Set-based Analysis of Reactive Infinite-state Systems

Witold Charatonik* Andreas Podelski

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

Abstract. We present an automated abstract verification method for infinite-state systems specified by logic programs (which are a uniform and intermediate layer to which diverse formalisms such as transition systems, pushdown processes and while programs can be mapped). We establish connections between: logic program semantics and CTL properties, set-based program analysis and pushdown processes, and also between model checking and constraint solving, viz. theorem proving. We show that set-based analysis can be used to compute supersets of the values of program variables in the states that satisfy a given CTL property.

1 Introduction

Testing runtime properties of systems with infinite state spaces is generally undecidable. Therefore, the best one can hope for are semi-algorithms implementing a test, or always terminating algorithms implementing a *semi-test* (which either yields yes/don't know answers or, dually, no/don't know answers). Based on the idea that any automated method that sometimes detects programming errors is useful, we investigate semi-tests in this paper.

One way to obtain a semi-test is to apply a test to a finite approximation of the infinite system of interest. An essential part of an automated semi-test computes the approximation from a finite representation of the original system, viz. a program. We will study representations of infinite-state systems by *logic programs*. Logic programs are a uniform and intermediate layer to which diverse formalisms such as finite-state transition systems, pushdown processes and while programs can be mapped. The connection between transition systems and logic via logic programs allows us to establish the correspondence between:

- program semantics and temporal logic properties,
- abstraction and logical implication,
- the *Cartesian* abstraction of set-based analysis and pushdown processes,
- model-checking and first-order, resolution-based theorem proving.

Specifically, we consider the temporal logic CTL [14] (which allows one to express safety, inevitability and other important behavioral properties excluding fairness

* On leave from Wrocław University. Partially supported by Polish KBN grant 8T11C02913.

conditions). For a (possibly infinite-state) transition system represented by a logic program, the set of states satisfying a CTL property can be characterized through the semantics of logic programs; see Theorem 4, Section 5.

Now, static program analysis based on abstract interpretation (see, e.g., [10]) may be used to compute a conservative approximation of a CTL property by computing an abstraction of the logic program semantics. The soundness of the abstract-verification method thus obtained holds by the soundness of the abstraction. This is in contrast with the work in, e.g., [22,11,23], where the test of a CTL property is applied to an abstraction of the original system.

We use one particular form of static analysis called *set-based analysis*. Here, the abstraction consists of mapping a set of tuples to the smallest Cartesian product of sets containing it (e.g., $\{\langle a,1\rangle, \langle b,2\rangle\} \mapsto \{a,b\} \times \{1,2\}$). The abstract semantics computed by this analysis defines a Cartesian product of sets; each set describes runtime values of a variable at a program point. This set is sometimes called the *type* of the program variable. Now, if the concrete program semantics is used to characterize the set of *correct* input states, the type of an input variable denotes a conservative approximation of the set of all its values in correct input states (where 'correct' refers to states for which a given CTL property is satisfied); see Theorem 5, Section 6.

Logically, the set-based abstraction amounts to replacing a formula, say, $\varphi[x,y]$ with the free variables x and y, by the conjunction $(\exists y\, \varphi)[x] \wedge (\exists x\, \varphi)[y]$ (which is logically implied by φ). Applying this replacement systematically to a program $\mathcal{P}$ yields a new program $\mathcal{P}^\sharp$. This program defines the degree of abstraction of CTL properties in set-based analysis: the full test for the system defined by $\mathcal{P}^\sharp$ is the semi-test for the system defined by $\mathcal{P}$.

The system obtained by the set-based abstraction of a program $\mathcal{P}$ (defined by the program $\mathcal{P}^\sharp$) is not finite-state. Instead, it is a kind of *pushdown process*. Pushdown processes have raised interest as a class of infinite-state systems for which temporal properties are decidable. The systems considered here extend this class by adding parallel composition, tree-like stacks and non-deterministic guesses of stack contents. The latter extension introduces a non-determinism of infinite branching degree. Since set-based analysis here adds no extra approximation, it yields a full test of CTL properties of pushdown processes even with this extension; see Theorem 2, Section 4.

When we use set-based analysis as a verification method, the constraint-solving algorithms which form its computational heart (e.g., [17,16,8,12]) replace the traditional fixpoint iteration of model checking. The constraints used here can be represented by logic programs (see Section 6). Then, constraint-solving (more precisely, testing emptiness of the solution of interest) amounts to first-order theorem proving based on resolution. We are currently working on making the algorithm [7] for computing the greatest solution practical. One algorithm for computing the least solution is already implemented in the saturation-based theorem prover SPASS [29]; due to specific theorem-proving techniques like powerful redundancy criteria, one obtains an efficient decision procedure for the emptiness test, viz. model checking.

Related work. In [25] we present a direct application of the set-based analysis of logic programs to error diagnosis in concurrent constraint programs. The error can be defined as a special case of a CTL property for a transition system that consists of *non-ground* derivations of logic programs.

Our direct inspiration for investigating transition systems specified by *ground* derivations of logic programs was the work on pushdown processes in [2,5,28]. Here, we extend the result in [2] about CTL model-checking in DEXPTIME to a more general notion of pushdown processes.

Historically, our work started with the *abstract debugging* scheme of [4]. The invariant and intermittent assertions used there correspond to two special cases of CTL properties. Here, we consider trees instead of numbers for the data domain, an abstract domain of regular sets of trees instead of intervals, and Cartesian instead of convex-hull approximation. Our characterization of CTL properties can be extended to while programs over numeric data by using constraint logic programs (over numbers instead of trees) as an intermediate layer.

In [26], Ramakrishna et al. present an implementation of a model checker for the verification of finite-state systems specified by DATALOG programs (*i.e.*, logic programs without function symbols). The correctness of their implementation (in a logic programming language with tabling called XSB) relies implicitly on the characterization of CTL properties that we formally prove for logic programs with function symbols. In contrast to the work in [26] which applies programming techniques that that are proper to logic programming languages, we view logic programs rather as an automata-theoretic formalism.

Structure of the paper. The first three sections are to give a flavor of our method, which we present in technical terms in the subsequent three sections. Section 2 explains our view of logic programs as an intermediate layer for while programs. For every while program with data structures modeled as trees (e.g., lists), we can find a logic program that represents the same transition system. The purpose of Section 3 is to give the intuition of our characterization of CTL properties (also to readers who are not so familiar with logic programs, to which we do not refer in this section). We show how one can translate a finite transition system to a simple logical formula; the formula belongs to a logic program whose operational semantics is that transition system. We then show how the logical formula must be modified so that a particular solution characterizes a given CTL property; this is the formula that belongs to a *program with oracles*. Section 4 explains our view of logic programs as automata at hand of pushdown processes. Section 5 formally introduces the concepts that we used informally in the previous three sections, and it presents the characterization of CTL properties for monolithic transition systems. Section 6 gives a self-contained account of set-based analysis and presents the results about the conservative approximation of CTL properties that lead to an abstract-verification method. Section 7 gives an extension of these results to multi-processor transition systems defined by logic programs with conjunction (which corresponds to parallel composition); *Basic Parallel Processes* (see, e.g., [15]) are here a special case. Finally, in conclusion, we mention possible directions for future work.

2 While programs

We consider an imperative programming language with the two data constructors *cons* and *nil* (integers etc. play the role of constant data constructors). For convenience, $cons(x, y)$ is written as $[x|y]$, $cons(x_1, cons(\ldots, x_n, nil)\ldots)$ as $[x_1, \ldots, x_n]$ and *nil* as $[\,]$. We also have the data destructors *hd* and *tl*, where $hd([x|y]) = x$ and $tl([x|y]) = y$. We will neither formally define the language nor present the translation of its programs to monolithic total logic programs. Instead, we present two example programs which will illustrate that such a translation is possible in principle.

The first program consists of one instruction, a while loop, with the program labels p and q before and after the instruction.

```
[p]  while x =/= nil do
       i := i+1
       x := tl(x)
[q]
```

The program manipulates the two variables x and i. States are thus pairs $\langle p, e\rangle$ formed by the program location p and the environment e which assigns values v_x and v_i to the variables x and i. We write such a state as an atom $p(v_x, v_i)$. The program induces an infinite-state transition system; possible transitions are, for example,

$$\begin{array}{lll} p([a,b],0) & \longrightarrow & p([b],1), \\ p([b],1) & \longrightarrow & p([\,],2), \\ p([\,],2) & \longrightarrow & q([\,],2). \end{array}$$

Since the transition function must be total, we assume that there exist transitions modeling an explicit exception handling; for example,

$$\begin{array}{lll} p(3,i) & \longrightarrow & exception, \\ exception & \longrightarrow & exception. \end{array}$$

We translate the while loop above to the logic program below. Each program location corresponds to a predicate whose arguments correspond to the variables that are visible at that location. We express conditionals through the heads of the clauses. Since our framework requires that the program is total, we add clauses in order to model an exhaustive case statement (in a practical setting, such clauses could be presented implicitly). The transition systems induced by the while program and the logic program coincide.

$p([x|y], i) \leftarrow p(y, i+1)$

$p([\,], i) \leftarrow q([], i)$

$p(a, i) \leftarrow exception$ (for each other data constructor a)

$exception \leftarrow exception$

The CTL property $EF(\{q(v'_x, v'_i) \mid true\})$ specifies the set of all states at location p from which the location q can be reached. The values v_x for the variable x

in such states are exactly the finite lists. The set of these values can thus be presented as the least solution of the equation $list = cons(T_\Sigma, list) \cup nil$ over sets of trees, or of the program below (and this is also the result of the method presented in this paper).

$$\begin{array}{l} list(cons(x,y)) \leftarrow list(y) \\ list(nil) \end{array}$$

That is, if the while loop is executed with an initial value other than a finite list for x, then it will not reach the program point q (a fact which may be useful for debugging purposes).

The property $EG(\{p(v'_x, i'_x) \mid true\})$ holds for the states $p(v_x, i_x)$ where v_x is an infinite list (which models a circular list). The set of all infinite lists is the greatest solution of the equation $list = cons(T_\Sigma, list)$ over sets of infinite trees, or of the program below interpreted over the domain of infinite trees (again, this is also the result of the method outlined in this paper).

$$list(cons(x,y)) \leftarrow list(y)$$

That is, if the while loop is executed with an initial value other than an infinite list for x, then a program location other than p will be reached or an exception will be raised.

The next example is a program fragment (whose task is to reverse the list x) containing a typographical error ("$[tl(L)]$" instead of "$tl(L)$"). Again, we note p and q the program points before and after the while loop.

```
    y := []
[p] while x =/= nil do
      x := [tl(x)]
      y := [hd(L)|y]
[q] x:=y
[r]
```

We construct the corresponding logic program.

$$\begin{array}{l} init(x) \leftarrow p(x, [\,]) \\ p([x|x'],) \leftarrow p([x'], [x|y]) \\ p([\,], y) \leftarrow q([\,], y) \\ q(x,y) \leftarrow r(y,y) \\ p(a,i) \leftarrow exception \text{ (for each other data constructor } a\text{)} \\ exception \leftarrow exception \end{array}$$

Our method will derive that for any other initial value than the empty list for the variable x the program location q can never be reached.

3 Transition systems

Abstracting away from the fine structure of states and of transitions, we may present a reactive system with finitely many states as a transition system $\mathcal{S} = \langle S, \tau \rangle$ with the finite set S of states and the non-deterministic transition function $\tau : S \rightarrow 2^S$. The state q is a successor state of the state p if $q \in \tau(p)$. We translate $\mathcal{S}$ into a formula $\mathcal{P}_{\mathcal{S}}$ of propositional logic. Here, for each state p, we have a symbol p standing for a nullary predicate (or, a Boolean variable).

$$\mathcal{P}_{\mathcal{S}} \;=\; \bigwedge_{p \in S} (p \leftrightarrow \bigvee_{q \in \tau(p)} q) \tag{1}$$

An *interpretation* of $\mathcal{P}_{\mathcal{S}}$ is presented as a set $I \subseteq S$ of states; I specifies the set of all *atoms* p that are valued *true*. A *model* (or, solution) of $\mathcal{P}_{\mathcal{S}}$ is an interpretation under which the formula $\mathcal{P}_{\mathcal{S}}$ holds. Models are partially ordered by subset inclusion. If we require, as usual, that τ is total (i.e., $\tau(s) \neq \emptyset$ for all $s \in S$; thus,every state has at least one successor), then the least model of $\mathcal{P}_{\mathcal{S}}$ is the empty set $\emptyset$ and its greatest model is the set S of all atoms.

We now consider the safety property "P will never happen", written: $AG(S-P)$, or: $S - EF(P)$ in CTL notation, for some property $P \subseteq S$. The set $EF(P)$ of all states from which a state in P is reachable, is exactly the set of atoms in the least model of the following formula.

$$\mathcal{P}_{\mathcal{S}} \wedge P \;=\; \bigwedge_{p \notin P} (p \leftrightarrow \bigvee_{q \in \tau(p)} q) \;\wedge\; \bigwedge_{p \in P} p \tag{2}$$

The following explanation may help to understand this characterization of $EF(P)$. The formula $\mathcal{P}_{\mathcal{S}} \wedge P$ entails p iff there exists a sequence of implications $p \leftarrow p_1 \leftarrow \ldots \leftarrow p_n$ in $\mathcal{P}_{\mathcal{S}}$ and an implication $p_n \leftarrow$ *true*, which is, p_n is an element of P. The least model of $\mathcal{P}_{\mathcal{S}} \wedge P$ is the set of all entailed atoms ("all atoms that must be true in any model").

Now consider the inevitability property "P will always finally happen", written $AF(P)$, or $S - EG(S - P)$). The set $EG(S-P)$ is the set of atoms in the *greatest* model of the following formula. (The notation $\mathcal{P} \wedge\neg\, P$ must not be confused with $\mathcal{P} \wedge (S - P)$.)

$$\mathcal{P}_{\mathcal{S}} \wedge\neg\, P \;=\; \bigwedge_{p \notin P} (p \leftrightarrow \bigvee_{q \in \tau(p)} q) \;\wedge\; \bigwedge_{p \in P} \neg p \tag{3}$$

This may be explained as follows. The formula above entails $\neg p$ (i.e., it can be valid only if the model does not contain p) iff every maximal sequence of implications of the form $p \rightarrow p_1 \rightarrow p_2 \rightarrow \ldots$ in $\mathcal{P}_{\mathcal{S}}$ is finite and terminates with $p_n \rightarrow$ *false*. Thus, an atom p is in the greatest model of $\mathcal{P}_{\mathcal{S}} \wedge\neg\, P$ iff there exists an infinite sequence of implications avoiding *false*, which is, there exists an infinite sequence of transitions that starts in the state p and avoids the states in P.

The formulas (1), (2) and (3) are the *Clark completion* of logic programs that we formally introduce in Section 5. They are used to define the semantics of program logically.

4 Pushdown processes

We can model a system consisting of finite-state processes, one of which uses a pushdown stack as a data structure, by a pushdown automaton. In order to describe the ongoing behavior of such a system, we will consider input-less pushdown automata without an acceptance condition. Formally, a pushdown process is a tuple

$$\mathcal{A} = \langle Q, \Sigma, \delta, q^0 \rangle$$

consisting of a finite set of control states Q, the stack alphabet Σ, the non-deterministic transition function

$$\begin{aligned} \delta \subseteq \quad & (Q \times \Sigma) \times (Q \times \{\varepsilon\}) \\ \cup\, & (Q \times \{\varepsilon\}) \times (Q \times \Sigma) \end{aligned}$$

and the initial control state q^0. The states in the corresponding transition system

$$\mathcal{S}_{\mathcal{A}} = \langle Q \times \Sigma^\star, \tau_{\mathcal{A}} \rangle$$

are pairs $\langle q, w \rangle$ consisting of the control state $q \in Q$ and the stack contents $w \in \Sigma^\star$ (where $w = \varepsilon$ if the stack is empty). The transitions either read one symbol and remove it from the stack or add one.

$$\begin{aligned} \tau_{\mathcal{A}}(\langle q, w \rangle) = \{\langle q', w' \rangle \mid\ & w = a.w' \text{ where } a \in \Sigma,\ \langle \langle q, a \rangle, \langle q', \varepsilon \rangle \rangle \in \delta \text{ or} \\ & w' = a.w \text{ where } a \in \Sigma,\ \langle \langle q, \varepsilon \rangle, \langle q', a \rangle \rangle \in \delta \} \end{aligned}$$

Given a pushdown process $\mathcal{A}$, we define the program $\mathcal{P}_{\mathcal{A}}$ below. We now view Σ as a set of unary function symbols and ε as a constant symbol, and we consider terms over the signature $\Sigma \cup \{\varepsilon\}$.

$$\begin{aligned} \mathcal{P}_{\mathcal{A}} = \quad & \{q(a(x)) \leftarrow q'(x) \mid \langle \langle q, a \rangle, \langle q', \varepsilon \rangle \rangle \in \delta\} \\ \cup\, & \{q(x) \leftarrow q'(a(x)) \mid \langle \langle q, \varepsilon \rangle, \langle q', a \rangle \rangle \in \delta\} \end{aligned}$$

A program with the first kind of clauses only corresponds to a word automaton. A clause of the form $q(a(x)) \leftarrow q'(x)$ can be read as the instruction: "in state q, reading the word $w = a(x)$ with the first letter a and remaining suffix x, go to state q' and read the suffix x"; q is a final state iff the program contains a clause of the form $q(\varepsilon)$. Here, a word $a_1 a_2 \ldots a_n$ is represented as a unary tree $a_1(a_2(\ldots a_n(\varepsilon) \ldots))$.

The second kind of clause $q(x) \leftarrow q'(a(x))$ can be read as the "push" instruction: "in state q with stack contents w, go to state p' with stack contents $a(w)$."

The next remark is a consequence of the formal definition of the transition system $\mathcal{S}_{\mathcal{P}}$ induced by a program $\mathcal{P}$, which we defer to the next section.

Remark. The transition system $\mathcal{S}_{\mathcal{A}}$ of the pushdown process $\mathcal{A}$ and the transition system $S_{\mathcal{P}_{\mathcal{A}}}$ induced by the program $\mathcal{P}_{\mathcal{A}}$ that corresponds to $\mathcal{A}$ coincide. $\square$

The assumption that the transitions modify the size of the stack by exactly one symbol is not a proper restriction as long as acceptance is considered. If, however, we try to simulate the non-deterministic guessing of a new stack contents via a sequence of transitions that each guess one symbol to be added, then the necessary modification of the transition relation of the pushdown process would not leave the temporal properties invariant (because the guessing sequence can be infinite). Thus, the following generalization seems to be a proper one.

Definition 1 (Generalized pushdown processes). A generalized pushdown process is specified by any monolithic total program over the signature consisting of unary function symbols and one constant symbol.

We may restrict the syntax wlog. to three kinds of Horn clauses.

$$\begin{aligned} q(a(x)) &\leftarrow q'(x) \\ q(x) &\leftarrow q'(a(x)) \\ q(x) &\leftarrow q'(y) \end{aligned}$$

Given the clause $q(x) \leftarrow q'(y)$, every state of the form $\langle q', w' \rangle$ with any stack contents w' can be a successor state of the state $\langle q, w \rangle$. Thus, we here have a non-determinism of branching degree ω.

We will state already here the following theorem, which was shown in [3] for pushdown processes in the restricted sense, i.e., without non-deterministic guesses of stack contents (and hence, with a finite degree of branching).

Theorem 2. Given a generalized pushdown process and a CTL property φ with regular atomic propositions, the set of all states satisfying φ is again regular; its representation in the form of a non-deterministic finite automaton can be computed in single-exponential time (in the number of states).

Proof. The statement is an instance of Theorem 6. □

5 Monolithic programs

In a *multi-processor* transition system, the states have a structure and the transition function is defined by referring to that structure; we will consider such systems and their modeling through general logic programs in Section 7. In contrast, in a *monolithic* transition system, the transition function is defined directly on the states (i.e., as monolithic items). We can model such a system by a logic program P whose clauses' bodies contain exactly one atom. By extension, we then say that P is a *monolithic program.* Thus, a monolithic program $\mathcal{P}$ is given through implications of the form

$$p(t) \leftarrow p'(t')$$

where p and p' are predicates (different from *true*) and t and t' are terms over a given signature Σ of function symbols. When we refer to the logical semantics of $\mathcal{P}$, we use the formula below.

$$\mathcal{P} \;\equiv\; \bigwedge_{p} \;\forall x \;\; p(x) \leftrightarrow \bigvee_{i} \exists_{-x} \; (x = t_i \wedge p_i'(t_i'))$$

Here, p ranges over the set $Pred$ of all predicates defined by the program and i ranges over a suitable index set I_p such that $\{p(t_i) \leftarrow p'(t_i') \mid i \in I_p\}$ are all clauses with the predicate p in the head. As usual, $\exists_{-x}$ stands for the quantification of all variables in t_i and t_i' but x. For technical convenience, we assume that all predicates are unary; the results can easily be extended to the case without this restriction (for example, by extending the signature of function symbols with symbols forming tuples).

We note T_Σ the set of trees (i.e., ground terms) over the signature Σ. We use the same meta variables t, t', etc. for terms and trees. Given the program $\mathcal{P}$ defining the set of predicates $Pred$ and the signature Σ, the *Herbrand base* $\mathcal{B}_\mathcal{P}$ is the set of all *ground atoms* $p(t)$, which are applications of predicates to trees. (Note that $\mathcal{B}_\mathcal{P}$ does not include the propositional constant *true*.)

$$\mathcal{B}_\mathcal{P} = \{p(t) \mid p \in Pred, t \in T_\Sigma\}$$

A *ground clause* of $\mathcal{P}$ is an implication between ground atoms that is entailed by $\mathcal{P}$; thus, it is of the form $p(\sigma(t)) \leftarrow p'(\sigma(t'))$ where $p(t) \leftarrow p'(t')$ is a clause of $\mathcal{P}$ and $\sigma : Var \rightarrow T_\Sigma$ is a valuation (extended from variables to terms in the canonical way).

We will always assume that $\mathcal{P}$ is *total*, which means that for all ground atoms $p(t)$ there exists a ground clause of $\mathcal{P}$ of the form $p(t) \leftarrow p'(t')$.

An interpretation I, which we present as a subset of the Herbrand base, (i.e., $I \subseteq \mathcal{B}_\mathcal{P}$), interprets a predicate p as the set $\{t \in T_\Sigma \mid p(t) \in I\}$. A *model* of the program $\mathcal{P}$ is an interpretation under which the formula $\mathcal{P}$ is valid. Models are ordered by subset inclusion. The least model of $\mathcal{P}$, $lm(\mathcal{P})$, and the greatest model of $\mathcal{P}$, $gm(\mathcal{P})$, always exist. The least [greatest] model of a monolithic total program $\mathcal{P}$ is *always* the empty [universal] set, i.e., $lm(\mathcal{P}) = \emptyset$ and $gm(\mathcal{P}) = T_\Sigma$. The models of the programs that we will define next turn out to be more interesting. These programs consists of Horn clauses with additional conjuncts (the "oracles"). Note that $\neg\Gamma_p(t)$ is equivalent to $p(t) \in (\mathcal{B}_\mathcal{P} - \Gamma)$. We always use $\mathcal{P} \wedge \Gamma$ for defining least models, and $P \wedge\neg\; \Gamma$ for defining greatest models.

Definition 3 (Programs with oracles). Given a monolithic program $\mathcal{P}$ and a subset Γ of the Herbrand base, we define two kinds of programs with oracles.

$$\mathcal{P} \wedge \Gamma = \mathcal{P} \cup \{p(x) \leftarrow \Gamma_p(x) \mid p \in Pred\}$$
$$\mathcal{P} \wedge\neg\; \Gamma = \{p(t) \leftarrow p'(t') \wedge \neg\Gamma_p(t) \;\mid p(t) \leftarrow p'(t') \text{ is a clause in } \mathcal{P}\}$$

The operational semantics of $\mathcal{P}$ can be described as a transition system

$$\mathcal{S}_\mathcal{P} = \langle \mathcal{B}_\mathcal{P}, \tau_\mathcal{P} \rangle$$

whose states are the ground atoms (not (!) including *true*), and whose transition function $\tau_{\mathcal{P}} : \mathcal{B}_{\mathcal{P}} \to 2^{\mathcal{B}_{\mathcal{P}}}$ is defined as follows.

$$\tau_{\mathcal{P}}(p(t)) = \{p'(t') \mid p(t) \leftarrow p'(t') \text{ is a ground clause of } \mathcal{P}\}$$

Since we have assumed that $\mathcal{P}$ is total, we have that $\tau_{\mathcal{P}}(s) \neq \emptyset$ for all states s ("the transition function $\tau_{\mathcal{P}}$ is total"). The fixpoint semantics of P is given through the $T_{\mathcal{P}}$ operator on subsets of the Herbrand base.

$$T_{\mathcal{P}}(I) = \{p(t) \mid p(t) \leftarrow p'(t') \text{ is a ground clause of } \mathcal{P} \text{ and } p'(t') \in I\}$$

We immediately note the connection with the inverse of the transition function (as usual, $\tau_{\mathcal{P}}^{-1}(P) = \{s \in \mathcal{B}_{\mathcal{P}} \mid \tau_{\mathcal{P}}(s) \subseteq P\}$ for subsets P of states).

$$\tau_{\mathcal{P}}^{-1} = T_{\mathcal{P}}$$

In order to define the logic CTL over the transition system $\mathcal{S}_{\mathcal{P}}$ induced by the program $\mathcal{P}$, we first need to fix the set *Prop* of *atomic propositions*. As in the finite-state case, an atomic proposition Γ denotes a set of states, which we also write as Γ. When dealing with algorithmic issues, we will require that Γ can be finitely represented. This is possible, for example, when we require that Γ is a *regular atomic proposition*, which means that the set

$$\Gamma_p = \{t \in T_{\Sigma} \mid p(t) \in \Gamma\}$$

is a recognizable set of trees, for each predicate p.

Usually, the denotation of an atomic proposition is described via the detour of a *labeling* function $L : S \to 2^{Prop}$, where Γ denotes the set $\{s \in S \mid \Gamma \in L(s)\}$. In our setting, the labeling function L is implicit by $L(s) = \{\Gamma \in Prop \mid s \in \Gamma\}$). The finite representation of the sets Γ_p for the atomic propositions occurring in the CTL formula is part of the input. (It is not clear how L could be represented finitely otherwise.)

Given the set *Prop* of atomic propositions, the set of formulas of the logic CTL and their meaning are defined as in the finite-state case.

$$\varphi \ ::= \ \Gamma \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid EX(\varphi) \mid E(\varphi_1 U \varphi_2) \mid A(\varphi_1 U \varphi_2)$$

In addition, we use the following abbreviations: $AX(\varphi) = \neg EX(\neg\varphi)$, $EF(\varphi) = E(\mathit{true} U \varphi)$, $AF(\varphi) = A(\mathit{true} U \varphi)$, $EG(\varphi) = \neg AF(\neg\varphi)$, $AG(\varphi) = \neg EF(\neg\varphi)$.

We write $\mathcal{S}, p \models \varphi$ if the transition system $\mathcal{S}$ with the initial state p satisfies the formula φ. Given $\mathcal{S}$, we simply write φ for the set of all states for which the formula φ is satisfied.

$$\varphi \equiv \{s \mid \mathcal{S}, s \models \varphi\}$$

Theorem 4 (CTL properties and program semantics). Given the transition system $\mathcal{S}_{\mathcal{P}}$ corresponding to the monolithic program $\mathcal{P}$, each set of states denoted by a CTL formula φ can be characterized in terms of subsets of the

Herbrand base defined through the semantics of programs with oracles, via the following correspondences.

$$
\begin{aligned}
EX(P) &= T_{\mathcal{P}}(P) \\
AX(S-P) &= S - T_{\mathcal{P}}(P) \\
EF(P) &= lm(\mathcal{P} \wedge P) \\
AF(P) &= S - gm(\mathcal{P} \wedge\neg\, P) \\
E(S-P_1)UP_2 &= lm((\mathcal{P} \wedge\neg\, P_1) \wedge P_2) \\
A(S-P_1)UP_2 &= S - \big(gm(\mathcal{P} \wedge\neg\, (S-P_2)) \cup lm((\mathcal{P} \wedge\neg\, P_2) \wedge P_1)\big)
\end{aligned}
$$

These correspondences hold for all subsets of states $P \subseteq \mathcal{B}_{\mathcal{P}}$, which may be defined by atomic propositions Γ or by CTL formulas.

Proof. The first two equalities hold by the definitions of $EX = \tau_{\mathcal{S}_{\mathcal{P}}}^{-1}$ and T_P. The next two equalities follow by: (1) the definition of EF $[EG]$ through the least [greatest] fixpoint of EX, (2) the correspondence between the semantics of logic programs defined by the least [greatest] fixpoint and the least [greatest] model (which extend to programs with oracles), and (3) the following identities between fixpoint operators over properties $P \subseteq \mathcal{B}_{\mathcal{P}}$, for any given property $P^0 \subseteq \mathcal{B}_{\mathcal{P}}$.

$$
\begin{aligned}
\lambda P.\ (P^0 \cup EX(P)) &= T_{\mathcal{P} \wedge P^0} \\
\lambda P.\ ((S-P^0) \cap EX(P)) &= T_{\mathcal{P} \wedge\neg\, P^0}
\end{aligned}
$$

The proof of the two remaining equalities uses two basic general facts about the operational semantics and the model-theoretic semantics of a given logic program $\mathcal{P}$. A ground atom $p(t)$ has an execution in $\mathcal{S}_{\mathcal{P}}$ that leads to the atom *true*, i.e., $p(t) \in EF(\{true\})$, if and only if $p(t) \in lm(\mathcal{P})$; it has a non-failing execution in $\mathcal{S}_{\mathcal{P}}$ i.e., $p(t) \in EG(\mathcal{B}_{\mathcal{P}} \cup \{true\})$ if and only if $p(t) \in gm(\mathcal{P})$. We apply these two facts to programs with oracles instead of $\mathcal{P}$.[1] We note that $E(S-P_1)UP_2$ is the set of all ground atoms $p(t)$ which have an execution that reaches a state in P_2 while avoiding states in P_1, i.e., which reaches *true* in the program $(\mathcal{P} \wedge\neg\, P_1) \wedge P_2$. Similarly, a state s is not in $A(S-P_1)UP_2$ if it either has an execution that never reaches a state in P_2, i.e., it has a non-terminating execution in the program $\mathcal{P} \wedge\neg\, (S-P_2)$, or it has an execution that reaches a state s' in P_1 while avoiding states in P_2 (in the execution up to, and including s'), i.e., it has an execution in the program $(\mathcal{P} \wedge\neg\, P_2) \wedge P_1$. $\square$

6 Set-based analysis

In set-based analysis, an abstract semantics of a program is represented as a particular solution of a formula with set-valued variables (often called a set-constraint). The formula is syntactically inferred from the program. The values

[1] By our assumptions on *monolithic total* programs $\mathcal{P}$, $EF(\{true\}) = \emptyset$ because *true* does not appear in the body of a clause in $\mathcal{P}$, and $EG(\mathcal{B}_{\mathcal{P}}) = \mathcal{B}_{\mathcal{P}}$ because the transition function is total.

in the solution are regular sets of trees. Thus, they can be represented through non-deterministic tree automata, which have a linear emptiness test. The algorithmic essence of set-based analysis is the *solving* of the set constraint. This means to compute the particular solution that represents the *set-based* abstract semantics, which again means to compute a non-deterministic tree automaton that represents the solution.

We will give here an introduction to the set-based analysis of logic programs with *uniform programs* (as in [16]). Uniform programs subsume several classes of set constraints used in the set-based analysis of logic and imperative languages (e.g., in [20,18,13,8]) modulo simple translations. Note that we can view any logic program as a formula whose monadic predicate symbols stand for variables ranging over sets of trees, and whose individual variables ranging over trees are all quantified; thus, its free variables are set-valued. We now need to consider general logic programs, which are sets of Horn clauses $p(t) \leftarrow p_1(t_1) \wedge \ldots \wedge p_n(t_n)$ with any number $n \geq 0$ of body atoms. The definitions of the least [greatest] model semantics and fixpoint semantics for monolithic programs in Section 5 carry over directly to the general case (with some extra notational burden). We will discuss the operational semantics in Section 7.

A *uniform program* [16] consists of Horn clauses in one of the following two forms. (In a *linear* term t, each variable occurs at most once.)

- $p(t) \leftarrow p_1(x_1) \wedge \ldots \wedge p_k(x_m)$, where the term t is linear.
- $q(x) \leftarrow p_1(t_1) \wedge \ldots \wedge p_m(t_m)$, where $t_1, \ldots, t_m$ are any terms over Σ.

We derive a uniform program $\mathcal{P}^\sharp$ from any logic program $\mathcal{P}$ in the following way. For every Horn clause

$$p(t) \leftarrow \text{body}$$

where the term t has the n variables $x_1, \ldots, x_n$, we apply renamings x_{ij} to every occurrence of x_i in t in order to obtain a linear term $\tilde{t}$. We introduce n new predicate symbols p_i. Then, $\mathcal{P}^\sharp$ contains the following $n+1$ clauses for every such Horn clause.

$$\begin{aligned} p(\tilde{t}) &\leftarrow \textstyle\bigwedge_{i=1}^{n} \bigwedge_j p_i(x_{ij}) \\ p_1(x_1) &\leftarrow \text{body} \\ &\vdots \\ p_n(x_n) &\leftarrow \text{body} \end{aligned}$$

The program $\mathcal{P}^\sharp$ expresses exactly the so-called set-based abstraction of $\mathcal{P}$ defined in [18] in semantic terms. We can easily translate every uniform program into one whose Horn clauses are in one of the following three forms.

- $p(f(x_1, \ldots, x_n)) \leftarrow p_1(x_1) \wedge \ldots \wedge p_n(x_n)$
  where $n \geq 0$ and $x_1, \ldots, x_n$ pairwise different
- $p(x) \leftarrow p_1(x) \wedge \ldots \wedge p_m(x)$
- $p(x) \leftarrow p'(t)$

A *nondeterministic tree automaton* is a uniform program consisting of the first kind of clauses only. Predicates correspond to states.

The second kind of rules introduces conjunctions of states. An *alternating tree automaton* is a uniform program consisting of the first two kinds of clauses. Note that this is a nice formalization of this concept, without the need to define complicated acceptance conditions.

Finally, the third kind of rule introduces a kind of *push* operation. The rule $p(x) \leftarrow p'(f(x, y))$ can be read as the instruction: "in state p with stack contents t (where t is a tree), go to state p' with stack contents $f(t, t')$ where t' is any tree chosen non-deterministically" (which can be viewed as pushing the context $f(\cdot, t')$ onto the stack).

The following facts hold for the interpretation of programs over finite trees and over infinite trees. The first fact is obvious; the second and third are shown in [16,6] (the second statement of Fact 2 follows from the form of uniform programs). We call a model $M \subseteq \mathcal{B}_{\mathcal{P}}$ regular if it can be defined as the least (or, equivalently, as the greatest) model of a tree automaton program; i.e., if all sets $M_p = \{t \in T_\Sigma \mid p(t) \in M\}$ are regular sets of trees (in the standard sense).

Facts about set-based analysis.

1. The uniform program $\mathcal{P}^\sharp$ derived from the logic program $\mathcal{P}$ in the way described above approximates $\mathcal{P}$ in the following sense.

$$lm(\mathcal{P}) \subseteq lm(\mathcal{P}^\sharp)$$
$$gm(\mathcal{P}) \subseteq gm(\mathcal{P}^\sharp)$$

2. The least [greatest] model of a uniform program $\mathcal{P}^\sharp$ is regular; i.e., it can represented by a non-deterministic tree automaton, which again can be represented as a program $\mathcal{P}_{\mathrm{sba}}$ (which we define to be the output of the set-based analysis).

$$lm(\mathcal{P}^\sharp) = lm(\mathcal{P}_{\mathrm{sba}})$$
$$gm(\mathcal{P}^\sharp) = gm(\mathcal{P}_{\mathrm{sba}})$$

   More specifically, if the predicate p is defined by clauses with the heads $p(t_k[x_1, \ldots, x_{n_k}])$ with n_k free variables, then there exist regular sets of trees $T_1^k, \ldots, T_{n_k}^k$ (for each k) such that the set of all p-atoms in the least model [greatest] is a union of *Cartesian products* in the following sense.

$$lm(\mathcal{P}^\sharp) = \bigcup_{p \in Pred} \bigcup_k p(t_k[T_1^k, \ldots, T_{n_k}^k])$$

3. The step $\mathcal{P}^\sharp \mapsto \mathcal{P}_{\mathrm{sba}}$ can be computed in single-exponential time. This is also a lower bound for the case of an arbitrary signature; the best known lower bound for the word case is PSPACE.

Theorem 5 (Conservative approximation of CTL properties). Given a transition system specified by a monolithic program $\mathcal{P}$, we can conservatively approximate applications of CTL operators in the following sense.

$$\begin{array}{ll}
EX(P) & \subseteq T_{\mathcal{P}^\sharp}(P) \\
AX(S-P) & \supseteq S - T_{\mathcal{P}^\sharp}(P) \\
EF(P) & \subseteq lm(\mathcal{P}^\sharp \wedge P) \\
AF(P) & \supseteq S - gm(\mathcal{P}^\sharp \wedge\neg\, P) \\
E(S-P_1)UP_2 & \subseteq lm((\mathcal{P}^\sharp \wedge\neg\, P_1) \wedge P_2) \\
A(S-P_1)UP_2 & \supseteq S - \big(gm(\mathcal{P}^\sharp \wedge\neg\, (S-P_2)) \cup lm((\mathcal{P} \wedge\neg\, P_2) \wedge P_1)\big)
\end{array}$$

Proof. We only need to put together Fact 1 and Theorem 4. □

Theorem 6 (Set-based verification of CTL properties). Given a monolithic uniform program $\mathcal{P}$ and a CTL property φ with regular atomic propositions, the set of all states satisfying φ is again regular; its representation through a non-deterministic tree automaton can be computed in single-exponential time (in the number of predicates of $\mathcal{P}$).

Proof. This follows by Facts 2 and 3 and Theorem 4. □

Set-based abstract verification and types. Theorem 5 suggests the following procedure. Given any monolithic program $\mathcal{P}$ and a CTL property φ built up from regular atomic propositions and the operator EX, EF, EG and EU, we compute a superset $\varphi^\sharp$ of the set of all states satisfying φ in single-exponential time. The set $\varphi^\sharp$ consists of all states that satisfy φ according to the set-based abstract semantics of $\mathcal{P}$.[2]

$$\varphi^\sharp = \{p(t) \in \mathcal{B}_\mathcal{P} \mid \mathcal{S}_{\mathcal{P}^\sharp}, p(t) \models \varphi\}$$

By Fact 2, we know that $\varphi^\sharp$ is a union of Cartesian products of regular sets of trees.

$$\varphi^\sharp = \bigcup_{p \in Pred} \bigcup_k p(t_k[T_1^k, \ldots, T_{n_k}^k])$$

We call the set $T_i = \bigcup_k T_i^k$ the *type* of the variable x_i, for $i = 1, \ldots, n$. If the value for x_i in a ground instance $s \in \mathcal{B}_\mathcal{P}$ of the atom $p(t)$ does not lie in T_i, then the state s does not satisfy φ.

In summary, we obtain an abstract *falsification* procedure for the ∃-CTL fragment of CTL, and an abstract *verification* procedure for the ∀-CTL fragment.

Set constraints. It is also possible to apply the two approaches in [18] and [25] of set-based analysis of logic programs with least [greatest] models, which infer

[2] The set-based abstract semantics of $\mathcal{P}$ is given by the transition system $\mathcal{S}_{\mathcal{P}^\sharp}$ induced by $\mathcal{P}^\sharp$. Here, a state is a conjunction of atoms. Its successor states are obtained by the *simultaneous* rewriting of all its atoms. We ignore the newly introduced predicates; these can be eliminated easily.

a definite [co-definite] set constraint from a logic program and then computes its solution in single-exponential time [17,8]. We have preferred the formalism of uniform programs for the presentation in this paper because it allows us to greatly simplify the presentation; there is not much difference technically, and no difference at all in the case of unary function symbols (which is relevant for pushdown processes).

7 Parallel programs

In this section, we will consider transition systems specified by *parallel programs* that consist of Horn clauses of the form

$$p(t) \leftarrow p_1(t_1) \wedge \ldots \wedge p_n(t_n)$$

with any number $n \geq 1$ of body atoms. Operationally, the conjunction of body atoms corresponds to parallel composition. Thus, a parallel program defines a concurrent system, which has, however, only a restricted way of communication (in particular, there is no synchronization between parallelly composed atoms). The logic program P defines a *fair* transition system $\mathcal{S}_\mathcal{P} = \langle S_P, \tau_\mathcal{P} \rangle$. The states are the *ground queries*, which are nonempty conjunctions of ground atoms.

$$S_P = \{p_1(t_1) \wedge \ldots \wedge p_m(t_m) \mid m \geq 1 \text{ and } p_1(t_1), \ldots, p_m(t_m) \in \mathcal{B}_\mathcal{P}\}$$

The one-step transition relation $\tau_\mathcal{P}$ is defined as usual for ground derivations of logic programs or constraint logic programs [21,19]. We identify conjunctions modulo commutativity and associativity of conjunction. We give the definition of $\tau_\mathcal{P}$ in a form that relies on this convention (i.e., every of the conjuncts forming a state can be chosen to be the first in a representation of the state).

$$\begin{aligned}
&\tau_\mathcal{P}(p_1(t_1) \wedge \ldots \wedge p_m(t_m)) \\
&= p_1(t_1) \wedge \ldots \wedge p_{i-1}(t_{i-1}) \wedge \mathit{body} \wedge p_{i+1}(t_{i+1}) \wedge \ldots \wedge p_m(t_m) \mid \\
&\qquad\qquad\qquad i \in \{1, \ldots, m\},\ p_i(t_i) \leftarrow \mathit{body} \text{ is a ground clause of } \mathcal{P}\}
\end{aligned}$$

The fairness condition of the transition system is related to the fairness of the selection rule which is usually associated with the execution of logic programs. It says that in every execution sequence containing the state $p_1(t_1) \wedge \ldots \wedge p_m(t_m)$, a ground clause with head $p_j(t_j)$ is eventually applied to yield a successor state (for every $j = 1, \ldots, m$).

We require that the program $\mathcal{P}$ is total (which is, there exists a ground clause with the head $p(t)$ for every ground atom $\mathcal{B}_\mathcal{P}$), and thus obtain that $\tau_\mathcal{P} : S \rightarrow 2^S$ is totally defined (even if the fairness condition is taken into account).

In order to define the logic CTL over the transition system $\mathcal{S}_\mathcal{P}$ induced by the parallel program $\mathcal{P}$, we will first fix a restricted set *Prop* of *atomic propositions*. These are of the form $\mathit{states}(\Gamma)$ for some subset Γ of the Herbrand base. It is defined as the set of the states whose conjuncts all lie in Γ.

$$\mathit{states}(\Gamma) = \{p_1(t_1) \wedge \ldots \wedge p_m(t_m) \mid p_1(t_1), \ldots, p_m(t_m) \in \Gamma\}$$

Definition 7 (Restricted CTL formulas). A restricted CTL formula (for a given set of predicates $Pred$ and a signature Σ defining the Herbrand base $\mathcal{B}_{\mathcal{P}}$) is a CTL formula such that either: the atomic propositions P are of the form $P = \mathit{states}(\Gamma)$ where $\Gamma \subseteq \mathcal{B}_{\mathcal{P}}$ and the quantifiers are only among EX, EF, EG, or: the atomic propositions P are of the form $P = \mathcal{B}_{\mathcal{P}} - \mathit{states}(\Gamma)$ and the quantifiers are among AX, AF, AG.

Theorem 8 (CTL properties and semantics of parallel programs). Given the fair transition system $\mathcal{S}_{\mathcal{P}}$ corresponding to the parallel program $\mathcal{P}$, each set of states φ denoted by a restricted CTL formula φ can be characterized through the semantics of programs with oracles, via the following correspondences (for all subsets of states $\Gamma \subseteq \mathcal{B}_{\mathcal{P}}$).

$$\begin{aligned} EF(\mathit{states}(\Gamma)) &= \mathit{states}(lm(\mathcal{P} \wedge \Gamma)) \\ AF(S - \mathit{states}(\Gamma)) &= S - \mathit{states}(gm(\mathcal{P} \wedge\neg\, \Gamma)) \\ EG(\mathit{states}(S - \Gamma)) &= \mathit{states}(gm(\mathcal{P} \wedge\neg\, \Gamma)) \\ AG(S - \mathit{states}(\Gamma)) &= S - \mathit{states}(lm(\mathcal{P} \wedge\neg\, \Gamma)) \end{aligned}$$

Proof. The proof follows the lines of the proof of Theorem 4. □

We can now rephrase Theorem 5 for parallel programs and for restricted CTL formulas with the CTL operators appearing in the statement above. Therefore, we can apply [abstract] set-based verification (in the sense of Theorem 6) also to this setting.

Les us note that parallel programs over nullary predicates represent *Basic Parallel Processes* (see, e.g., [15]). We leave open the question of the connections with other, infinite-state concurrent systems.

8 Conclusion and future work

The use of logic programs as specifications of transition systems gives us a new view on accurate and abstract verification of CTL properties for finite and infinite systems. The use of set-based analysis as a method to define an approximation function gives a new view on pushdown processes (namely, as the target of the approximation function) and yields a notion of descriptive types of program variables (wrt. a given CTL property). The use of set-based analysis as an algorithm to compute the conservative approximation of CTL properties gives a new view of model checking of pushdown processes, namely as constraint-solving, viz. theorem proving.

One obvious issue of further research is the extension of our method of abstract verification via set-based analysis to other temporal logics. The extension to the alternation-free mu-calculus seems to be possible directly. The work in [6] is a first step for the extension to the full mu-calculus.

The applications of our method that we have presented in this paper indicate its usefulness for the detection of “simple” programming errors, i.e., for the falsification of programs with respect to behavioral properties. The method may, however, also be useful for the verification of parameterized systems. The idea here is to use logic programs that first non-deterministically guess a parameter (say, the length of the token ring in the example used in [9]) and then simulate the corresponding system.

In the finite-state case, computing the model of a program with oracles is the fixpoint iteration in standard model checking. The programs are Boolean formulas. This is a new view of model checking which may be interesting in connection with BDD's. Also, we need to explore the connection with work in [24] which shows the equivalence of solving Boolean equation systems and model-checking in the modal μ-calculus.

In the finite-state case, programs with oracles are closely related to the product construction of [1]. It may be interesting to explore the connection with the similar product construction of [27] in the infinite-state case.

Acknowledgments. We thank Ahmed Bouajjani, Javier Esparza, David McAllester, Damian Niwiński and Moshe Vardi for fruitful discussions and the anonymous referees for useful remarks.

References

1. O. Bernholtz, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Computer Aided Verification, Proc. 6th Int. Workshop*, volume 818 of *LNCS*, pages 142–155, Stanford, California, June 1994. Springer-Verlag. Full version available from authors.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model Checking. In *CONCUR'97.* LNCS 1243, 1997.
3. A. Bouajjani and O. Maler. Reachability Analysis of Pushdown Automata. In *Infinity'96.* tech. rep. MIP-9614, Univ. Passau, 1996.
4. F. Bourdoncle. Abstact debugging of higher-order imperative languages. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI'93)*, LNCS, pages 46–55. ACM Press, 1993.
5. O. Burkart and B. Steffen. Composition, Decomposition and Model-Checking of Pushdown Processes. *Nordic Journal of Computing*, 2, 1995.
6. W. Charatonik, D. McAllester, D. Niwiński, A. Podelski, and I. Walukiewicz. The Horn mu-calculus. To appear in Vaughn Pratt, editor, Proceedings of the 13th IEEE Annual Symposium on Logic in Computer Science.
7. W. Charatonik, D. McAllester, and A. Podelski. Computing the greatest model of the set-based abstraction of logic programs. Presented at the Dagstuhl Workshop on Tree Automata, October 1997.
8. W. Charatonik and A. Podelski. Co-definite set constraints. In T. Nipkow, editor, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications*, LNCS, Tsukuba, Japan, March-April 1998. Springer-Verlag. To appear.
9. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 343–354. ACM Press, 1992.
10. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. POPL '92*, pages 83–94. ACM Press, 1992.

11. D. R. Dams. *Abstract interpretation and partition refinement for model checking.* PhD thesis, Eindhoven University of Technology, 1996.
12. P. Devienne, J.-M. Talbot, and S. Tison. Solving classes of set constraints with tree automata. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming - CP97*, volume 1330 of *LNCS*, pages 68–83. Springer-Verlag, October 1997.
13. P. Devienne, J.-M. Talbot, and S. Tison. Solving classes of set constraints with tree automata. In G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming - CP97*, volume 1330 of *LNCS*, Berlin, Germany, October 1997. Springer-Verlag.
14. E. Emerson and E. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.
15. J. Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatika*, 34:85–107, 1997.
16. T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 300–309, July 1991.
17. N. Heintze and J. Jaffar. A decision procedure for a class of set constraints (extended abstract). In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 42–51, 1990.
18. N. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 197–209, January 1990.
19. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19/20:503–582, May-July 1994.
20. N. D. Jones and S. S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.
21. J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Springer-Verlag, Berlin, Germany, second, extended edition, 1987.
22. C. Loiseaux, S.Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
23. D. E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie Mellon University, 1993.
24. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems.* Phd thesis, Technische Universität München, 1997.
25. A. Podelski, W. Charatonik, and M. Müller. Set-based error diagnosis of concurrent constraint programs. Submitted for publication. Available under www.mpi-sb.mpg.de/~podelski/papers/diagnosis.ps, 1998.
26. Y. Ramakrishna, C. Ramakrishnan, I. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient model checking using tabled resolution. In *Computer Aided Verification (CAV'97)*, LNCS 1254. Springer-Verlag, June 1997.
27. M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, pages 167–176, Ithaca, 1987. IEEE Computer Society Press.
28. I. Walukiewicz. Pushdown Processes: Games and Model Checking. In *CAV'96*. LNCS 1102, 1996.
29. C. Weidenbach. Spass version 0.49. *Journal of Automated Reasoning*, 18(2):247–252, 1997.