Constraint-Based Model Checking of Data-Independent Systems*

Beata Sarna-Starosta and C. R. Ramakrishnan

Department of Computer Science,
State University of New York at Stony Brook
Stony Brook, New York, 11794-4400, U.S.A.
E-mail: {bss,cram}@cs.sunysb.edu

Abstract Data-independent systems are an important class of infinite-state systems which can be subject to model checking by first building finite-state property-preserving abstractions. Exploiting data independence in practice involves user guidance, either in terms of the abstraction itself or in terms of symmetry properties of the system. In this paper we present a constraint-based verification technique that automatically handles data-independent systems. Our technique introduces a unified, automata-based model for infinite-state systems and LTL formulas. The technique can be seen as a generalization of explicit state model checker for reachability and LTL properties. We have implemented our technique using logic programming with tabulation and constraints. We also describe an extension to the automata model that permits verification of a richer class of systems. We show its power by analyzing configuration (security) vulnerabilities in a computer system.

1 Introduction

Many real-world systems and designs are naturally modeled as systems with infinite state space. Systems that have a finite number of control locations (analogous to program counter values) but manipulate data ranging over arbitrary unbounded domains are used to model software artifacts and control systems such as communication protocols or hardware controllers. Such systems can be modeled as extended finite automata (EFA) where each control location has a set of local variables and the transitions have (i) a guard that tests the valuation of variables in the source location and (ii) a relation that maps values of variables in the source location to the variables in the destination location. For instance, Figure 1 shows an EFA model for a two-place FIFO buffer. Many infinite-state systems use only operations such as input, output and copy that do not inspect the individual values themselves. Notice for instance, that input values to the buffer in Figure 1 do not affect the system's observable behavior except for the corresponding changes to the output values. Such infinite-state systems are called *data-independent* [25].

* This research was supported in part by NSF grants EIA-9705998, CCR-9876242, IIS-0072927, CCR-0205376, and ONR grant N000140110967.

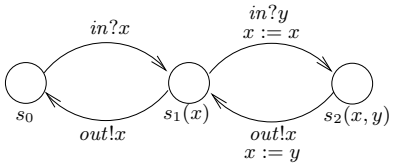

Figure 1. Example EFA for 2-place FIFO buffer.

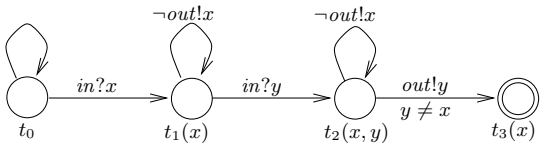

Figure 2. Automaton representing a run with "out of order" message delivery.

The State of the Art: Since the control behavior of data-independent systems does not depend on the actual values of the data, such systems can be verified using traditional model checking techniques as follows. First, temporal properties of such systems can be specified using only data values drawn from a finite domain. Then a data-independent system can be abstracted to a finite-state system by restricting the data variables to take values only over this finite data domain. The crucial problem in verifying data-independent systems is, then, to identify the appropriate (finite) data domain that makes the abstraction property-preserving. In the seminal work of [25], this abstraction is performed manually. Since then, several techniques [12,10,13,17] for identifying the appropriate abstraction have been developed. However, existing techniques either require user guidance or expect the temporal properties to be solely about the control behavior of the system. (These issues are explored in more detail in Section 7.)

Summary of Our Approach: We model data-independent systems, as well as their temporal properties, as extended finite automata. For instance, Figure 2 shows an EFA representing runs that deliver messages "out-of-order", i.e. where there are two data objects, x and y, such that x is read before y, but y is written out before x. The structure of EFAs is such that a product of two two such automata is also an EFA. Following the automata-based approach we can verify properties of systems by looking for particular runs in their product EFAs: safety and liveness can be verified using reachability analysis, while LTL properties can be checked by good cycles detection.

We represent EFAs as constraint logic programs; analyzing the runs can be then posed as query evaluation over these programs. Note that, resolution, the widely-used query evaluation mechanism, ensures that the variables in the EFA are bound only to the extent necessary to answer the query. This follows our earlier approaches to constructing model checkers of finite and infinite-state

systems based on query evaluation over (constraint) logic programs. Moreover, the EFA product construction itself can be encoded as a constraint logic program, meaning that the product automaton itself is constructed on demand: only portions of the product automaton needed to answer the particular query are materialized. Finally, interpreting the query over a domain of equalities and disequalities ensures that we can verify temporal properties of data-independent systems without attempting to enumerate specific valuations of the data variables. This approach can be used to automatically verify the data-independent systems that have been reported in the literature [25,10,13] without needing any user intervention or annotations. Our approach can also automatically verify correctness properties of cache-memory systems which have not been amenable for automatic treatment using the existing techniques (see Section 4).

Extensions and Applications: We initially define EFAs such that they manipulate only equality and disequality constraints (Section 3). Even these relatively simple models are expressive enough to represent data-independent systems as defined in [25], as well as their extensions [13]. We describe the verification of cache-memory systems using such EFAs in Section 4. We further extend EFAs to use membership constraints (e.g. $x \in y$) in order to represent a richer class of systems (Section 5). With these constraints, the model checking problem is no longer decidable. We hence devise abstractions that ensure termination of the analysis but with very little loss of information in practice. We use this technique to verify properties of a generalized cache-memory system, as well as to detect vulnerabilities in computer system configurations (Section 6).

The technique presented in this paper provides a way for direct and automatic verification of data-independent systems. The system models may in fact be specified in a familiar process algebraic notation that can be automatically translated to the underlying EFAs. This enables direct application of our technique on system models constructed for use in finite-state model checkers such as XMC [22]. The systems handled by the model checker include those that compare data variables using equality (e.g. $x = y$ where x and y are data variables) and disequality (e.g., $x \neq y$) tests. The temporal properties can naturally express both data and control behaviors of these systems. Furthermore, our implementation of this technique can be seen as an extension to an explicit-state model checker for LTL properties [2,20].

2 Preliminaries

We assume the standard notion of variables, function symbols, predicates, terms, substitutions, and unification [16]. Variables range over an enumerable set $\mathcal{V}$; we use $x, y, \ldots$ to denote variables. Function symbols range over $\mathcal{F}$; 0-ary function symbols are called constants (denoted by the set $\mathcal{C}$). We use $\mathcal{T}$ to denote the set of all terms constructed from $\mathcal{V}$ and $\mathcal{F}$; σ, θ to denote substitutions; and mgu to denote the most general unifier of a set of terms. A term t under a substitution σ is denoted by $t\sigma$. By $\sigma[t/x]$ we denote a substitution σ' that maps x to t and is identical to σ everywhere else.

Constraints, Constraint Languages and Assertions: A formal definition of extended finite automata (see Section 3) is based on a language of constraints. Constraint languages are parameterized with respect to a set of *primitive constraints* PC. For instance, *equality* and *disequality* constraints are defined using the following set of primitive constraints:

$$PC_{\{=\}} = \{v_1 = v_2, v_1 \neq v_2\}$$

where $v_1, v_2 \in \mathcal{V} \cup \mathcal{C}$. A constraint language L defined with respect to PC is built using the constraints in PC, Boolean connectives $\wedge$ and $\vee$, and existential quantification $\exists$ over variables. Elements of L are also called *assertions* and are denoted by φ (possibly primed and/or subscripted). Formally, L_s, the language of constraints defined over PC_s is the smallest set such that (i) $PC_s \subseteq L_s$; (ii) $\varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2 \in L_s$ if $\varphi_1, \varphi_2 \in L_s$; and (iii) $\exists v.\ \varphi \in L_s$ if $\varphi \in L_s$ and $v \in \mathcal{V}$. We assume that PC is closed with respect to negation, and hence do not have an explicit connective for negation.

The set of variables in an assertion φ is denoted by $vars(\varphi)$. We use the standard notion of bound and free variables (due to quantifiers) in assertions. The set of bound variables in an assertion is denoted by $bv(\varphi)$ and the set of free variables by $fv(\varphi)$. We also use the standard notion of *meaning* of assertions in $L_{\{=\}}$, by interpreting them over the data domain $\mathcal{C}$. With each assertion φ we associate a set $[\![\varphi]\!]$ of substitutions mapping $fv(\varphi)$ to $\mathcal{C}$. Note that we define the meaning by substituting only the free variables of an assertion. Each substitution σ in $[\![\varphi]\!]$ is said to "satisfy" φ (written as $\sigma \models \varphi$).

In the first part of the paper, we use only equality and disequality constraints, since they suffice to describe and analyze data-independent systems. We subsequently expand our techniques to analyze systems whose control behaviors are dependent on certain infinite-domain values. We model such systems using an expanded constraint language $L_{\{=,\in\}}$ which considers two distinguished set constants, $\{\}$ (empty set) and $\mathcal{U}$ (the universal set) in $\mathcal{C}$, and has the following set of primitive constraints:

$$PC_{\{=,\in\}} = \{v_1 = v_2, v_1 \neq v_2, v_1 \in v_2, v_1 \notin v_2\}$$

where $v_1, v_2 \in \mathcal{V} \cup \mathcal{C}$. Assertions in $L_{\{=,\in\}}$ are interpreted by first classifying variables into many sorts: base variables that take values over $\mathcal{C}$, first-order set variables that take values over $2^{\mathcal{C}}$, etc.

Expressions and Assignments: Given a substitution that associates values with variables, expressions compute new values. For data-independent systems, we consider the constraint language $L_{\{=\}}$ and the set of expressions $\mathcal{E}_{\{=\}} = \mathcal{V} \cup \mathcal{C}$. When considering the constraint language $L_{\{=,\in\}}$, we will use a richer expression language $\mathcal{E}_{\{=,\in\}}$ which is the smallest set such that (i) $\mathcal{V} \cup \mathcal{C} \subseteq \mathcal{E}_{\{=,\in\}}$; (ii) $2^{\mathcal{V}} \cup 2^{\mathcal{C}} \subseteq \mathcal{E}_{\{=,\in\}}$; and (iii) if $e_1, e_2 \in \mathcal{E}_{\{=,\in\}}$ then $e_1 \cup e_2$, $e_1 \cap e_2$, $e_1 - e_2$

are all in $\mathcal{E}_{\{=,\in\}}$. The value of an expression e with respect to a substitution σ is denoted by $e\sigma$.

Assignments are written as $x := e$ where $x \in \mathcal{V}$ and $e \in \mathcal{E}$. The *meaning* of an assignment $x := e$ can be captured as a binary relation between substitutions such that $(\sigma, \sigma') \in [\![x := e]\!]$ iff $\sigma' = \sigma[e\sigma/x]$. Simultaneous assignments are denoted by $\bar{V} := \bar{E}$ where $\bar{V}$ and $\bar{E}$ are (equal-length) sequences of variables and expressions respectively. An assignment can also be seen as transforming assertions: from one that is satisfied before the assignment to another that is satisfied after the assignment. This assertion mapping corresponding to an assignment ρ is denoted by Ξ_ρ.

Standard Forms and Equivalence: We say that two assertions are identical if they differ only in the names of bound variables. Two assertions are *equivalent* if and only if they are satisfied by the same set of substitutions. Note that while identical assertions will be equivalent, the converse does not always hold: e.g. $\varphi_1 = (x = y \wedge y = z)$ and $\varphi_2 = (x = y \wedge x = z)$ are equivalent but not identical.

When processing assertions, it is often useful to reduce them to equivalent *standard form*, defined as follows:

Definition 1 [Standard Form of Assertions] An assertion $\varphi \in L_{\{=\}}$ is said to be in standard form if the following hold:

- *Structure:* φ is in disjunctive normal form, i.e., is of the form $\varphi_1 \vee \varphi_2 \cdots \varphi_n$ such that each φ_i itself is of the form $\exists V. \varphi_{i,1} \wedge \varphi_{i,2} \wedge \cdots \wedge \varphi_{i,k_i}$ where $\varphi_{i,j} \in PC_{\{=\}}$; the assertions φ_i are called the conjuncts of φ.
- *Non-redundancy:* A primitive constraint occurs at most once in any conjunct in φ.
- *Naming:* $bv(\varphi) \cap fv(\varphi) = \{\}$;
- *Order:* For each conjunct φ_i in φ, if $x = y$ occurs in φ_i then there are no primitive constraints of the form $y = z$, $z = y$, $y \neq z$ or $z \neq y$ in φ_i for any variable z.

□

Given a conjunction φ of primitive constraints over $PC_{\{=\}}$, it is easy to see that we can group its variables in several equivalence classes ($x = y$ in φ means that x and y belong to the same class). Moreover, φ is satisfiable (when interpreted over an infinite data domain) if only if whenever $x \neq y$ occurs in φ, x and y belong to different classes. These observations immediately yield a procedure to convert every assertion into an equivalent standard form.

Proposition 1 *For every satisfiable assertion $\varphi \in L_{\{=\}}$ there is an equivalent assertion φ' such that φ' is in standard form.*

An assertion φ in standard form is said to be quantifier-free if it has no bound variables. For any conjunct $\exists V.\ \varphi$ let $\exists V.\ \varphi'$ be an equivalent standard form such

that for every $x = y$ in φ', $x \in V$ implies $y \in V$. Let φ'' be the assertion obtained by dropping from φ' every primitive constraint that contains a variable in V. It can then be shown that φ'' is equivalent to $\exists V \varphi$ whenever the assertions are interpreted over an infinite domain of constants.

Example 1 Consider $\varphi : \exists y.x = y \wedge y \neq z \wedge z \neq w \wedge y \neq w$.

- The assertion $\varphi_1 : \exists y.x = y \wedge x \neq z \wedge z \neq w \wedge x \neq w$ is in standard form, and is equivalent to φ.
- Shrinking the scope of the quantifier yields the assertion $\varphi_2 : (\exists y.x = y) \wedge x \neq z \wedge z \neq w \wedge x \neq w$ which is equivalent to φ_1 and hence φ.
- Since $(\exists y.x = y)$ is satisfiable (under interpretation over any domain), the assertion $\varphi_3 : x \neq z \wedge z \neq w \wedge x \neq w$, and φ_2 are equivalent.

∎

In the above example, the assertion $(\exists y.\ x = y)$ is always satisfiable and hence the final quantifier-free form φ_3 is always equivalent to the initial assertion φ. In general, however, the dropped assertion may be satisfiable only over domains that are sufficiently large. For instance, $\exists y, z.\ x \neq y \wedge x \neq z \wedge y \neq z$ is satisfiable only when the data domain has at least two elements. However, the dropped assertions are always satisfiable when interpreted over an infinite data domain. Hence we have:

Proposition 2 *For every satisfiable assertion $\varphi \in L_{\{=\}}$ there is assertion φ' such that φ and φ' are equivalent over infinite data domains and φ' is in quantifier-free standard form.*

Quantifier-free standard forms are important since they allow us to finitely represent all possible assertions over a finite set of free variables.

Proposition 3 *Let V be a set of variables and $\Phi \subseteq L_{\{=\}}$ be the set of all assertions in quantifier-free standard form such that $fv(\varphi) = V$ for all $\varphi \in \Phi$. Then the set Φ is finite.*

3 Extended Finite Automata

We now describe an automata-based model to specify infinite-state systems with finite number of control locations. The automaton's behavior can be observed based on the labels, called *actions*, on the transitions taken by the automaton. We distinguish between *input actions* (denoted by $c?x$), *output actions* (denoted by $c!e$) and *internal actions* (denoted by special symbol τ) where $c \in \mathcal{C}$, $x \in \mathcal{V}$, and $e \in \mathcal{V} \cup \mathcal{C}$. The set of actions is denoted as Act; we use α to range over actions. We refer to the sets of free and bound variables involved in an action α as $fv(\alpha)$ and $bv(\alpha)$, respectively. These sets are defined as follows: $fv(c?x) = \emptyset$, $bv(c?x) = \{x\}$, $fv(c!e) = vars(e)$, $bv(c!e) = \emptyset$.

Definition 2 [Extended Finite Automaton (EFA)] An *extended finite automaton* over the constraint language $L_{\{=\}}$ is defined by the sextuple $\mathcal{A} = \langle \mathcal{L}, \delta, \iota, \ell_0, \varrho, \mathcal{F} \rangle$ where:

- $\mathcal{L}$ is a finite set of (control) *locations*;
- $\delta = \mathcal{L} \times 2^{\mathcal{V}}$, the variable map, is a function that maps each location ℓ_i to a finite set of variables local to ℓ_i;
- ι is a function that maps each location to an assertion (invariant) such that $\iota(\ell_i) \in L_{\{=\}}$ and $\iota(\ell_i)$ must be satisfied by $\delta(\ell_i)$ whenever ℓ_i is reached;
- $\ell_0 \in \mathcal{L}$ is the *initial location*;
- ϱ is the *transition relation* such that for all $(\ell_i, \ell_j, \langle \gamma, \alpha, \rho \rangle) \in \varrho$,
  - $\ell_i, \ell_j \in \mathcal{L}$ are the source and destination locations of the transition, respectively
  - $\langle \gamma, \alpha, \rho \rangle$ is the label on the transition consisting of:
    - $\gamma \in L_{\{=\}}$, the *enabling condition*: an assertion over $\delta(\ell_i)$ which specifies the condition under which the transition may be taken
    - $\alpha \in Act$, the *action* associated with the transition, such that $bv(\alpha) \cap \delta(\ell_i) = \emptyset$
    - ρ, the *update relation*: a set of simultaneous assignments defining the values assumed by the variables $\delta(\ell_j)$ of the destination location in terms of values of variables $\delta(\ell_i)$ in the source location;
- $\mathcal{F} \subseteq \mathcal{L}$, is the set of *final locations*.

□

Example 2 The 2-place FIFO buffer shown in Figure 1 is formally represented as the EFA $\mathcal{S} = \langle \mathcal{L}, \delta, \iota, \ell_0, \varrho, \mathcal{F} \rangle$ where $\mathcal{L} = \{s_0, s_1, s_2\}$, the variable map $\delta(s_0) = \emptyset$, $\delta(s_1) = \{x\}$ and $\delta(s_2) = \{x, y\}$; the invariants $\iota(s_0) = \iota(s_1) = \iota(s_2) = \mathit{true}$, the initial location $\ell_0 = s_0$, the transition relation is defined as $\varrho = \{(s_0, s_1, \langle \mathit{true}, \mathit{in}?x, \{\} \rangle), (s_1, s_0, \langle \mathit{true}, \mathit{out}!x, \{\} \rangle), (s_1, s_2, \langle \mathit{true}, \mathit{in}?y, \{x := x\} \rangle), (s_2, s_1, \langle \mathit{true}, \mathit{out}!x, \{x := y\} \rangle)\}$, and the set of final locations $\mathcal{F} = \emptyset$. ∎

Behaviors of an EFA: Note that an EFA is analogous to a program: control locations correspond to program counter values (program points) and the local variables correspond to the data variables live at each program point. We call $q = \langle \ell, \theta \rangle$ a *concrete* state of an automaton if ℓ is a location and θ is a ground substitution of variables in $\delta(\ell)$ defined over a data domain D. We can define behaviors of an EFA with respect to specific valuations of its variables, as follows.

Definition 3 [Concrete run of an EFA] A *concrete run* ω_D of $\mathcal{A}$ is a (possibly infinite) sequence of alternating concrete states and actions $\langle \ell_0, \theta_0 \rangle \; \alpha_0 \; \langle \ell_1, \theta_1 \rangle \; \alpha_1 \ldots$ such that:

- ℓ_0 is the initial location of $\mathcal{A}$ and θ_0 is a ground substitution of variables in $\delta(\ell_0)$ to D such that $\theta_0 \models \iota(\ell_0)$
- for all i $(\ell_i, \ell_{i+1}, \langle \gamma, \alpha, \rho \rangle) \in \varrho$ such that:
  - *Transition is enabled:* $\theta_i \models \gamma$

```
reach(A, Ls,Ss, Ld,Sd) :-
   gtrans(A, Ls,Ss, Act, Ld,Sd).
reach(A, Ls,Ss, Ld,Sd) :-
   gtrans(A, Ls,Ss, Lm,Sm),
   reach(A, Lm,Sm, Ld,Sd).
```

```
gtrans(A, Ls,Ss, Ld,Sd) :-
   inv(A, Ls,Ss),
   trans(A, Ls,Ss, Act, Ld,Sd),
   ground(Act),
   inv(A, Ld,Sd).
```

Figure 3. Relation describing the reachability of concrete states in an EFA

- *Input Value is bound:* θ'_i is a ground extension of θ_i to $vars(\alpha)$ such that $\alpha_i = \alpha[\theta'_i]$
- *Data is transferred from source to destination:* $(\theta_i, \theta''_i) \in [\![\rho]\!]$
- *Input value is transferred:* $\theta'_{i+1} = \theta''_i \circ \sigma$ where σ is such that $\theta'_i = \theta_i \circ \sigma$, and
- *Destination invariant holds:* $\theta_{i+1} \models \iota(\ell_{i+1})$.

□

EFA as a Logic Program: An EFA can be readily represented as a set of Prolog rules. The following relations specify an EFA A:

- `init(`A`, L)`: a relation with a single tuple, specifying the initial location `L`.
- `inv(`A`, L,V)`: a relation specifying the invariants at each location.
- `trans(`A`, Ls,Vs, Act, Ld,Vd)`: a relation defining transitions from source location `Ls`, with the list `Vs` representing `Ls`'s variables, to destination location `Ld`, with variables `Vd`. `Act` denotes the action taken by the automaton. The body of each rule corresponds to transition's enabling condition γ (i.e. facts imply $\gamma = true$). Finally, the update relation is specified either by unifying corresponding variables in `Vs` and `Vd` or using additional predicates in the body of the rule.
- `final(`A`, L)`: a relation specifying the final locations.

Example 3 The EFA $\mathcal{S} = \langle \mathcal{L}, \delta, \iota, \ell_0, \varrho, \mathcal{F} \rangle$ from Example 2 can be represented as the following logic program:

```
init(S, s0).

inv(S, s0,[]).
inv(S, s1,[X]).
inv(S, s2,[X,Y]).
```

```
trans(S, s0,[], in(X), s1,[X]).
trans(S, s1,[X], out(X), s0,[]).
trans(S, s1,[X], in(Y), s2,[X,Y]).
trans(S, s2,[X,Y], out(X), s1,[Y]).
```

∎

The reachability of a concrete state of an EFA can be computed using the transitive closure relation over the transitions of `A` shown in Figure 3. In the figure, the relation `gtrans` nondeterministically selects an applicable transition and binds any variables in its action (i.e. if `Act` is an input action, binds the input

variable to some value in the data domain), and ensures that the invariants at the source and destination states hold. The set of all concrete states of an automaton `A` that are reachable from a given concrete state $\langle \ell, \theta \rangle$ can be computed as answers to the query `reach(A,` ℓ`,`$\delta(\ell)\theta$`, Ld,Sd)` over the concrete reachability program. It can be easily shown that evaluating the above query using resolution is stepwise equivalent to computing concrete runs using Definition 3.

A run that reaches a concrete state can be easily computed based on the resolution steps needed to establish the reachability of the state using the above program (e.g. using the notion of *justification* of a logic programming proof [23]).

Abstracting the Behaviors of an EFA: To ensure that behaviors of an EFA can be analyzed even when it has an infinite number of concrete states, we use an alternative representation of the behaviors. For this, we introduce the notion of an abstract state: a pair $\langle \ell, \varphi \rangle$ where ℓ is a control location and $\varphi \in L_{\{=\}}$ is an assertion (representing constraints on the valuations of the local variables at ℓ) such that $fv(\varphi) \subseteq \delta(\ell)$.

Definition 4 [Abstract run of an EFA] An *abstract run* ω of $\mathcal{A}$ is a (possibly infinite) sequence of alternating abstract states and actions $\langle \ell_0, \varphi_0 \rangle\ \alpha_0\ \langle \ell_1, \varphi_1 \rangle\ \alpha_1 \ldots$ such that:

- ℓ_0 is the initial location of $\mathcal{A}$ and $\varphi_0 = \iota(\ell_0)$.
- for all i $(\ell_i, \ell_{i+1}, \langle \gamma, \alpha, \rho \rangle) \in \varrho$ such that:
  - *Transition is enabled:* $\varphi_i \wedge \gamma$ is satisfiable
  - *Constraint is transferred to destination:* $\varphi'_i = \Xi_\rho(\varphi_i)$ and $\varphi_{i+1} = (\exists V \varphi'_i) \wedge \iota(\ell_{i+1})$ where $V = vars(\varphi'_i) - \delta(\ell_{i+1})$

□

An abstract state $q = \langle \ell, \varphi \rangle$ of an EFA corresponds to a (possibly infinite) set S of concrete states over value domain D such that $\forall \langle \ell, \theta \rangle \in S.\theta \models \varphi$; in other words, φ cannot distinguish between the valuations of the concrete states. We say that each element $\langle \ell, \theta \rangle$ of S is a *concretization* of q, and q is an *abstraction* of S.

Given the relationship between abstract and concrete states, we can construct an abstract run from a concrete run and vice versa. The close correspondence between abstract and concrete states is formalized by the following theorem:

Theorem 4 *A concrete state $\langle \ell_n, \theta_n \rangle$ is reachable in a concrete run ω_D of an extended finite automaton, iff there exists an abstract state $\langle \ell_n, \varphi_n \rangle$ which is reachable in an abstract run ω such that $\theta_n \models_D \varphi_n$.*

Finiteness: Two abstract states $\langle \ell, \varphi \rangle$, $\langle \ell', \varphi' \rangle$ are equivalent iff $\ell = \ell'$ and φ and φ' are equivalent. Since all assertions at location ℓ can be written in quantifier-free standard form with free variables from $\delta(\ell)$, from Proposition 3 we know that there are only finite number of abstract states involving ℓ. This immediately leads to the following result:

```
reach(A, Ls,Ss, Ld,Sd) :-
   atrans(A, Ls,Ss, Act, Ld,Sd).
reach(A, Ls,Ss, Ld,Sd) :-
   atrans(A, Ls,Ss, Lm,Sm),
   reach(A, Lm,Sm, Ld,Sd).
```

```
atrans(A, Ls,Ss, Ld,Sd) :-
   inv(A, Ls,Ss),
   trans(A, Ls,Ss, Act, Ld,Sd),
   inv(A, Ld,Sd).
```

Figure 4. Abstract reachability relation for EFAs

Proposition 5 *Reachability of any abstract state of an EFA is decidable.*

From Theorem 4 and Proposition 5 we have:

Corollary 6 *Reachability of a concrete state of an EFA is decidable.*

Data Domain Size: Note that the finiteness results above used quantifier-free standard forms for assertions, and hence are valid when we interpret EFAs over infinite data domains. These results also carry over to finite domains that are "large enough". A domain size above which the results always hold can be estimated as follows. Consider all the quantifier elimination steps applied while computing an abstract run. At each step, say to eliminate the quantifier in $\exists V \varphi$, let φ be in standard form, and let N_D be the number of variables of V in disequality constraints. Then the quantifier-free form is equivalent to the original assertion for all domains of size N_D or greater. Thus the above correctness results hold for domains of size N or greater, where N is the largest N_D among all quantifier elimination steps used in computing the run.

Query Evaluation for Abstract State Reachability: Reachability of abstract states can be computed using the reachability relation shown in Figure 4. The relation `atrans` in the figure selects an applicable transition and ensures that the invariants at source and destination states hold. However, in order to ensure that query evaluation using resolution w.r.t. the abstract reachability program is equivalent to computing abstract runs using Definition 4, we need to first augment the evaluation mechanism with constraint solving. Traditional logic programming systems resolve queries by keeping track of substitutions. When a subgoal such as $x \neq y$ is encountered, these mechanisms will fail if x and y are not already bound to specific values in the data domain. Constraint Logic Programming (CLP)[11] provides a very expressive framework to resolve such queries by generalizing substitutions to assertions in a constraint language. We can check for reachability of abstract states by resolving queries w.r.t. the reachability program in a CLP system that handles constraints over $L_{\{=\}}$. Tabled resolution [24] can be used to ensure the termination of query evaluation. We built such a query evaluation system as a tabled constraint meta-interpreter that handles constraints over $L_{\{=\}}$.

Verification using EFAs

When an EFA is interpreted as an automaton over finite words, a (concrete/abstract) run of an EFA is said to be accepting if the run is finite and ends in a final state. For EFAs that model systems to be verified, we are typically interested in all possible runs. Hence all locations in an EFA representing system models will be final locations. Such automata can be seen as equivalent to Symbolic Transition Systems (STS [3]) and generalizations of Symbolic Transition Graphs (STGs [9]) and STGs with Assignments (STGAs [14]).

Property Specification using EFAs: Liveness properties and negations of safety properties can be simply encoded as EFAs.

Example 4 Consider the "ordered message delivery" property which states that for any two messages x, y such that x is read before y, x will be written out before y. Note that this is a safety property and hence has to hold throughout a run. The negation of this property, called "out-of-order delivery" is expressed by the nondeterministic EFA in Figure 2. Note that "out-of-order delivery" is a liveness property that is satisfied on a run if it is satisfied at some point in the run. Liveness properties can be simply verified by checking for reachability of final states. ∎

When an EFA is interpreted as an automaton over finite words, a (concrete/abstract) run of an EFA is said to be accepting if it is finite and ends in a final state. Using this interpretation, we can verify safety and liveness properties of EFA models. We can easily expand this framework to verify linear-time properties with data values, by using the Büchi acceptance condition: a run is accepting only if it is infinite and visits a final state infinitely often. We call EFAs with Büchi acceptance condition as constraint Büchi automata (CBA). The problem of determining whether a CBA has an accepting abstract/concrete run is decidable, since the reachability of abstract/concrete states is decidable.

The definition of CBA is apparently similar to Pnueli's Büchi Automaton with Data (BAD) [19] but differs mainly by treating data variables as local to a control location. Moreover, a CBA allows finite number of data variables to be introduced (new or temporary locals) into states during system execution. Data variables in a CBA have the following properties:

- they may be generated in the states or introduced by the transitions: for any $\ell_i, \ell_j \in \mathcal{L}$ such that $(\ell_i, \ell_j, \langle \gamma, \alpha, \rho \rangle) \in \varrho$, $\delta(\ell_j) \subseteq \delta(\ell_i) \cup vars(\alpha)$;
- whenever a variable x is introduced to ℓ, it overwrites the value of x previously assigned to ℓ; another way to say this is that the interpretation of x is different upon every visit to a location containing x (this in particular applies to self loops);
- initial location ℓ^0 may contain a non-empty, finite set of variables.

```
init((A1,A2), (L1,L2)) :-
  init(A1, L1), init(A2, L2).

inv((A1,A2), (L1,L2), V) :-
   inv(A1, L1,V1), inv(A2, L2,V2),
   append(V1,V2,V).

trans((A1,A2), (Ls1,Ls2),Vs, Act, (Ld1,Ld2),Vd) :-
   trans(A1, Ls1,Vs1, Act, Ld1,Vd1),
   trans(A2, Ls2,Vs2, Act, Ld2,Vd2),
   append(Vs1,Vs2,Vs), append(Vd1,Vd2,Vd).

final((A1,A2), (L1,L2)) :-
  final(A1, L1), final(A2, L2).
```

Figure 5. Relations describing the product of two EFAs

Product Construction: Automata-based model checkers pose the verification problem in terms of checking whether the intersection of two automata's languages is empty. Critical to this formulation is the construction of a product automaton $\mathcal{A} = (\mathcal{A}_1 \times \mathcal{A}_2)$ whose language corresponds to the intersection of the languages of $\mathcal{A}_1$ and $\mathcal{A}_2$. We can construct the product of two EFAs such that the result is also an EFA. In fact, given two EFAs $\mathtt{A_1}$ and $\mathtt{A_2}$ represented as logic programs, the product EFA $(\mathtt{A_1}, \mathtt{A_2})$ can be computed using a logic program given in Figure 5. Each location in the product automaton is defined as a pair $(\mathtt{L_1}, \mathtt{L_2})$ consisting of the locations of the component automata. The non-trivial part of the encoding is the handling of action labels: the label on a transition in the product automaton is obtained by unifying the action labels the two component automata. It is easy to show that two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ have a common run if and only if their product has a run.

4 Example: Verifying a Write-Back Cache

Below we describe an EFA model of a memory system with a write-back cache. We use this model to verify that a memory read at an arbitrary but specific address retrieves the value previously written to that address. The model captures the behavior of a memory system with potentially infinite memory addresses and an infinite domain of data values. We first build a model for a system with a single-line cache: exactly one address-value pair is stored in the cache. We generalize this model to cache with an arbitrary size in Section 5.

The data state of a single-line cache is denoted as a triple `(CA,CV,CD)` representing the address `CA` in the cache, a current data value `CV` at that address, and a "dirty bit" `CD` that is 1 if the value in the cache has been modified (and hence possibly different from the value in the main memory) and 0 otherwise. The cache services read and write requests received from the processor.

```
% receive request to write value V to address A
trans(cm, c0,[CA,CV,CD, MA,MV], write?(A,V), c1,[CA,CV,CD, MA,MV, A,V]).

% A is in cache: update cache value to V and set dirty bit to 1
trans(cm, c1,[CA,CV,CD, MA,MV, A,V], tau, c0,[CA,V,1, MA,MV]) :-
   A = CA.

% A is not in cache, but either cache value has not been modified,
% or cache address is different from that in the memory:
% replace cache contents with the tuple (A,V,1)
trans(cm, c1,[CA,CV,CD, MA,MV, A,V], tau, c0,[CA,V,1, MA,MV]) :-
   A ≠ CA, (CD ≠ 1; CA ≠ MA).

% A is not in cache, cache value has been modified,
% and cache address is the same as the address in the memory:
% update value in the memory to current cache value
trans(cm, c1,[CA,CV,CD, MA,MV, A,V], tau, c2,[CA,CV,CD, MA,CV, A,V]) :-
   A ≠ CA, CD = 1, CA = MA.

% write new tuple, with dirty bit set to 1, to the cache
trans(cm, c2,[CA,CV,CD, MA,MV, A,V], tau, c0,[A,V,1, MA,MV]).
```

Figure 6. Transition rules for handling writes in the cache-memory system

Upon a request `write?(A,V)` to write value `V` to address `A`, if `A` is in the cache (i.e. `A = CA`), then the value in the cache is replaced with `V` (i.e. `CV := V`) and `CD` is set to 1. Otherwise, the current cache entry is flushed to memory if `CD` is 1, and the data state of the cache is set to `(A,V,1)`. Read requests are processed similarly (flushing the current cache contents to memory on a cache miss); details are omitted.

Note that we are interested in verifying whether a read to a specific memory address returns the previously written value. This enables us to model a memory of arbitrary capacity by a single memory cell with a distinguished address. The data state of the memory is represented by a tuple `(MA,MV)`. The memory responds to read and write requests from the cache. A write to `(A,V)` changes the data state to `(MA,V)` if `A = MA`, and leaves the data state unchanged otherwise. A read request to address `A` returns `MV` if `A = MA`; otherwise the read returns an arbitrary value.

The cache-memory system can be readily modeled as an EFA; Figure 6 shows the a fragment of a logic program that represents the transition relation of the EFA model (more specifically, the portion of the relation that pertains to write requests). Note that all locations in the EFA for the system model are final locations.

The correctness condition for the cache-memory system is that after a read from an arbitrary address A will return V where V is most recent value written

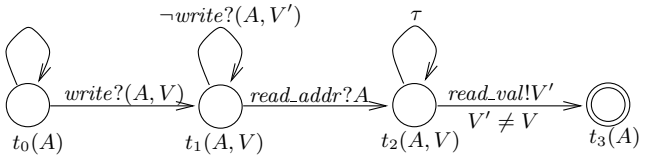

Figure 7. EFA for the correctness condition of cache-memory system

to address A. We represent the negation of this safety property by the EFA in Figure 7.

To verify whether the cache-memory system observes the correctness condition, we check whether a final location in the product of the system EFA and the property EFA is reachable. This check is done using the query

`reach( (cm,p), (c0,t0),[CA,CV,CD, MA,MV, PA], (_,t3),_), MA=PA.`

over a program consisting of the abstract reachability relation (Figure 4), the product construction relations (Figure 5) and the relations representing the system and property EFAs. In the query, `PA` is the address of interest to the property. Note that the unification `MA=PA` in the query ensures that the address of interest to the property is same as the address maintained by the memory. From the results of Section 3, we know that the above query evaluation will terminate and hence we can verify the given correctness property of the cache-memory model.

Comparison with Other Work: It should be noted that the verification of the cache-memory system as described above is not possible with any of the methods of [25,12,10,13,15]. Specifically, the definition of data independence in [25,12] does not admit any comparisons between data objects, so neither of them can handle a problem that requires equality tests.

The cache can be modeled in the scalarset-based approach of [10] as an array storing data values and indexed by the memory addresses. However, since they are used as array indices, the domain of memory addresses themselves cannot be reduced to a small, finite set of elements required for automatic verification.

The method of [13] is applicable to the problem only after a series of manual transformations of its specification that reduce memory addresses and data values to range over finite domains.

Finally, the system as specified above cannot be directly encoded as a symbolic transition graph with assignments [15], as STGAs require all variables in output transitions from a state to be present in the source state; note that the behavior of a memory cell upon receiving a read request needs the ability to output arbitrary values.

5 EFAs: Beyond Equalities

EFAs have been defined only using the constraint language $L_{\{=\}}$. This choice turns out to be crucial in being able to accurately verify systems and properties

specified as EFAs. However, this choice also restricts the class of systems that can be modeled using DFAs. Below we describe the consequences of augmenting EFAs with richer constraint languages such as $L_{\{=,\in\}}$.

Computational Aspects of using Assertions over $L_{\{=,\in\}}$: The ability to finitely represent the potentially infinite set of assertions using quantifier-free standard forms in $L_{\{=\}}$ is a key factor that makes the verification problem of EFAs decidable. Assertions over $L_{\{=,\in\}}$, in contrast, do not have equivalent quantifier-free representations in general. For example, consider the assertion $\varphi = \exists x.\ x \neq y \wedge x \in z \wedge y \in z$ which states that $y \in z$ and there is another element distinct from y that is also in z. This assertion cannot be expressed using only variables y and z without bound variables. The assertion $y \in z$ is an approximation of φ but fails to capture the fact that z has at least two elements.

This example illustrates that one can maintain counts (as number of elements in a set) in $L_{\{=,\in\}}$ and hence there is no finite representation of assertions in $L_{\{=,\in\}}$. For instance, consider the assignment $z := z \cup \{x\}$ evaluated under the constraint $x \notin z$. This assignment increases the cardinality of the set represented by z and hence simulates counting.

A classic approach to deal with problems due to counts is to approximate the counts: for instance, a widely-used approach is to maintain counts using the finite domain $\{0, 1, \mathit{many}\}$. This abstraction corresponds to representing assertions in $L_{\{=,\in\}}$ using at most one bound variable. We call assertions with a fixed ceiling on the number of bound variables as assertions in limited quantifier form. We can represent every $\varphi \in L_{\{=,\in\}}$ by an assertion $\varphi^\flat$ in limited quantifier form such that $\varphi \implies \varphi^\flat$ (i.e. by "relaxing" the meaning of the assertion). The direction of the approximation ensures that for every concrete run in an EFA there is an abstract run, but converse may not hold. Consequently, identifying an accepting abstract run during verification simply means that the property "may" hold; conversely, failure to find an an accepting abstract run means that the property "definitely" does not hold.

Extended Data Types: As the primitive constraints of $L_{\{=\}}$ involve only variables and constants, all data elements we have been considering so far are taken from $\mathcal{V} \cup \mathcal{C}$. When modeling systems with membership constraints and sets, it is useful to consider non-recursive compound terms (e.g. tuples) to represent data records. This improves the expressiveness without unduly complicating its formal framework. Therefore, in the following, we assume that constraints in $L_{\{=,\in\}}$ are built over variables, constants, and shallow, non-recursive compound terms. Such structures are used in both examples presented below.

Application: Mutli-line Cache: We augment the EFA model of cache-memory system (Section 4) to handle cache with an arbitrary number of cache lines. The contents of the cache are now represented as a set `Cs` of tuples $(\texttt{CA}, \texttt{CV}, \texttt{CD})$ each corresponding to one cache line, where `CA`, `CV` and `CD` are address, value

and dirty bit of that cache line. There are several modifications necessary for the transition rules to accommodate the extended specification. Checking for a presence of a tuple in the cache will now use membership constraints rather than equality. Upon cache miss, the line to be flushed to memory is chosen from the set nondeterministically (again using membership constraints). Finally, updating value in the cache upon a write hit requires first locating the appropriate cache line (using a membership constraint), and then updating its data value and dirty bit (using set difference and union operations, and equality constraints).

6 Example: Detecting Vulnerabilities in Computer Systems

A computer system consists of concurrent, interacting processes and services and users. Unexpected interactions between these entities often lead to subtle vulnerabilities that can be exploited to compromise system security. For example, `comsat` is a mail notification program, which prints the initial lines of incoming mails on a user's terminal. It obtains the user's terminal information from a system file `/etc/utmp` (terminal information is stored as records in this file). Misconfiguration of this file may permit any system user to obtain root privileges, as follows. If records in `/etc/utmp` can be changed by a user, then an attacker can replace their terminal in the file with '`/etc/passwd`'. The attacker can then send mail to self, thereby overwriting the password file. By choosing the mail message appropriately, the user can obtain root privileges.

In [21] we presented a model of this system in a value-passing process algebra with four processes: a user, the mailer service, `comsat`, and the file system `fs`. The first three processes interact via the file system. The model of the file system, thus, is central and most interesting. There are several distinct infinite-domain data types involved in these model: names of files and users, contents of files, etc. Some of the components of the system, such as the mailer and `comsat` are data-independent (in the type of message contents) in the sense of [25]. Similarly, the file system's control behavior is independent of the contents of the files. Below we describe an EFA model of the file system using assertions from $L_{\{=,\in\}}$.

Figure 8 shows a logic program encoding the transition relation of an EFA model of the file system. In the model, the variable FS holds the current state of the file system — the files and their contents. Its value is represented as a set of triples $(FN, FPerm, FCont)$, each triple expressing the state of a particular file with name FN, permissions $FPerm$, and contents $FCont$. File's permissions, in turn, are given by a set of pairs (U, P), denoting that user U has permission P ($P \in \{w, r\}$). The contents of a file are defined as a set of data records.

The program in Figure 8 uses several predicates representing the following assertions:

– `exists`(FS, N) checks for existence of file named N in the file system FS:

$$\exists\, P, C\,.\,(N, P, C) \in FS$$

```
trans(fs, s0,[FS], read?(U,N), s1,[FS,U,N]).
trans(fs, s1,[FS,U,N], tau, s0,[FS]) :-
   not(exists(FS,N), access(FS,U,N,r)).
trans(fs, s1,[FS,U,N], tau, s2,[FS,U,R]) :-
   exists(FS,N), access(FS,U,N,r), r_rec(FS,U,N,R).
trans(fs, s2,[FS,U,R], read_return!(U,R), s0,[FS]).

trans(fs, s0,[FS], write?(U,N,D), s3,[FS,U,N,D]).
trans(fs, s3,[FS,U,N,D], tau, s0,[FS]) :-
   not(access(FS,U,N,w)).
trans(fs, s3,[FS,U,N,D], tau, s0,[FS']) :-
   access(FS,U,N,w), add_rec(FS,U,N,D,FS').
```

Figure 8. Transition relation for an EFA model of a file system.

– `access`(FS, U, N, T) verifies that user U has access of type T to file named N in the file system:

$$\exists\, P, C\,.\,(N, P, C) \in FS \;\wedge\; (U, T) \in P$$

– `r_rec`(FS, U, N, R) extracts record R from file named N in the file system:

$$\exists\, P, C\,.\,(N, P, C) \in FS \;\wedge\; (U, r) \in P \wedge\; R \in C$$

– `add_rec`(FS, U, N, D, FS') adds record D to file named N in the file system FS, giving the modified file system in FS':

$$\exists\, P, C, C'\,.\,(N, P, C) \in FS \wedge (U, w) \in P \;\wedge\; C' = C \cup \{D\} \\ \wedge\; FS' = FS - \{(N, P, C)\} \cup \{(N, P, C')\}$$

One of the simplest safety properties expected to hold in a model of a computer system is that there are no unauthorized `write`s to `/etc/passwd`. We begin by checking a straightforward reachability property: whether there are any `write`s to `/etc/passwd` in the above EFA model. The analysis shows that `write`s to `/etc/passwd` are indeed possible, but many of the runs that are witnesses to this property show "normal" behavior that does not reveal the vulnerability. For instance, one of the runs corresponds to root issuing an explicit `write` to `/etc/passwd`. Hence we refine the property to rule out expected runs (called the "intentions model" [21]). We then observe `write`s to `/etc/passwd` using a sequence of operations — overwriting `/etc/utmp` and then sending mail — that exploit the vulnerability described earlier.

Note that the EFA model uses assertions over $L_{\{=,\,\in\}}$ and hence abstract runs may not have corresponding concrete runs. Thus a detected vulnerability may not, in general, actually exist in the model. However, given an abstract run, we can estimate whether or not the abstract states represent approximations; for instance, loss of accuracy in manipulating assertions in $L_{\{=,\,\in\}}$ occurs when

we eliminate bound variables to derive assertions in limited quantifier form. We can therefore modify the order in which transitions are taken during reachability analysis, preferring transitions that involve no loss of information. Abstract runs composed solely of transitions whose effects are computed losslessly always have corresponding concrete runs. Using this heuristic, we can isolate vulnerabilities that exist in the model despite using an expressive constraint language. This heuristic is not limited to reachability analysis alone and can be readily extended to good cycle detection in CBA.

7 Related Work and Discussion

In this paper we presented an automata-based approach to the analysis of behavior of infinite-state systems. We used EFAs as a unified model for infinite-state systems and their properties. Our technique can be used to automatically verify properties of data-independent systems, and can be extended to analyze more general infinite-state systems as well.

Considerable research has been done on generating a finite-state, property-preserving abstraction of data-independent systems. The method of [25] relies on the user to identify a data-independent program and manually transform its specification. A similar method was suggested in [1] to verify the alternating bit protocol. An automatic abstraction is proposed in [12] where a special countable set of values, called *schematic names* are used to bind data variables. However, this method is applicable only to programs that do not have tests on the values of data variables. A similar set of "symbolic values" is used in [15] which gives an algorithm for model checking data-independent value-passing processes. Because of working on process variables rather than inventing an extra set of values, we can claim our approach more directly implementable than the one presented in this work. Data independence is considered as a form of symmetry in [10] where a method is given to reduce the size of the data domain. The reduction is automatic once the user identifies a data-independent program and specifically annotates data-independent types ("data scalarsets"). Moreover, the approach works only for safety properties.

In [13], algorithms for refinement checking among data-independent systems are developed. They are based on the notion of *threshold collections*: finite collections of data types such that if the refinement holds for each of these types substituted for the data domain of the processes, it holds for an arbitrary data domain. The threshold collections must be identified manually before applying the appropriate refinement algorithms. Finally, an automatic method to abstract a large class of systems including data-independent systems is developed in [17]. That method is analogous to predicate abstraction [8] and constructs a finite-state system by introducing Boolean variables for every predicate in the original system. It is shown that for data-independent systems the method will terminate, producing a finite Boolean program which simulates the original system (with respect to control behaviors).

Our technique is most closely related to the abstraction method of [17]: the constraints in our case correspond to the Boolean variables introduced by [17]. However, note that [17] abstracts only the system and preserves only control flow properties. Thus, given an arbitrary property φ of a data-independent system S, one must first construct a product system S' and pose the verification of φ in S in terms of an appropriate control flow property of S'. In contrast, our technique constructs the product space, performs the necessary abstraction, and verifies the appropriate property on the product system, all in one phase. Such "on demand" abstraction is especially advantageous when the properties can be proved without constructing the entire abstract state space of the system.

There have been several works on constraint-based model checking and the use of constraint (logic) programming for verification of infinite-state systems (e.g. [5,4,6]). We have developed verification techniques for infinite-state systems based on tabled resolution and constraint processing: for timed systems [7,18], systems with mobility [26], and for symbolic bisimulation of systems [3]; each of those techniques can be seen as a conservative extension to our finite state model checker, XMC [22]. This paper presents a model checker for data-independent systems using a similar approach, but uses an LTL-based logic to specify properties. We are currently investigating easy-to-implement tableau-based techniques to verify a general class of value passing systems (to model mobility as well as data independence). Our current work also includes modeling other security-related problems to further the application and development of infinite-state verification techniques.

References

1. S. Aggarwal, R.P. Kurshan, and K. Sabnani. A calculus for protocol specification and validation. *Protocol Specification, Testing and Verification, III*, 1983.
2. S. Basu, K. Narayan Kumar, L.R. Pokorny, and C.R. Ramakrishnan. Resource-constrained model checking of recursive programs. In *TACAS*, 2002.
3. S. Basu, M. Mukund, C.R. Ramakrishnan, I.V. Ramakrishnan, and R.M. Verma. Local and symbolic bisimulation using tabled constraint logic programming. In *ICLP*, 2001.
4. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *CAV*, 1997.
5. W. Chan, R.J. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *CAV*, 1997.
6. G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS*, 1999.
7. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *RTTS*, 2000.
8. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
9. M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.
10. C. Norris Ip and D. L. Dill. Better verification through symmetry. *FMSD*, 1996.
11. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL*, 1987.

12. B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and Computation*, 107(2), December 1993.
13. R. S. Lazić. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University, 1999.
14. H. Lin. Symbolic transition graphs with assignments. In *CONCUR*, 1996.
15. H. Lin. Model checking value-passing processes. In *APSEC*, 2001.
16. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.
17. K.S. Namjoshi and R.P. Kurshan. Syntactic program transformations for automatic abstractions. In *CAV*, 2000.
18. G. Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan. Efficient model checking of real time systems using tabled logic programming and constraints. In *ICLP*, 2002.
19. A. Pnueli, Y. Kesten, and M. Vardi. Yes, Matilda! Abstraction can Replace Deduction, even for Computational Models which are BAD (Buchi Automata with Data). In *VHS Meeting*, Grenoble, 1999.
20. L. Robert Pokorny and C. R. Ramakrishnan. Model checking linear temporal logic using tabled logic programming. In *TAPD*, 2000.
21. C. R. Ramakrishnan and R. Sekar. Model-based analysis of configuration vulnerabilities. *Journal of Computer Security (JCS)*, 10(1 / 2):189–209, 2002.
22. C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, et al. XMC: A logic-programming-based verification toolset. In *CAV*, 2000.
23. A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *PPDP*, 2000.
24. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *ICLP*, 1986.
25. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL*, 1986.
26. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A logical encoding of the pi-calculus: Model checking mobile processes using tabled resolution. In *VMCAI*, 2003.