Automatic Generation of Inductive Invariants by SUP(LA) [1]

Arnaud Fietzke Evgeny Kruglov
Christoph Weidenbach

MPI–I–2012–RG1-002 March 2012

Authors' Addresses

Arnaud Fietzke
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
Germany

Evgeny Kruglov
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
Germany

Christoph Weidenbach
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken
Germany

Abstract

The hierarchic combination of linear arithmetic and first-order logic with free function symbols, FOL(LA), results in a strictly more expressive logic than its two parts. The SUP(LA) calculus can be turned into a decision procedure for interesting fragments of FOL(LA). For example, reachability problems for timed automata can be decided by SUP(LA) using an appropriate translation into FOL(LA). In this paper, we extend the SUP(LA) calculus with an additional inference rule, automatically generating inductive invariants from partial SUP(LA) derivations. The rule enables decidability of more expressive fragments, including reachability for timed automata with unbounded integer variables. We have implemented the rule in the SPASS(LA) theorem prover with promising results, showing that it can considerably speed up proof search and enable termination of saturation for practically relevant problems.

Keywords

Superposition, linear arithmetic, induction

Contents

1 Introduction

One important aspect for successful development of automated reasoning calculi for logical languages is the potential of the calculus to act as a decision procedure for known decidable classes and to be an instrument for detecting new decidable fragments. This is because a sound and complete calculus for some logical language that can at the same time be used as a decision procedure has a high potential to be successfully applied in practice. The superposition calculus has been very successful in this respect for first-order logic, e.g., [3, 12, 19]. This is further illustrated by the fact that the leading first-order ATPs (E, SPASS, Vampire) are all superposition-based.

In this paper we continue this line of work for the FOL(LA) language, the hierarchic combination of first-order logic with linear arithmetic. The hierarchic superposition calculus SUP(LA) [1] is a sound calculus for FOL(LA) and together with a sufficient completeness assumption, also complete. Completeness cannot be achieved in general, because the FOL(LA) language can express second-order properties. For example, starting with LA over the reals, the naturals can be expressed in FOL(LA) [20] and it is known that the addition of a single monadic predicate to the LA language already causes undecidability [17], in general.

Nevertheless, the SUP(LA) calculus is a decision procedure for the FOL(LA) ground case [21] and for the FOL(LA) fragment resulting from the translation of timed automata [15]. In this paper we extend the latter result to the fragment corresponding to the translation of timed automata extended with unbounded integer variables. Termination of the SUP(LA) calculus on this fragment is made possible by a new simplification technique based on the automatic generation of inductive invariants. The invariant generation rule combines ideas from acceleration for automata [18, 6] with the automatic detection of infinite loops [22] in SUP(LA) derivations.

The following example illustrates the basic idea: assume we have used the clause $x = 1 \parallel \ \rightarrow P(x)$ in a derivation of $x = 2 \parallel \ \rightarrow P(x)$ (clauses are in purified form: arithmetic literals to the left of $\parallel$, first-order literals to

the right; $x = 1 \parallel \ \rightarrow P(x)$ means $\forall x(x = 1 \rightarrow P(x))$. Depending on how it was derived, the same sequence of inferences may be applied to the second clause, yielding a third clause with right-hand side $P(x)$. For instance, the second clause may have been obtained by resolving the first one with $x' = x + 1 \parallel P(x) \rightarrow P(x')$. Then we could also derive $x = 3 \parallel \ \rightarrow P(x)$, $x = 4 \parallel \ \rightarrow P(x)$ and so on. The idea of the invariant generation rule is to detect such loops during proof search, in the form of clauses with the same free (i.e., non-arithmetic) part (up to variable renaming), and to determine the transformation relating their arithmetic constraints. If it is possible to express the transitive closure of this transformation as a conjunction of arithmetic literals, then a corresponding invariant clause is derived. In the above example, such a clause would be $k \geq 1, x = k \parallel \ \rightarrow P(x)$, where k is an integer variable.

This paper is organized as follows: Section 2 gives some preliminary definitions relating to superposition modulo linear arithmetic. Section 3 defines the constraint induction rule in its general form, and presents a class of linear arithmetic constraints for which it can be effectively implemented. In Section 4, we define timed automata extended with unbounded integer variables, and we show that SUP(LA) together with the constraint induction rule provides a decision procedure for the corresponding reachability problem. Section 5 deals with our implementation of the rule and shows some promising experimental results. We end with a summary of the results and an outlook in Section 6. Detailed definitions and proofs can be found in a technical report [14].

2 Preliminaries

We will use the notions and notations for hierarchic superposition modulo linear arithmetic SUP(LA) [4, 1]. In SUP(LA), clauses appear in purified form $\Lambda \parallel \Gamma \to \Delta$ where Λ is a sequence of linear arithmetic literals over real and integer variables, called the *clause constraint*, and Γ, Δ are sequences of free first-order atoms, called the *free part*, sharing universally quantified variables with Λ. Semantically, a clause $\Lambda \parallel \Gamma \to \Delta$ is interpreted as the universal closure of the implication $(\bigwedge \Lambda \wedge \bigwedge \Gamma) \to \bigvee \Delta$. A constrained empty clause $\Lambda \parallel \Box$ represents a contradiction if Λ is satisfiable.

We use lowercase Latin characters x, y, z to denote variables. Vectors of variables are denoted by boldface characters ($\mathbf{x}$). We use the notation $\Lambda[\mathbf{x}]$ to mean that $\mathbf{x}$ are the variables occurring in Λ. When $\mathbf{x}$ is clear from the context, we also denote by $\Lambda[\mathbf{y}]$ the result of substituting all occurrences of variables from $\mathbf{x}$ in Λ by the corresponding variables from $\mathbf{y}$. Substitutions are denoted by lowercase Greek letters (σ, τ). A substitution is called *simple*, if it maps every variable of arithmetic sort to an arithmetic term.

The overall superposition calculus is based on a reduction ordering that is total on ground atoms. In particular all ground terms of the arithmetic sort containing only arithmetic symbols are assumed to be strictly smaller than any ground term containing a free function symbol. For example, this can be achieved by an LPO (lexicographic path ordering) where the arithmetic symbols are smaller in the precedence than any free symbol. This ordering on the ground atoms is then lifted to clauses via the usual twofold multiset extension. A ground clause C is *redundant* in some clause set N, if it follows from smaller clauses in N. Redundancy is lifted by instantiation to clauses with variables. The minimality of the arithmetic symbols ensures that, whenever a clause C is made redundant by smaller clauses $C_1, \ldots, C_n$, it remains redundant if we modify the constraints of the C_i in an equivalence-preserving way. For instance, we may simplify constraints by eliminating variables not occurring in the free part of the clause.

To keep the presentation simple, we use superposition left (ordered res-

olution) as the only inference rule, and subsumption as the only reduction rule. We will not need factoring for the types of clause sets considered in this paper.

A clause $C_1 = \Lambda_1 \parallel \Gamma_1 \to \Delta_1$ *subsumes* a clause $C_2 = \Lambda_2 \parallel \Gamma_2 \to \Delta_2$ if there is a substitution σ such that $\Gamma_1\sigma \subseteq \Gamma_2$, $\Delta_1\sigma \subseteq \Delta_2$ and $\forall \mathbf{x}\, \exists \mathbf{y}\, (\Lambda_2 \to \Lambda_1\sigma)$ holds in the theory of linear arithmetic, where $\mathbf{x}$ are the variables occurring in Λ_2 and $\mathbf{y}$ the variables occurring in $\Lambda_1\sigma$ but not in Λ_2. Note that in theorem proving derivations, forward subsumption (i.e. removing a newly derived clause which is subsumed by an old clause) does not need to be strict to maintain completeness.

The ordered resolution rule is

$$\frac{\Lambda_1 \parallel \Gamma_1, A \to \Delta_1 \qquad \Lambda_2 \parallel \Gamma_2 \to \Delta_2, B}{\Lambda_3 \parallel (\Gamma_1, \Gamma_2 \to \Delta_1, \Delta_2)\sigma}$$

such that σ is the most general simple unifier of A and B; A is strictly maximal in $\Gamma_1, A \to \Delta_1$; B is strictly maximal in $\Gamma_2 \to \Delta_2, B$.

The calculus SUP(LA) is complete for clause sets that enjoy *sufficient completeness*, meaning that every ground non-arithmetic term is equal to some arithmetic ground term. A sufficient condition for a clause set to be sufficiently complete is the absence of function symbols ranging into the arithmetic sorts (real or integer).

3 Constraint Induction

Given a relation $R \subseteq \mathbb{R}^{2n}$, the composition $R \circ R$ is the relation such that $(R \circ R)(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ holds if and only if there exist $y_1, \ldots, y_n$ such that $R(x_1, \ldots, x_n, y_1, \ldots, y_n)$ and $R(y_1, \ldots, y_n, x'_1, \ldots, x'_n)$. If we define $R^1 = R$ and $R^k = R^{k-1} \circ R$, then the *transitive closure* of R is the relation R^+ such that $R^+(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ if and only if there exists $k \geq 1$ such that $R^k(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$.

If a clause in a derivation has an ancestor (i.e., a clause to which it is related by a sequence of rule applications) with the same free part (modulo variable renaming), then the clause can be used to derive a third clause with the same free part, and so on. This yields a potentially infinite (depending on the constraints) sequence of inferences, where clauses differing only in the arithmetic constraint are being derived.

The idea of the invariant generation rule is to find the transformation relating the constraints along the sequence and to compute its transitive closure. To find the transformation, the sequence of inferences is applied to a parameterized version of the initial clause, as shown in Figure 3.1. If the closure can itself be expressed as a constraint, then we can derive a corresponding *inductive invariant clause* which can be used to subsume all its instances, thereby avoiding repeated applications of the same sequence of inference rules.

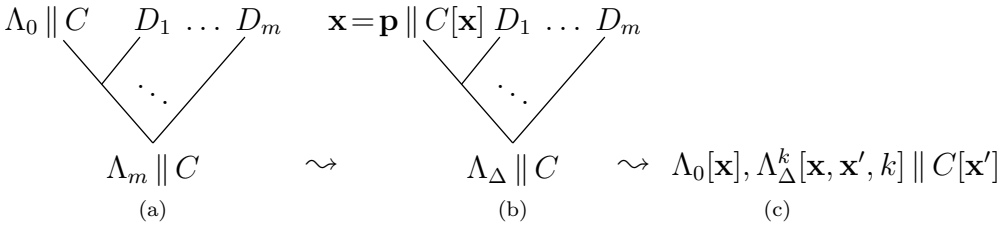

Figure 3.1: After a loop has been detected during proof search (a), the corresponding inferences are replayed on a parameterized clause (b) and the inductive invariant clause is derived (c).

The parameterized clause is of the form $x_1 = p_1, \ldots, x_n = p_n \parallel C[x_1, \ldots, x_n]$, where p_i are fresh parameters (i.e., arithmetic constants) not appearing anywhere in the clause set, one for each arithmetic variable in the clause $\Lambda_0 \parallel C$. After the inferences leading from $\Lambda_0 \parallel C$ to $\Lambda_m \parallel C$ have been performed on the parameterized clause, a clause of the form $\Lambda_\Delta \parallel C$ is obtained. This replaying of inferences is always possible, because the SUP(LA) calculus does not take the clause constraints into account when deciding which inferences to perform (the constraints are only considered when testing for subsumption, or when checking satisfiability of an empty clause's constraint). Also note that the parameters p_i are introduced only for the purpose of replaying the derivation, and do never appear in the actual clause set, thus they play no semantic role. The constraint Λ_Δ will contain variables from the free part, as well as parameters p_i, which stand for the constraint variables of the original parameterized clause[1].

Example 1. *Consider the inference*

$$\frac{x = 1 \parallel \rightarrow P(x) \qquad x' = x + 1 \parallel P(x) \rightarrow P(x')}{x = 2 \parallel \rightarrow P(x)}$$

from the introduction. We would now perform the inference

$$\frac{x = p \parallel \rightarrow P(x) \qquad x' = x + 1 \parallel P(x) \rightarrow P(x')}{x = p + 1 \parallel \rightarrow P(x)}$$

to get $x = p + 1$ *as* Λ_Δ.

If we replace the parameters by their corresponding variables, and replace the remaining variables by their primed versions, we obtain $\Lambda_\Delta[x_1, \ldots, x_n, x'_1, \ldots, x'_n]$, which describes a relation[2] $R_\Delta \subseteq \mathbb{R}^{2n}$. We write Λ^k_Δ for the constraint representing R^k_Δ, if it exists. This constraint will in general contain k as an additional integer variable (we chose k to be distinct from all x_i, x'_i). Note that $(\Lambda_0[\mathbf{x}] \wedge \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k])\{k \mapsto 1\}$ is equivalent to $\Lambda_m[\mathbf{x}]$.

Definition 2 (Constraint Induction)**.** *Let N be a clause set containing two clauses $\Lambda_0 \parallel C, \Lambda_m \parallel C$ with identical free part (up to variable renaming) such that $\Lambda_m \parallel C$ was derived from $\Lambda_0 \parallel C$ using clauses $D_1, \ldots, D_m$ in N. Let Λ_Δ be the constraint obtained by replaying the derivation as described above,*

[1] Possibly after simplification and variable elimination to get rid of variables not occurring in C.

[2] Some parameters and variables may not occur in Λ_Δ, we may then just consider them to be unconstrained, i.e., they can take any value in $\mathbb{R}$.

and suppose that Λ^k_Δ exists. The constraint induction rule is the inference rule

$$\frac{\Lambda_0 \parallel C \quad D_1 \ \ldots \ D_m \quad \Lambda_m \parallel C}{\Lambda_0[\mathbf{x}], \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k] \parallel C[\mathbf{x}']}$$

Proposition 3 (Soundness of Constraint Induction)**.** *Let N be a clause set, and assume $\Lambda_0[\mathbf{x}], \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k]$ was derived from $\Lambda_0 \parallel C, D_1, \ldots, D_m, \Lambda_m \parallel C \in N$ by constraint induction. Then $N \models \Lambda_0[\mathbf{x}], \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k]$.*

Proof.

$$C[\mathbf{p}], D_1, \ldots, D_m \models \Lambda_\Delta[\mathbf{p}, \mathbf{x}'] \to C[x'] \tag{3.1}$$

$$\implies \quad D_1, \ldots, D_m \models (C[\mathbf{p}] \wedge \Lambda_\Delta[\mathbf{p}, \mathbf{x}']) \to C[\mathbf{x}'] \tag{3.2}$$

$$\implies \quad D_1, \ldots, D_m \models (C[\mathbf{x}] \wedge \Lambda_\Delta[\mathbf{x}, \mathbf{x}']) \to C[\mathbf{x}'] \tag{3.3}$$

$$\implies \quad D_1, \ldots, D_m \models (C[\mathbf{x}] \wedge \Lambda_\Delta[\mathbf{x}, \mathbf{x}'] \wedge \Lambda_\Delta[\mathbf{x}', \mathbf{x}''] \wedge \cdots \wedge \Lambda_\Delta[\mathbf{x}^{(k-1)}, \mathbf{x}^{(k)}]) \to C[\mathbf{x}^{(k)}] \tag{3.4}$$

$$\implies \quad D_1, \ldots, D_m \models (C[\mathbf{x}] \wedge \exists k(\Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k])) \to C[\mathbf{x}'] \tag{3.5}$$

$$\implies \quad D_1, \ldots, D_m \models (C[\mathbf{x}] \wedge \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k]) \to C[\mathbf{x}'] \tag{3.6}$$

$$\implies \quad N \models (\Lambda[\mathbf{x}] \wedge \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k]) \to C[\mathbf{x}'] \tag{3.7}$$

(3.1) holds by soundness of SUP(LA) and the fact that $\mathbf{x} = \mathbf{p} \parallel C[\mathbf{x}]$ is equivalent to $C[\mathbf{p}]$, (3.2) follows because $C[\mathbf{p}]$ is ground, (3.3) follows because the $\mathbf{p}$ do not occur outside of $C[\mathbf{p}]$ and $\Lambda_\Delta[\mathbf{p}, \mathbf{x}']$, (3.4) follows by induction on k, (3.5) follows by definition of Λ^k_Δ, (3.6) is obtained by turning the existential quantifier on the left-hand side of the implication into an implicit outermost universal quantifier, as k doesn't appear in $C[\mathbf{x}']$, (3.7) follows because $D_1, \ldots, D_m \in N$ and $N \models \Lambda[\mathbf{x}] \to C[\mathbf{x}]$.

□

The constraint induction rule is only applicable if Λ^k_Δ exists and can be effectively computed. We will now look at a class of linear arithmetic constraints for which this is always the case. Given two relations $R_1 \subseteq \mathbb{R}^{2n}$ and $R_2 \subseteq \mathbb{R}^{2m}$, the *product of* R_1, R_2 is the relation $R \subseteq \mathbb{R}^{2(m+n)}$ such that $R(\mathbf{x}, \mathbf{y}, \mathbf{x}', \mathbf{y}')$ if and only if $R_1(\mathbf{x}, \mathbf{x}')$ and $R_2(\mathbf{y}, \mathbf{y}')$, where $\mathbf{x} = x_1, \ldots, x_n$ and $\mathbf{y} = y_1, \ldots, y_m$. If R is the product of R_1, R_2, then $R^k(\mathbf{x}, \mathbf{y}, \mathbf{x}', \mathbf{y}')$ if and only if $R^k_1(\mathbf{x}, \mathbf{x}')$ and $R^k_2(\mathbf{y}, \mathbf{y}')$. Hence we can compute the transitive closure of a product relation if we can compute the transitive closure for each component relation.

Proposition 4. *Let* $R(x_1, \ldots, x_n, x'_1, \ldots, x'_n) \subseteq \mathbb{R}^{2n}$ *be defined by*

$$\bigwedge_{i \in I} x_i + \alpha_{ij} x_j + a_i \ \# \ x'_i \ \wedge \bigwedge_{\alpha_{ij} \neq 0} x'_j = 0$$

for $I \subseteq \{1, \ldots, n\}$, $\alpha_{ij} \in \mathbb{R}$, $\alpha_{ii} = 0$ *for all* $1 \leq i \leq n$, $a_i \in \mathbb{R}$ *and* $\# \in \{<, \leq, \geq, >\}$. *Then* $R^k(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ *holds if and only if*

$$\bigwedge_{i \in I} x_i + \alpha_{i,j} x_j + k a_i \ \# \ x'_i \ \wedge \bigwedge_{\alpha_{ij} \neq 0} x'_j = 0$$

Proof. By induction on k. □

Proposition 5. *Let* $R(x_1, \ldots, x_n, x'_1, \ldots, x'_n) \subseteq \mathbb{R}^{2n}$ *be defined by*

$$\bigwedge_{l=1}^{m} \sum_{j \in J} \beta_{lj} x_j \leq d_l \ \wedge \bigwedge_{j \in J} x'_j = \delta_j x_j + c_j$$

for $J \subseteq \{1, \ldots, n\}$, $m \geq 1$, $\delta_j \in \{0, 1\}$ *and* $c_j, \beta_{lj}, d_l \in \mathbb{R}$.
Then $R^k(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ *holds for* $k \geq 2$ *if and only if*

$$\bigwedge_{l=1}^{m} \left(\sum_{j \in J} \beta_{lj} x_j \leq d_l \ \wedge \ \sum_{j \in J} \beta_{lj} \left(\delta_j \left(x_j + (k-2) c_j \right) + c_j \right) \leq d_l \right)$$
$$\wedge \bigwedge_{j \in J} x'_j = \delta_j \left(x_j + (k-1) c_j \right) + c_j$$

Proof. A straightforward proof using matrix operations can be found in [6]. □

In the following, we will apply the induction rule to constraints that describe products of the kinds of relations described in Propositions 4 and 5. It turns out that this is sufficient to turn SUP(LA) with constraint induction into a decision procedure for timed automata extended with unbounded integer variables, as long as they satisfy certain flatness properties (Section 4) and also speed up proof search, shorten proofs and enable termination of saturation for other kinds of problems (Section 5).

If we don't insist on being able to express the transitive closure as a single conjunction, then it becomes possible to compute the transitive closure of more involved types of constraints [10, 25, 16, 8]. For instance, if the closure can be expressed in Presburger arithmetic, we can derive several clauses that together constitute the inductive invariant (by expressing the closure in disjunctive normal form and introducing one clause per disjunct). For the time being, we restrict ourselves to constraints of the above form. We plan to investigate extensions of the rule in future work.

4 Finite Saturation of Extended Timed Automata

For a set of variables X, the sets $\mathsf{CC}(X)$, $\mathsf{IG}(X)$ and $\mathsf{IA}(X)$ of *clock constraints* and *integer guards*, respectively, are defined as

$$\mathsf{CC}(X) : \mathrm{cc} ::= x \circ c \mid x - y \circ c \mid \mathrm{cc} \wedge \mathrm{cc} \mid \mathsf{true}$$
$$\mathsf{IG}(X) : \mathrm{ig} ::= a_1 x_1 + \cdots + a_n x_n \leq a \mid \mathrm{ig} \wedge \mathrm{ig}$$

where $x \in X$, $c \in \mathbb{N}$, $\circ \in \{<, \leq, =, \geq, >\}$, and $a_i, a \in \mathbb{Z}$. The set $\mathsf{IA}(X)$ of *integer assignments* consists of all substitutions mapping each $x \in X$ to a term of the form a or $x + a$, for $a \in \mathbb{Z}$.

Definition 6 (Extended Timed Automaton)**.** *An* extended timed automaton *is a tuple*

$$\mathcal{T} = (L, l_{init}, X, \mathrm{ig}_{init}, \{\mathrm{inv}_l\}_{l \in L}, E)$$

where L is a finite set of locations *with* initial location $l_{init} \in L$*, X is a finite set of variables partitioned into subsets X_C, X_D of real-valued* clock variables *and integer-valued variables, respectively;* $\mathrm{ig}_{init} \in \mathsf{IG}(X_D)$ *describes the initial values of the integer variables;* $\mathrm{inv}_l \in \mathsf{CC}(X_C)$ *is the* invariant *of location l;* $E \subseteq L \times \mathsf{CC}(X_C) \times \mathsf{IG}(X_D) \times \mathsf{IA}(X_D) \times 2^{X_C} \times L$ *is a finite set of* edges*. An edge $(l, \mathrm{cc}, \mathrm{ig}, \mathrm{ia}, Z, l')$ represents a transition from location l to location l'. The constraints* cc *and* ig *determine when the edge is enabled, and the set Z contains the clocks to be reset to zero when taking the edge, together with the assignment* ia*. If $X = X_C$, $\mathcal{T}$ is a classical timed automaton [2, 15].*

A *state* of an extended timed automaton is a tuple (l, ν) consisting of a location $l \in L$ and a valuation $\nu \in X \to \mathbb{R}$ for all variables. The initial states are of the form $(l_{\mathrm{init}}, \nu_{\mathrm{init}})$ where ν_{init} assigns zero to all clocks and the values of integer variables satisfy $\mathrm{ig}_{\mathrm{init}}$. In any location, the values of all clock variables increase continously at a constant rate. The automaton can stay in a location as long as the clock values satisfy the location's invariant.

When the valuation of a state satisfies the guards cc and ig of an outgoing edge, the corresponding transition can be taken, resetting the clocks in Z and applying the assignment ia. We can thus view a transition as a relation over states, i.e., a set of pairs of states. We say that a state s is *reachable from a state* s_0, if there exists a sequence of states $s_1, \ldots, s_{n-1}$ such that (s_i, s_{i+1}) is contained in some transition, for all $0 \leq i \leq n-1$. In this case we also say that s_0 is *backward-reachable from* s_n. If there exists some initial state s_0 such that s_n is reachable from s_0, then we simply call s_n *reachable*.

Let $\mathcal{T} = (L, l_{\text{init}}, X, \text{ig}_{\text{init}}, \{\text{inv}_l\}_{l \in L}, E)$ be an extended timed automaton. The encoding of reachability for extended timed automata is analogous to that for classical timed automata [15], except that clauses encoding discrete transitions now also include integer guards and assignments. We use a reachability predicate Reach, and constant symbols $l \in L$ for every location[1]. The vector $\mathbf{x}$ contains the clock variables variables X_C, $\mathbf{z}$ contains the integer variables X_D. Furthermore, we fix a bijection $' : X \to X'$ such that $x' \in X'$ for any $x \in X$. The clause

$$\mathbf{x}{=}0,\ \ \text{ig}_{\text{init}}(\mathbf{z}) \parallel\ \rightarrow \text{Reach}(\mathbf{x}, \mathbf{z}, l_{\text{init}}).$$

encodes reachability of the initial states. For every location $l \in L$,

$$t{\geq}0,\ \mathbf{x}'{=}\mathbf{x}{+}t,\ \ \text{inv}_l[\mathbf{x}'] \parallel\ \text{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow \text{Reach}(\mathbf{x}', \mathbf{z}, l).$$

encodes time-reachability for location l. For a variable x and set of variables Z, we define the substitution ρ_Z to be $\rho_Z(x) = 0$ if $x \in Z$, and $\rho_Z(x) = x$ otherwise, and we extend it to vectors of variables pointwise. For every edge $e = (l, \text{cc}, \text{ig}, \text{ia}, Z, l')$ in E, the clause

$$\text{cc}[\mathbf{x}],\ \mathbf{x}'{=}\rho_Z(\mathbf{x}),\ \text{ig}(\mathbf{z}),\ \mathbf{z}'{=}\text{ia}(\mathbf{z}),\ \text{inv}_{l'}[\mathbf{x}'] \parallel\ \text{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow \text{Reach}(\mathbf{x}', \mathbf{z}', l').$$

represents the discrete transition from l to l' via e. We refer to the set containing all of the above clauses as the *reachability theory* of the automaton. A *reachability conjecture* is a clause of the form $\Lambda \parallel\ \text{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow$.

A clause $\Lambda \parallel\ \rightarrow \text{Reach}(\mathbf{x}, \mathbf{z}, l)$ represents the reachability of the set of states $\{(l, \nu) \mid \nu \text{ satisfies } \exists \mathbf{y}.\Lambda\}$, where $\mathbf{y}$ are the variables of Λ different from $\mathbf{x}, \mathbf{z}$: if N is a reachability theory, then $N \models (\Lambda \parallel\ \rightarrow \text{Reach}(\mathbf{x}, \mathbf{z}, l))$ if and only if the above states are reachable. It follows that $N \cup \{\Lambda \parallel\ \text{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow\ \}$ is unsatisfiable if and only if at least one of the states $\{(l, \nu) \mid \nu \text{ satisfies } \exists \mathbf{y}.\Lambda\}$ is reachable.

[1]For readability, we omit the additional terms ensuring maximality of right-hand sides [15]

We can also view a conjecture clause $\Lambda \parallel \text{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow$ as asserting the backward-reachability of a set of states from some given final states. Unsatisfiability of $N \cup \{\Lambda \parallel \text{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow\}$ then means that the intersection of the set of reachable states and the set of states backward-reachable from some final states is non-empty, i.e., there exists reachable final states.

By abuse of terminology, we will say that a clause *represents a set of states* S, whenever it represents either the reachability or the backward-reachability of S, when no confusion arises.

Finally, a transition clause $\Lambda \parallel \text{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow \text{Reach}(\mathbf{x}', \mathbf{z}', l')$ represents a transition T, such that $((l, \nu), (l', \nu')) \in T$ if and only if v, v' satisfies $\exists \mathbf{y}.\Lambda$, where $\mathbf{y}$ are the variables of Λ different from $\mathbf{x}, \mathbf{z}, \mathbf{x}', \mathbf{z}'$. Resolving a positive clause with a transition clause yields another positive clause, which represents the successor states under the given transition of the states represented by the first clause. Similarly, resolving a negative clause (i.e., a reachability conjecture) with a transition clause yields again a negative clause which representes the predecessor states under the given transition of the states represented by the first clause.

In [15], we show how to ensure that the positive literals of such clauses are always strictly maximal in the clause. This guarantees that starting from the encoding of an extended timed automaton and one (or more) reachability conjecture, only negative unit clauses can be derived (that's why we don't need factoring). The inferences correspond to a backward traversal of the automaton's state space, starting from the states represented by the reachability conjecture. This restriction to backward traversal ensures termination of saturation for the encoding of classical timed automata (without integer variables). In the case of extended timed automata, this alone is no longer sufficient, since the assignments to the integer variables cannot be assumed to be monotonic. Thus assignments to integer variables that occur on a cycle may lead to non-termination of saturation, because such a cycle will induce a loop during proof search. This loop however can be handled by the constraint induction rule if the clock constraints and clock resets on such a cycle satisfy certain properties.

Definition 7 (Acceleratable cycle)**.** *Let* $(L, l_{init}, X, \text{ig}_{init}, \{\text{inv}_l\}_{l \in L}, E)$ *be an extended timed automaton. A sequence* $(e_0, \ldots, e_{n-1})$ *of edges* $e_i = (l_i, \text{cc}_i, \text{ig}_i, \text{ia}_i, Z_i, l'_i) \in E$ *is called a* cycle *if* $l'_i = l_{i+1 \bmod n}$ *for all* $0 \leq i < n$*. It is called a* simple cycle*, if additionally* $l_i \neq l_j$ *for all* $i \neq j$*. Following [18], a simple cycle is called* acceleratable*, if all invariants and guards on the cycle contain at most a single clock variable, called the* clock of the cycle*, which is the same for all invariants and guards on the cycle, and this clock is reset on all incoming edges to* l_0*. The location* l_0 *is called the* reset location*. By*

acceleratable cycle*, we mean an acceleratable simple cycle. By an* integer cycle*, we mean a cycle where at least one edge contains an assignment to integer variables.*

In [18] it is shown that for any acceleratable cycle, there exists an interval $[a, b]$ of clock values, called the *window* of the cycle, such that $[a, b]$ contains exactly all the possible execution times of the cycle, independently of any path prefix. It follows that any $k \geq 1$ consecutive executions of the cycle take time in $[ka, kb]$. The idea is that we can decompose the cycle into maximal segments of the form $(e_s, e_{s+1}, \ldots, e_{s'})$ $(s \geq s')$, where the clock of the cycle, say y, is reset on edge $e_{s'}$ but on none of the edges $e_s, \ldots, e_{s'-1}$. There is at least one such segment, since y is reset on edge e_{n-1} (see Figure 4.1 for an illustration). Now assume without loss of generality that each

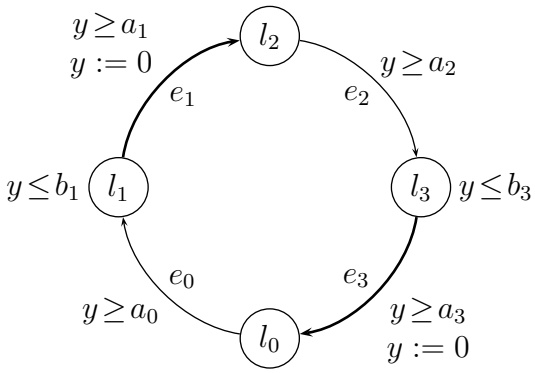

Figure 4.1: An acceleratable cycle: l_0 is the reset location, there are two segments (e_0, e_1) and (e_2, e_3).

edge e_i has a guard of the form $y \geq a_i$, and that each location l_i has an invariant of the form $y \leq b_i$ $(a_i, b_i \geq 0$, and possibly $b_i = \infty$ to represent the invariant $\mathsf{true})$. It is not hard to prove that traversing a segment $e_s, e_{s+1}, \ldots, e_{s'}$ must take between $\max\{a_s, a_{s+1}, \ldots, a_{s'}\}$ and $b_{s'}$ time units. Writing $s(i), s'(i)$ for the first and last indices of the ith segment, respectively, one can then show that traversing the whole cycle must take time in $[a, b]$ for $a = \sum_i \max\{a_s(i), a_{s(i)+1}, \ldots, a_{s'(i)}\}$ and $b = \sum b_{s'(i)}$ (more details can be found in [18]).

Proposition 8. *Let N be the reachability theory of an extended timed automaton with an acceleratable cycle $(e_0, \ldots, e_{n-1})$ with clock y and reset location l_0, and let $C_0 = \Lambda_0 \parallel \mathrm{Reach}(\mathbf{x}, y, \mathbf{z}, l_0) \rightarrow$ be a reachability conjecture. Then we can derive an invariant clause from $N \cup \{C_0\}$ representing all l_0-states backward-reachable from C_0 by transitions $e_0, \ldots, e_{n-1}$.*

Proof. Let use write Disc(i) for the discrete transition clause of e_i, and Time(i) for the time transition clause of l_i. For a clause C, let $\mathrm{pre}(C, i)$ denote the result of resolving C with Disc(i) and then resolving the result with Time(i), and let $\mathrm{pre}(C, i_1, \ldots, i_k)$ abbreviate $\mathrm{pre}(\ldots \mathrm{pre}(C, i_1), \ldots, i_k)$. Starting with C_0, after $2n$ resolution steps, we obtain the clause $C_{2n} = \mathrm{pre}(C_0, n-1, \ldots, 0)$ which has the same free part as C_0, so we may attempt to apply the constraint induction rule (see Figure 4.2). Let C_0^p be the

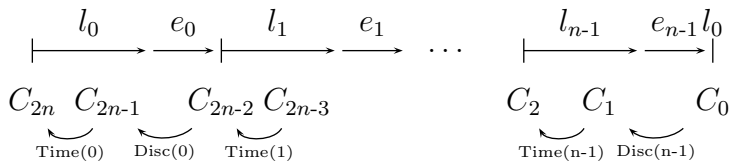

Figure 4.2: Backward traversal of a cycle $(e_0, \ldots, e_{n-1})$.

parameterized version of C_0, i.e.,

$$\mathbf{x} = \mathbf{p}, y = p_y, \mathbf{z} = \mathbf{q} \parallel \mathrm{Reach}(\mathbf{x}, y, \mathbf{z}, l_0) \rightarrow .$$

The constraint Λ_Δ, on which the transitive closure computation is based, is obtained as the constraint of the clause $\mathrm{pre}(C_0^p, 0, \ldots, n-1)$. Observe that none of the transition clauses Disc(i), Time(i), nor C_0^p has a constraint containing (in-)equations between variables from X_C and X_D. This property thus also holds for all clauses derived during the replaying of the derivation, and Λ_Δ is therefore of the form $\Lambda_\Delta^C, \Lambda_\Delta^D$, where Λ_Δ^D contains only variables from X_D and Λ_Δ^C contains none of them. Therefore also Λ_Δ^k can be written as $(\Lambda_\Delta^C)^k, (\Lambda_\Delta^D)^k$.

Let us first focus on Λ_Δ^C. Let $(e_s, e_{s+1}, \ldots, e_{s'})$ be an arbitrary segment of the cycle, and let $s \leq i \leq s'$. The clause Time(i) is of the form

$$t_i \geq 0,\ \mathbf{x}' = \mathbf{x} + t_i,\ y' = y + t_i,\ y \leq b_i \parallel \mathrm{Reach}(\mathbf{x}, y, \mathbf{z}, l_i) \rightarrow \mathrm{Reach}(\mathbf{x}', y', \mathbf{z}, l_i)$$

and clause Disc(i) is of the form

$$a_i \leq y \leq b_{i+_n 1}, y' = 0, \mathrm{ig}_i, \mathbf{z}' = \mathrm{ia}_i(\mathbf{z}) \parallel \mathrm{Reach}(\mathbf{x}, y, \mathbf{z}, l_i) \rightarrow \mathrm{Reach}(\mathbf{x}, y', \mathbf{z}', l_{i+_n 1})$$

if $i = s$, and

$$a_i \leq y \leq b_{i+_n 1}, \mathrm{ig}_i, \mathbf{z}' = \mathrm{ia}_i(\mathbf{z}) \parallel \mathrm{Reach}(\mathbf{x}, y, \mathbf{z}, l_i) \rightarrow \mathrm{Reach}(\mathbf{x}, y, \mathbf{z}', l_{i+_n 1})$$

otherwise[2]. Let C be an arbitrary reachability conjecture referring to the location $l_{s'}$, and with a constraint Λ not containing any (in-)equations mixing

[2] $a +_n b$ stands for $(a + b) \mod n$.

variables from X_C and X_D. We can thus write Λ as Λ^C, Λ^D. It is easy to verify that the clause $\mathrm{pre}(C, s', \ldots, s)$ has a constraint of form

$$\begin{array}{lll} t_s, t_{s+1}, \ldots, t_{s'} \geq 0, & a_{s'} \leq y + t_s + \cdots + t_{s'} \leq b_{s'}, & \Lambda^C[\mathbf{x} + t_s + \cdots + t_{s'}, 0] \\ & a_{s'-1} \leq y + t_s + \cdots + t_{s'-1} \leq b_{s'}, & \\ & \vdots & \\ & a_s \leq y + t_s \leq b_{s'} & \end{array}$$

or, replacing $t_s + \cdots + t_{s'}$ by t,

$$t \geq 0, \max\{a_s, \ldots, a_{s'}\} \leq y + t \leq b_{s'}, \Lambda^C[\mathbf{x} + t, 0].$$

Now assume that the cycle consists of $k \geq 1$ segments indexed by $s(1)$, $s'(1), \ldots, s(k), s'(k)$, respectively, with $s(1) = 0$, $s'(k) = n-1$. The clause resulting from the replay of the dervation is $\mathrm{pre}(C_0^p, 0, \ldots, n-1) = \mathrm{pre}(C_0^p, s(1), \ldots, s'(k))$ and has constraint Λ_Δ. It follows by induction on k and the previous observation that Λ_Δ^C is of the form

$$\begin{array}{rl} t_1, \ldots, t_k \geq 0, & \max\{a_{s(1)}, \ldots, a_{s'(1)}\} \leq y + t_1 \leq b_{s'(1)}, \mathbf{p} = x + t_1 + \ldots + t_k, p_y = 0 \\ & \max\{a_{s(2)}, \ldots, a_{s'(2)}\} \leq t_2 \leq b_{s'(2)}, \\ & \max\{a_{s(3)}, \ldots, a_{s'(3)}\} \leq t_2 \leq b_{s'(3)}, \\ & \vdots \end{array}$$

or, replacing $t_1 + \cdots + t_k$ by t,

$$t \geq 0, a \leq y + t \leq b, \mathbf{p} = \mathbf{x} + t, p_y = 0$$

where $a = \sum_i \max\{a_{s(i)}, \ldots, a_{s'(i)}\}$ and $b = \sum_i b_{s'(i)}$. Eliminating t, we get

$$a \leq \mathbf{p} - \mathbf{x} + y \leq b, \mathbf{p} \geq \mathbf{x}, p_y = 0.$$

Replacing variables by their primed versions, and parameters by their corresponding variables, we get

$$a \leq \mathbf{x} - \mathbf{x}' + y' \leq b, \mathbf{x} \geq \mathbf{x}', y = 0.$$

By Proposition 4, the transitive closure $(\Lambda_\Delta^C)^k$ is

$$ka \leq \mathbf{x} - \mathbf{x}' + y' \leq kb, \mathbf{x} \geq \mathbf{x}', y = 0.$$

Now we consider Λ_Δ^D. Let $\mathrm{ia}_i, \mathrm{ia}_i$ be the integer guard and assignment of edge e_i, respectively. Remember that ia_i is a substitution with domain X_D. We can prove by induction over the derivation that Λ_Δ^D is of the form

$$\begin{array}{l} \mathrm{ig}_0[\mathbf{z}],\ (\mathrm{ig}_1[\mathbf{z}])\,\mathrm{ia}_0,\ (\mathrm{ig}_2[\mathbf{z}])(\mathrm{ia}_1\,\mathrm{ia}_0),\ \ldots,\ (\mathrm{ig}_{n-1}[\mathbf{z}])(\mathrm{ia}_{n-2}\ldots\mathrm{ia}_0), \\ \mathbf{q} = (\mathrm{ia}_{n-1}\ldots\mathrm{ia}_0)(\mathbf{z}). \end{array}$$

This constraint can be viewed as a *normalized instruction* [6], which has the form required by Proposition 5, and the transitive closure $(\Lambda_\Delta^D)^k$ represents the X_D-valuations corresponding to states reachable by k-fold application of the instruction. We can thus apply the constraint induction rule and obtain the invariant clause

$$\Lambda_0[\mathbf{x}, y, \mathbf{z}], (\Lambda_\Delta^C)^k[\mathbf{x}, y, \mathbf{x}', y', k], (\Lambda_\Delta^D)^k[\mathbf{z}, \mathbf{z}', k] \parallel \mathrm{Reach}(\mathbf{x}', y', \mathbf{z}', l_0) \rightarrow$$

representing all states backward-reachable from C_0 by the cycle $(e_0, \ldots, e_{n-1})$. □

Theorem 9. *Let $\mathcal{T}$ be an extended timed automaton such that any integer cycle is acceleratable, and any location belongs to at most one integer cycle. Let N be a clause set containing the encoding of $\mathcal{T}$ and a reachability conjecture. Then N can be finitely saturated by SUP(LA) with constraint induction.*

Proof. Consider a fair derivation $N = N_0, N_1, N_2, \ldots$ from N where $N_{i+1} = N_i \cup \{C_i\}$ and C_i is the non-redundant result of an inference from clauses from N_i, and no clause in N_i subsumes C_i. Assume for contradiction that the derivation is infinite. Since there are only finitely many locations, there must be infinitely many clauses in the derivation referring to the same location, say l, (those are clauses of the form $\Lambda \parallel \mathrm{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow$) and hence l must lie on a cycle. If no path from l back to itself involves any integer operations, then l can only repeat finitely often (see [15] for details). Hence l must lie on an integer cycle, which by assumption is unique and acceleratable, and at least one of its locations is a reset location, say l_r. Furthermore, l_r must also repeat infinitely often, hence there is an infinite sequence $C_{i_1}, C_{i_2}, \ldots$ of clauses referring to l_r. Since the derivation is fair, we eventually apply the constraint induction rule to two successive such clauses, say C_{i_j} and $C_{i_{j+1}}$. Assume the rule is a applied at step j of the derivation i.e., the resulting invariant clause is C_j. Writing $\Lambda_{i_j}, \Lambda_{i_{j+1}}$ for the constraints of clause $C_{i_j}, C_{i_{j+1}}$, respectively, the invariant clause has the form $\Lambda_{i_j}[\mathbf{x}], \Lambda_\Delta^k[\mathbf{x}, \mathbf{x}', k] \parallel \mathrm{Reach}(\mathbf{x}', l_r) \rightarrow$. This clause cannot eliminated by forward subsumption, for otherwise there would have to be a clause $\Lambda' \parallel \mathrm{Reach}(\mathbf{x}, l_r) \rightarrow$ in N_j such that

$$\forall \mathbf{x}, \mathbf{x}', k.\, \big(\Lambda_{i_j}[\mathbf{x}], \Lambda_\Delta^k[\mathbf{x}, \mathbf{x}', k] \rightarrow \exists \mathbf{y}.\Lambda'[\mathbf{x}', \mathbf{y}]\big)$$

would have to hold, where $\mathbf{y}$ are the variables of Λ' different from $\mathbf{x}, \mathbf{x}', k$. But then the last premise of the constraint induction rule would also be subsumed, because $\Lambda_{i_{j+1}}$ is equivalent to $\big(\Lambda_{i_j}[\mathbf{x}], \Lambda_\Delta^k[\mathbf{x}, \mathbf{x}', k]\big)\,\{k \mapsto 1\}$, and so the rule could not have been applied in the first place. It follows that

the invariant clause is contained in N_{j+1} and all subsequent clause sets, since backward subsumption has to be strict. The invariant clause can be resolved with the clauses corresponding to the edges in the cycle, yielding clauses of the form $\Lambda[\mathbf{x}, \mathbf{x}', k] \parallel \mathrm{Reach}(\mathbf{x}', l) \rightarrow$ for every location l on the cycle. Any further traversal of the cycle then yields clauses of the form $\Lambda[\mathbf{x}, \mathbf{x}', k+1] \parallel \mathrm{Reach}(\mathbf{x}', l) \rightarrow$, which are subsumed, as

$$\forall \mathbf{x}, \mathbf{x}', k.\, (\Lambda[\mathbf{x}, \mathbf{x}', k+1] \rightarrow \exists k'.\Lambda[\mathbf{x}, \mathbf{x}', k'])$$

holds. Finally, all clauses $C_{i_{j+m}}$, $m > 0$, are instances of C_j (via instantiation of k), and hence eliminated by forward subsumption, so the sequence $C_{i_1}, C_{i_2}, \ldots$ cannot be infinite, a contradiction.

□

Since the encoding of extended timed automata does not introduce any function symbols ranging into the arithmetic sorts, it is sufficiently complete, and SUP(LA) is therefore refutationally complete for such encodings. Together with Theorem 9, this implies that SUP(LA) is a decision procedure for the reachability problem in extended timed automata.

5 Implementation and Results

We have implemented the constraint induction rule in our SPASS(LA) theorem prover [1].

Premise selection. In the current implementation, whenever a clause new C with a non-empty constraint has been derived by resolution, the clause store is searched for potential partner clause for constraint induction. First, the term index is queried to find all clauses with same free part as C. Then the tree of parent clauses of C is recursively traversed to check whether one of the retrieved clauses is an ancestor of C. This traversal is stopped as soon as one of the potential partner clauses has been reached – in which case the constraint induction rule is applied –, or when the minimum of the derivation depths of all potential partner clauses has been reached.

Handling of mixed integer constraints. SPASS(LA) currently uses Z3 [11] as a back end for constraint solving, both for satisfiability and implication checking. Although Z3 supports mixed real/integer constraints, it turned out that when checking implication between two constraints both containing integer variables (as they arise in our approach), Z3 almost always returned "unknown". Since the implication check is needed for subsumption and hence is ultimately the key to termination, we decided to implement our own implication test for mixed constraints. The test consists of a preprocessing step, which tries to eliminate all conjuncts containing integer variables from the right-hand side of the implication, followed by a call to Z3 with the resulting implication problem. The preprocessing works as follows: suppose we are trying to prove the implication $\forall \mathbf{x}.\Lambda_2 \Rightarrow \exists \mathbf{y}.\Lambda_1$, where Λ_1, Λ_2 are constraints, $\mathbf{x}$ are the variables of Λ_2 and $\mathbf{y}$ are the variables of Λ_1 not occurring in Λ_2. Suppose there are atomic constraints $\phi_1 \in \Lambda_1$, $\phi_2 \in \Lambda_2$ such that $\phi_1 = x - \sum_{i=1}^{n} \alpha_i k_i \ \# \ c$ and $\phi_2 = x - \sum_{J \subseteq \{1,\ldots,n\}} \sum_{j \in J} \alpha_j k'_j \ \# \ c + d$, where $\#$ is one of $<, \leq, =, \geq$ or $>$, x is a real (or integer) variable, k_i, k'_j are integer

variables and $c, d \in \mathbb{R}$. If $d = \sum_{\substack{l \in L \\ L \subseteq \{1,\ldots,n\}}} m_l \alpha_{i_l}$ (where m_l are integer constants ≥ 1) such that L contains at least the indices missing from J, i.e., $(\{1,\ldots,n\} \setminus J) \subseteq L$, then ϕ_2 implies $\exists (k'_j)_{j \in J}.\phi_1$: assign m_l to k'_l, and either k_j or $k_j + m_j$ to the other k'_j. In this case, we can remove ϕ_1 from Λ_1. In the implementation, we currently only consider the case where $L = \{i\}$ for some $i \in \{1,\ldots,n\}$, and either $J = \{1,\ldots,n\}$ or $J = \{1,\ldots,n\} \setminus L$, which is enough to handle all implication problems arising in our examples. Nevertheless, we are investigating the use of other solvers that implement complete quantifier elimination for mixed constraints.

Example 10 (Extended timed automaton)**.** *Consider the extended timed automaton in Figure 5.1, where x_1, x_2 are clocks and z_1, z_2 are integer variables. We want to check whether location L_2 is reachable with a valuation such that $z_1 \geq z_2$ and $x_2 < 12$. Since x_2 is never reset to zero, its value represents the total time elapsed since first entering L_1. As the cycle at L_1 must be traversed four times before z_1 has overtaken z_2, and each cycle traversal takes at least three time units, such a state is not reachable. This problem can be encoded*

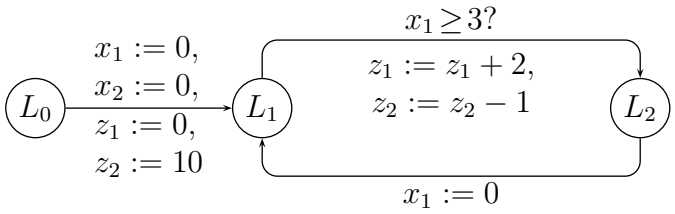

Figure 5.1: An extended timed automaton.

by the following clause set, where the last clause is the negated conjecture:[1]

$$
\begin{array}{rcl}
x_1{=}0,\ x_2{=}0,\ z_1{=}0,\ z_2{=}0 & \| & \rightarrow L_1(x_1, x_2, z_1, z_2) \\
t{\geq}0,\ x'_1{=}x_1{+}t,\ x'_2{=}x_2{+}t & \| & L_1(x_1, x_2, z_1, z_2) \rightarrow L_1(x'_1, x'_2, z_1, z_2) \\
z'_1{=}z'_1{+}2,\ z'_2{=}z'_2{-}1 & \| & L_1(x_1, x_2, z_1, z_2) \rightarrow L_2(x_1, x_2, z'_1, z'_2) \\
t{\geq}0,\ x'_1{=}x_1{+}t,\ x'_2{=}x_2{+}t & \| & L_2(x_1, x_2, z_1, z_2) \rightarrow L_2(x'_1, x'_2, z_1, z_2) \\
x'_1{=}0 & \| & L_2(x_1, x_2, z_1, z_2) \rightarrow L_1(x'_1, x_2, z_1, z_2) \\
z_1{\geq}z_2,\ x_2{<}12 & \| & L_2(x_1, x_2, z_1, z_2) \rightarrow
\end{array}
$$

The clause set is satisfiable, and without the constraint induction rule, SPASS(LA) does not terminate. With constraint induction activated, the invariant clause

$$k{\geq}1,\ x_1{=}0,\ x_2{\geq}3k,\ z_1{=}2k,\ z_2{=}10{-}k \ \| \ \rightarrow L_1(x_1, x_2, z_1, z_2)$$

[1]For simplicity, we use $L_i(\ldots)$ instead of $\text{Reach}(\ldots, L_i)$, and we also omit L_0.

is derived as soon as the cycle has been traversed once, and is used to subsume all other L_1-clauses. SPASS(LA) terminates with the answer "completion found"[2] *after deriving 23 clauses.*

The next example shows that the induction rule is also useful for speeding up proof search and finding shorter proofs in the case of unsatisfiable clause sets.

Example 11 (Water tank controller)**.** *Figure 5.2 depicts a water tank controller [1] monitoring the water level x in a water tank, into which water is flowing with a constant rate c_{in}. Whenever the water level is greater than 200, the controller opens a valve through which water leaves the tank at a constant rate of c_{out}. We may want to prove that, starting from an empty*

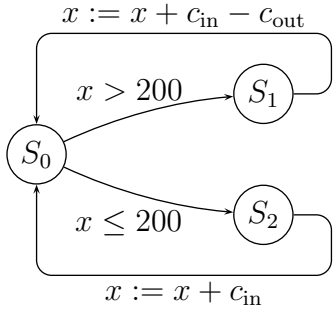

Figure 5.2: Water tank controller

tank, the water level can reach $200 + c_{in}$. This problem can be encoded by the following clause set:

$$\begin{array}{rcl} x > 200 & \| & S_0(x) \to S_1(x) \\ x \leq 200 & \| & S_0(x) \to S_2(x) \\ x' = x + c_{in} - c_{out} & \| & S_1(x) \to S_0(x') \\ x' = x + c_{in} & \| & S_2(x) \to S_0(x') \\ x = 0 & \| & \to S_0(x) \\ x \geq 201 & \| & S_0(x) \to \end{array}$$

For $c_{in} = 1$ and $c_{out} = 2$[3]*, SPASS(LA) without constraint induction needs to derive 1212 clauses before finding a proof of length 211. The proof consists of repeated traversals of the $S_0 \to S_1 \to S_0$ cycle with increasing values of x, until $x = 201$ is reached.*

[2] A completion is a satisfiable saturation of the initial clause set.

[3] In principle, c_{in} and c_{out} don't need to be instantiated, since the invariant computation does not care about the values of constants, but our implementation does not yet handle constant symbols in constraints.

With constraint induction activated, as soon as the clause $x = 1 \| \rightarrow S_0(x)$ *has been derived from the initial clause* $x = 0 \| \rightarrow S_0(x)$ *(using the second and fourth clause) SPASS(LA) detects the cycle and derives the invariant clause*

$$1 \leq k \leq 201,\ x = k \ \| \ \rightarrow S_0(x).$$

which is resolved with the negated conjecture, yielding the empty clause. The proof has length 9 and SPASS(LA) finds it after deriving 13 clauses in total.

If we replace the last clause with $x > 201 \ \| \ S_0(x) \rightarrow$*, the clause set becomes satisfiable. Without constraint induction, SPASS(LA) now derives 1214 clauses before answering "completion found", whereas with constraint induction, only 23 clauses need to be derived (among them the above invariant clause).*

Table 5.1 shows the results from the above examples, together with the total time spent on the problem.

		SUP(LA)		SUP(LA)+ind	
Problem		clauses derived	time	clauses derived	time
Extended TA	sat	–	–	23	0.25s
Water tank	unsat	1212	33s	13	0.15s
Water tank	sat	1214	33s	23	0.18s

Table 5.1: Summary of experimental results

6 Conclusion

We have presented the constraint induction rule that automatically generates inductive invariants during proof search in the context of superposition modulo linear arithmetic. The rule applies to loops in which repeated applications of the same sequence of inferences yield clauses which differ only in their arithmetic constraints (their free parts being identical up to renaming of universally quantified variables). The derived invariant summarizes these clauses by representing the transitive closure of the transformation relating the clauses in the loop. The loop can thus be avoided, by using the invariant clause to subsumes its instances, provided that the invariant clause is smaller in the clause ordering (which is required to maintain completeness of the calculus). In order to find a well-founded ordering for which this is the case, one has to ensure that the constraint induction rule is only applied a finite number of times.

As evidenced by our implementation, the constraint induction rule can considerably speed up proof search, enabling termination of saturation in cases where it would otherwise diverge, and allowing shorter proofs to be found. Since the induction rule applies to clauses with the same free part and invariants thus only talk about the arithmetic constraints, their computation does not require proof generalization and schematization techniques that are necessary to compute invariants for the full first-order setting [22]. Nevertheless, the induction rule significantly increases the power of the SUP(LA) calculus, making it possible to turn it into a decision procedure for reachability in timed automata extended with unbounded integer variables. The decidability of the reachability problem for extended timed automata is not a new result in itself, as it can be obtained from results on counter automata [10, 9]. However, we are able to obtain the result using a general-purpose approach like superposition (which applies to full first-order logic), extended with an induction rule that is also applicable outside the specific automata setting.

Preliminary testing of our implementation shows that the rule enables

termination of saturation and the finding of short proofs for practically interesting problems. We are currently evaluating the use of the rule for problems from program and protocol verification (particularly in the setting of first-order probabilistic timed automata [13]) and ontology reasoning. Finally, we are working on extending the rule to handle wider classes of constraints.

Bibliography

[1] E. Althaus, E. Kruglov, and C. Weidenbach. Superposition modulo linear arithmetic SUP(LA). In S. Ghilardi and R. Sebastiani, editors, *7th international Symposium on Frontiers of Combining Systems*, volume 5749 of *Lecture Notes in Artificial Intelligence*, pages 84–99, Trento, Italy, September 2009. Springer.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[3] L. Bachmair, H. Ganzinger, and U. Waldmann. Superposition with simplification as a decision procedure for the monadic class with equality. In G. Gottlob, A. Leitsch, and D. Mundici, editors, *Computational Logic and Proof Theory, Third Kurt Gödel Colloquium*, volume 713 of *LNCS*, pages 83–96. Springer, August 1993.

[4] L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierarchic first-order theories. *Applicable Algebra in Engineering, Communication and Computing, AAECC*, 5(3/4):193–212, 1994.

[5] L. Bachmair, H. Ganzinger, and U. Waldmann. Refutational theorem proving for hierarchic first-order theories. *Appl. Algebra Eng. Commun. Comput.*, 5:193–212, 1994.

[6] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *CAV*, pages 55–67, 1994.

[7] P. Bouyer, F. Laroussinie, and P.-A. Reynier. Diagonal constraints in timed automata: Forward analysis of timed systems. In *FORMATS*, pages 112–126, 2005.

[8] M. Bozga, R. Iosif, and F. Konecný. Fast acceleration of ultimately periodic relations. In *CAV*, pages 227–242, 2010.

[9] M. Bozga, R. Iosif, and Y. Lakhnech. Flat parametric counter automata. *Fundam. Inform.*, 91(2):275–303, 2009.

[10] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *CAV*, pages 268–279, 1998.

[11] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[12] C. G. Fermüller, A. Leitsch, U. Hustadt, and T. Tamet. Resolution decision procedures. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 25, pages 1791–1849. Elsevier, 2001.

[13] A. Fietzke, H. Hermanns, and C. Weidenbach. Superposition-based analysis of first-order probabilistic timed automata. In C. Fermueller and A. Voronkov, editors, *LPAR 2010*, volume 6397 of *LNCS*, pages 302–316. Springer, 2010.

[14] A. Fietzke, E. Kruglov, and C. Weidenbach. Automatic generation of inductive invariants by SUP(LA). Technical Report MPI-I-2012-RG1-002, Max-Planck-Institut fr Informatik, 2012.

[15] A. Fietzke and C. Weidenbach. Superposition as a decision procedure for timed automata. In *MACIS*, pages 52–62, 2011.

[16] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FSTTCS*, pages 145–156, 2002.

[17] J. Y. Halpern. Presburger arithmetic with unary predicates is Π^1_1 complete. *Journal of Symbolic Logic*, 56(2):637–642, 1991.

[18] M. Hendriks and K. G. Larsen. Exact acceleration of real-time model checking. *Electr. Notes Theor. Comput. Sci.*, 65(6), 2002.

[19] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Tree automata with equality constraints modulo equational theories. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2006.

[20] K. Korovin and A. Voronkov. Integrating linear arithmetic into superposition calculus. In J. Duparc and T. A. Henzinger, editors, *Computer Science Logic*, volume 4646 of *LNCS*, pages 223–237. Springer, 2007.

[21] E. Kruglov and C. Weidenbach. SUP(T) decides the first-order logic fragment over ground theories. In *MACIS*, pages 126–148, 2011.

[22] N. Peltier. A general method for using schematizations in automated deduction. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Computer Science*, pages 578–592. Springer, 2001.

[23] M. Suda, C. Weidenbach, and P. Wischnewski. On the saturation of YAGO. In *IJCAR*, pages 441–456, 2010.

[24] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 27, pages 1965–2012. Elsevier, 2001.

[25] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *CAV*, pages 88–97, 1998.

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available via WWW using the URL `http://www.mpi-inf.mpg.de`. If you have any questions concerning WWW access, please contact `reports@mpi-inf.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Campus E2 3
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-inf.mpg.de`
