.
- $[\text{gfp}(S_P)]_D = \text{gfp}(T_P)$.

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