Constructing Quantified Invariants via Predicate Abstraction [⋆]

Shuvendu K. Lahiri and Randal E. Bryant

Carnegie Mellon University, Pittsburgh, PA
shuvendu@ece.cmu.edu,Randy.Bryant@cs.cmu.edu

Abstract. Predicate abstraction provides a powerful tool for verifying properties of infinite-state systems using a combination of a decision procedure for a subset of first-order logic and symbolic methods originally developed for finite-state model checking. We consider models where the system state contains mutable function and predicate state variables. Such a model can describe systems containing arbitrarily large memories, buffers, and arrays of identical processes. We describe a form of predicate abstraction that constructs a formula over a set of universally quantified variables to describe invariant properties of the function state variables. We provide a formal justification of the soundness of our approach and describe how it has been used to verify several hardware and software designs, including a directory-based cache coherence protocol with unbounded FIFO channels.

1 Introduction

Graf and Saïdi introduced *predicate abstraction* [10] as a means of automatically determining invariant properties of infinite-state systems. With this approach, the user provides a set of k Boolean formulas describing possible properties of the system state. These predicates are used to generate a finite state abstraction (containing at most 2^k states) of the system. By performing a reachability analysis of this finite-state model, a predicate abstraction tool can generate the strongest possible invariant for the system expressible in terms of this set of predicates. Prior implementations of predicate abstraction [10, 7, 1, 8] required making a large number of calls to a theorem prover or first-order decision procedure and hence could only be applied to cases where the number of predicates was small. More recently, we have shown that both BDD and SAT-based Boolean methods can be applied to perform the analysis efficiently [12].

In most formulations of predicate abstraction, the predicates contain no free variables; they evaluate to true or false for each system state. The abstraction function α has a simple form, mapping each *concrete* system state to a single *abstract* state based on the effect of evaluating the k predicates. The task of predicate abstraction is to construct a formula ψ^* consisting of some Boolean combination of the predicates such that $\psi^*(s)$ holds for every reachable system state s.

[⋆] This research was supported in part by the Semiconductor Research Corporation, Contract RID 1029.

To verify systems containing unbounded resources, such as buffers and memories of arbitrary size and systems with arbitrary number of identical, concurrent processes, the system model must support state variables that are functions or predicates [11, 4]. For example, a memory can be represented as a function mapping an address to the data stored at an address, while a buffer can be represented as a function mapping an integer index to the value stored at the specified buffer position. The state elements of a set of identical processes can be modeled as functions mapping an integer process identifier to the state element for the specified process. In many systems, this capability is restricted to *arrays* that can be altered only by writing to a single location [7, 11]. Our verifier allows a more general form of mutable function, where the updating operation is expressed using lambda notation.

In verifying systems with function state variables, we require quantified predicates to describe global properties of state variables, such as "At most one process is in its critical section," as expressed by the formula $\forall i, j : \texttt{crit}(i) \wedge \texttt{crit}(j) \Rightarrow i = j$. Conventional predicate abstraction restricts the scope of a quantifier to within an individual predicate. System invariants often involve complex formulas with widely scoped quantifiers. The scoping restriction implies that these invariants cannot be divided into small, simple predicates. This puts a heavy burden on the user to supply predicates that encode intricate sets of properties about the system. Recent work attempts to discover quantified predicates automatically [7], but it has not been successful for many of the systems that we consider.

In this paper we present an extension of predicate abstraction in which the user supplies predicates that include free variables from a set of *index* variables $\mathcal{X}$. The predicate abstraction engine constructs a formula ψ^* consisting of a Boolean combination of these predicates, such that the formula $\forall \mathcal{X} \psi^*(s)$ holds for every reachable system state s. With this method, the predicates can be very simple, with the predicate abstraction tool constructing complex, quantified invariant formulas. For example, the property that at most one process can be in its critical section could be derived by supplying predicates $\texttt{crit}(\texttt{i})$, $\texttt{crit}(\texttt{j})$, and $\texttt{i} = \texttt{j}$, where $\texttt{i}$ and $\texttt{j}$ are the index symbols. Encoding these predicates in the abstract system with Boolean variables $\texttt{ci}$, $\texttt{cj}$, and $\texttt{eij}$, respectively, we can verify this property by using predicate abstraction to prove that $\texttt{ci} \wedge \texttt{cj} \Rightarrow \texttt{eij}$ holds for every reachable state of the abstract system.

Flanagan and Qadeer use a method similar to ours [8] for constructing universally quantified loop invariants for sequential software, and we briefly described our method in an earlier paper [12]. Our contribution in this paper is to describe the method more carefully, explore its properties, and to provide a formal argument for its soundness. The key idea of our approach is to formulate the abstraction function α to map a concrete system state s to the set of all possible valuations of the predicates, considering the set of possible values for the index variables $\mathcal{X}$. The resulting abstract system is unusual; it is not characterized by a state transition relation and hence cannot be viewed as a state transition system. Nonetheless, it provides an abstract interpretation of the concrete system [6] that can be used to find invariant system properties.

Assuming a decision procedure that can determine the satisfiability of a formula with universal quantifiers, we can prove the following completeness result for our formulation: Predicate abstraction can prove any property that can be proved by induction on the state sequence using an induction hypothesis expressed as a universally quantified formula over the given set of predicates. For many modeling logics, this decision problem is undecidable. By using quantifier instantiation, we can implement a sound, but incomplete verifier.

As an extension, we show that it is easy to incorporate *axioms* into the system, properties that must hold universally for every system state. Axioms can be viewed simply as quantified predicates that must evaluate to true on every step. For brevity, this paper only sketches the main proofs. We conclude the paper by describing our use of predicate abstraction to verify several hardware and software systems, including a directory-based cache coherence protocol devised by Steven German [9]. We believe we are the first to verify the protocol for a system with an unbounded number of clients, each communicating via unbounded FIFO channels.

2 Preliminaries

We assume the concrete system is defined in terms of some decidable subset of first-order logic. Our implementation is based on the CLU logic [4], supporting expressions containing uninterpreted functions and predicates, equality and ordering tests, and addition by integer constants, but the ideas of this paper do not depend on the specific modeling formalism. For discussion, we assume that the logic supports Booleans, integers, functions mapping integers to integers, and predicates mapping integers to Booleans.

2.1 Notation

Rather than using the common *indexed vector* notation to represent collections of values (e.g., $\boldsymbol{v} \doteq \langle v_1, v_2, \ldots, v_n \rangle$), we use a *named set* notation. That is, for a set of symbols $\mathcal{A}$, we let $\mathbf{v}$ indicate a set consisting of a value $v_{\mathtt{x}}$ for each $\mathtt{x} \in \mathcal{A}$.

For a set of symbols $\mathcal{A}$, let $\sigma_{\mathcal{A}}$ denote an *interpretation* of these symbols, assigning to each symbol $\mathtt{x} \in \mathcal{A}$ a value $\sigma_{\mathcal{A}}(\mathtt{x})$ of the appropriate type (Boolean, integer, function, or predicate). Let $\Sigma_{\mathcal{A}}$ denote the set of all interpretations $\sigma_{\mathcal{A}}$ over the symbol set $\mathcal{A}$.

For interpretations $\sigma_{\mathcal{A}}$ and $\sigma_{\mathcal{B}}$ over disjoint symbol sets $\mathcal{A}$ and $\mathcal{B}$, let $\sigma_{\mathcal{A}} \cdot \sigma_{\mathcal{B}}$ denote an interpretation assigning either $\sigma_{\mathcal{A}}(\mathtt{x})$ or $\sigma_{\mathcal{B}}(\mathtt{x})$ to each symbol $\mathtt{x} \in \mathcal{A} \cup \mathcal{B}$, according to whether $\mathtt{x} \in \mathcal{A}$ or $\mathtt{x} \in \mathcal{B}$.

For symbol set $\mathcal{A}$, let $E(\mathcal{A})$ denote the set of all expressions in the logic over $\mathcal{A}$. For any expression $e \in E(\mathcal{A})$ and interpretation $\sigma_{\mathcal{A}} \in \Sigma_{\mathcal{A}}$, let the *valuation of e with respect to $\sigma_{\mathcal{A}}$*, denoted $\langle e \rangle_{\sigma_{\mathcal{A}}}$ be the (Boolean, integer, function, or predicate) value obtained by evaluating e when each symbol $\mathtt{x} \in \mathcal{A}$ is replaced by its interpretation $\sigma_{\mathcal{A}}(\mathtt{x})$.

Let $\mathbf{v}$ be a named set over symbols $\mathcal{A}$, consisting of expressions over symbol set $\mathcal{B}$. That is, $v_{\mathtt{x}} \in E(\mathcal{B})$ for each $\mathtt{x} \in \mathcal{A}$. Given an interpretation $\sigma_{\mathcal{B}}$ of the symbols in $\mathcal{B}$, evaluating the expressions in $\mathbf{v}$ defines an interpretation of the symbols in $\mathcal{A}$, which we denote $\langle \mathbf{v} \rangle_{\sigma_{\mathcal{B}}}$. That is, $\langle \mathbf{v} \rangle_{\sigma_{\mathcal{B}}}$ is an interpretation $\sigma_{\mathcal{A}}$ such that $\sigma_{\mathcal{A}}(\mathtt{x}) = \langle v_{\mathtt{x}} \rangle_{\sigma_{\mathcal{B}}}$ for each $\mathtt{x} \in \mathcal{A}$.

A *substitution* $\boldsymbol{\pi}$ for a set of symbols $\mathcal{A}$ is a named set of expressions over some set of symbols $\mathcal{B}$ (with no restriction on the relation between $\mathcal{A}$ and $\mathcal{B}$.) That is, for each $\mathtt{x} \in \mathcal{A}$, there is an expression $\pi_{\mathtt{x}} \in E(\mathcal{B})$. For an expression $e \in E(\mathcal{A} \cup \mathcal{C})$, we let $e\,[\boldsymbol{\pi}/\mathcal{A}]$ denote the expression $e' \in E(\mathcal{B} \cup \mathcal{C})$ resulting when we (simultaneously) replace each occurrence of every symbol $\mathtt{x} \in \mathcal{A}$ with the expression $\pi_{\mathtt{x}}$.

2.2 System Model

We model the system as having a number of *state elements*, where each state element may be a Boolean or integer value, or a function or predicate. We use symbolic names to represent the different state elements giving the set of *state symbols* $\mathcal{V}$. We introduce a set of *initial state* symbols $\mathcal{J}$ and a set of *input* symbols $\mathcal{I}$ representing, respectively, initial values and inputs that can be set to arbitrary values on each step of operation. Among the state variables, there can be *immutable* values expressing the behavior of functional units, such as ALUs, and system parameters such as the total number of processes or the maximum size of a buffer. Since these values are expressed symbolically, one run of the verifi er can prove the correctness of the system for arbitrary functionalities, process counts, and buffer capacities.

The overall system operation is characterized by an *initial-state* expression set $\mathbf{q}^0$ and a *next-state* expression set $\boldsymbol{\delta}$. The initial state consists of an expression for each state element, with the initial value of state element $\mathtt{x}$ given by expression $q^0_{\mathtt{x}} \in E(\mathcal{J})$. The transition behavior also consists of an expression for each state element, with the behavior for state element $\mathtt{x}$ given by expression $\delta_{\mathtt{x}} \in E(\mathcal{V} \cup \mathcal{I})$. In this expression, the state element symbols represent the current system state and the input symbols represent the current values of the inputs. The expression gives the new value for that state element.

We will use a very simple system as a running example throughout this presentation. The only state element is a function $\mathtt{F}$. An input symbol $\mathtt{i}$ determines which element of $\mathtt{F}$ is updated. Initially, $\mathtt{F}$ is the identify function: $q^0_{\mathtt{F}} = \lambda\, u\, .\, u$. On each step, the value of the function for argument $\mathtt{i}$ is updated to be $\mathtt{F}(\mathtt{i}+1)$. That is, $\delta_{\mathtt{F}} = \lambda\, u\, .\, \mathit{ITE}(u = \mathtt{i},\, \mathtt{F}(\mathtt{i}+1),\, \mathtt{F}(u))$, where the if-then-else operation *ITE* selects its second argument when the fi rst one evaluates to true and the third otherwise.

2.3 Concrete System

A concrete system state assigns an interpretation to every state symbol. The set of states of the concrete system is given by $\Sigma_{\mathcal{V}}$, the set of interpretations of the state element symbols. For convenience, we denote concrete states using letters s and t rather than the more formal $\sigma_{\mathcal{V}}$.

From our system model, we can characterize the behavior of the concrete system in terms of an initial state set $Q^0_C \subseteq \Sigma_{\mathcal{V}}$ and a next-state function operating on sets $N_C \colon \mathscr{P}(\Sigma_{\mathcal{V}}) \rightarrow \mathscr{P}(\Sigma_{\mathcal{V}})$. The initial state set is defi ned as $Q^0_C \doteq \{\langle \mathbf{q}^0 \rangle_{\sigma_{\mathcal{J}}} | \sigma_{\mathcal{J}} \in \Sigma_{\mathcal{J}}\}$, i.e., the set of all possible valuations of the initial state expressions. The next-state function N_C is defi ned for a single state s as $N_C(s) \doteq \{\langle \boldsymbol{\delta} \rangle_{s \cdot \sigma_{\mathcal{I}}} | \sigma_{\mathcal{I}} \in \Sigma_{\mathcal{I}}\}$, i.e., the set of all valuations of the next-state expressions for concrete state s and arbitrary input. The function is then extended to sets of states by defining $N_C(S_C) = \bigcup_{s \in S_C} N_C(s)$. We can also characterize the next-state behavior of the concrete system by a transition relation T where $(s,t) \in T$ when $t \in N_C(s)$.

We define the set of reachable states R_C as containing those states s such that there is some state sequence $s_0, s_1, \ldots, s_n$ with $s_0 \in Q_C^0$, $s_n = s$, and $s_{i+1} \in N_C(s_i)$ for all values of i such that $0 \leq i < n$. We define the *depth* of a reachable state s to be the length n of the shortest sequence leading to s. Since our concrete system has an infinite number of states, there is no finite bound on the maximum depth over all reachable states.

With our example system, the concrete state set consists of integer functions f such that $f(u+1) \geq f(u) \geq u$ for all u and $f(u) = u$ for infinitely many arguments of f.

3 Predicate Abstraction

We use quantified predicates to express constraints on the system state. To define the abstract state space, we introduce a set of *predicate* symbols $\mathcal{P}$ and a set of *index* symbols $\mathcal{X}$. The predicates consist of a named set ϕ, where for each $\mathtt{p} \in \mathcal{P}$, predicate $\phi_{\mathtt{p}}$ is a Boolean formula over the symbols in $\mathcal{V} \cup \mathcal{X}$.

Our predicates define an abstract state space $\Sigma_{\mathcal{P}}$, consisting of all interpretations $\sigma_{\mathcal{P}}$ of the predicate symbols. For $k \doteq |\mathcal{P}|$, the state space contains 2^k elements.

As an illustration, suppose for our example system we wish to prove that state element $\mathtt{F}$ will always be a function f satisfying $f(u) \geq 0$ for all $u \geq 0$. We introduce an index variable $\mathtt{x}$ and predicate symbols $\mathcal{P} = \{\mathtt{p}, \mathtt{q}\}$, with $\phi_{\mathtt{p}} \doteq \mathtt{F}(\mathtt{x}) \geq 0$ and $\phi_{\mathtt{q}} \doteq \mathtt{x} \geq 0$.

We can denote a set of abstract states by a Boolean formula $\psi \in E(\mathcal{P})$. This expression defines a set of states $\langle\psi\rangle \doteq \{\sigma_{\mathcal{P}} | \, \langle\psi\rangle_{\sigma_{\mathcal{P}}} = \mathbf{true}\}$. As an example, our two predicates $\phi_{\mathtt{p}}$ and $\phi_{\mathtt{q}}$ generate an abstract space consisting of four elements, which we denote FF, FT, TF, and TT, according to the interpretations assigned to $\mathtt{p}$ and $\mathtt{q}$. There are then 16 possible abstract state sets, some of which are shown in Table 1. In this table, abstract state sets are represented both by Boolean formulas over $\mathtt{p}$ and $\mathtt{q}$, and by enumerations of the state elements.

Abstract System		Concrete System	
Formula ψ	State Set $S_A = \langle\psi\rangle$	System Property $\forall\mathcal{X}\psi^*$	State Set $S_C = \gamma(S_A)$
$\mathtt{p} \wedge \mathtt{q}$	{TT}	$\forall\mathtt{x} : \mathtt{F}(\mathtt{x}) \geq 0 \wedge \mathtt{x} \geq 0$	$\emptyset$
$\mathtt{p} \wedge \neg\mathtt{q}$	{TF}	$\forall\mathtt{x} : \mathtt{F}(\mathtt{x}) \geq 0 \wedge \mathtt{x} < 0$	$\emptyset$
$\neg\mathtt{q}$	{FF, TF}	$\forall\mathtt{x} : \mathtt{x} < 0$	$\emptyset$
$\mathtt{p}$	{TF, TT}	$\forall\mathtt{x} : \mathtt{F}(\mathtt{x}) \geq 0$	$\{f \| \forall x : f(x) \geq 0\}$
$\mathtt{p} \vee \neg\mathtt{q}$	{FF, TF, TT}	$\forall\mathtt{x} : \mathtt{x} \geq 0 \Rightarrow \mathtt{F}(\mathtt{x}) \geq 0$	$\{f \| \forall x : x \geq 0 \Rightarrow f(x) \geq 0\}$

Table 1. Example abstract state sets and their concretizations Abstract state elements are represented by their interpretations of $\mathtt{p}$ and $\mathtt{q}$. The terms are interpreted over $\mathcal{Z}$.

We define the *abstraction function* α to map each concrete state to the set of abstract states given by the valuations of the predicates for all possible values of the index variables:

$$\alpha(s) \doteq \left\{ \langle \phi \rangle_{s \cdot \sigma_{\mathcal{X}}} | \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} \right\} \tag{1}$$

Since there are multiple interpretations $\sigma_{\mathcal{X}}$, a single concrete state will generally map to multiple abstract states. This feature is not found in most uses of predicate abstraction, but it is the key idea for handling quantified predicates. We then extend the abstraction function to apply to sets of concrete states in the usual way: $\alpha(S_C) \doteq \bigcup_{s \in S_C} \alpha(s)$. We can see that α is monotonic, i.e., that if $S_C \subseteq T_C$, then $\alpha(S_C) \subseteq \alpha(T_C)$.

Working with our example system, consider the concrete state given by the function $\lambda\, u\, .\, u$. When we abstract this function relative to predicates $\phi_{\mathtt{p}}$ and $\phi_{\mathtt{q}}$, we get two abstract states: TT, when $\mathtt{x} \geq 0$, and FF, when $\mathtt{x} < 0$. This abstract state set is then characterized by the formula $\mathtt{p} \Leftrightarrow \mathtt{q}$.

We define the concretization function γ to require universal quantification over the index symbols. That is, for a set of abstract states $S_A \subseteq \Sigma_{\mathcal{P}}$:

$$\gamma(S_A) \quad \doteq \quad \left\{ s | \forall \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} : \langle \phi \rangle_{s \cdot \sigma_{\mathcal{X}}} \in S_A \right\} \tag{2}$$

The universal quantifier in this definition has the consequence that the concretization function does not distribute over set union. In particular, we cannot view the concretization function as operating on individual abstract states, but rather as generating each concrete state from multiple abstract states. Nonetheless, γ is montonic, i.e., if $S_A \subseteq T_A$, then $\gamma(S_A) \subseteq \gamma(T_A)$.

Consider our example system with predicates $\phi_{\mathtt{p}}$ and $\phi_{\mathtt{q}}$. Table 1 shows some example abstract state sets S_A and their concretizations $\gamma(S_A)$. As the first three examples show, some (altogether 6) nonempty abstract state sets have empty concretizations, because they constrain $\mathtt{x}$ to be either always negative or always non-negative. On the other hand, there are 9 abstract state sets having nonempty concretizations. We can see by this that the concretization function is based on the entire abstract state set and not just on the individual values. For example, the sets $\{$TF$\}$ and $\{$TT$\}$ have empty concretizations, but $\{$TF, TT$\}$ concretizes to the set of all non-negative functions.

Theorem 1. *The functions (α, γ) form a Galois connection* [1]*, i.e., for any sets of concrete states S_C and abstract states S_A:*

$$\alpha(S_C) \subseteq S_A \;\Leftrightarrow\; S_C \subseteq \gamma(S_A) \tag{3}$$

The proof follows by observing that both the left and the right-hand sides of (3) hold precisely when for every $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ and every $s \in S_C$, we have $\langle \phi \rangle_{s \cdot \sigma_{\mathcal{X}}} \in S_A$.

[1] This is one of several logically equivalent formulations of a Galois connection [6].

A Galois connection also satisfies the property (follows from (3)) that for any set of concrete states S_C:

$$S_C \subseteq \gamma(\alpha(S_C)). \tag{4}$$

The containment relation in (4) can be proper. For example, the concrete state set consisting of the single function $\lambda u \,.\, u$ abstracts to the state set $\mathtt{p} \Leftrightarrow \mathtt{q}$, which in turn concretizes to the set of all functions f such that $f(u) \geq 0 \Leftrightarrow u \geq 0$.

4 Abstract System

Predicate abstraction involves performing a reachability analysis over the abstract state space, where on each step we concretize the abstract state set via γ, apply the concrete next-state function, and then abstract the results via α. We can view this process as performing reachability analysis on an abstract system having initial state set $Q_A^0 \doteq \alpha(Q_C^0)$ and a next-state function operating on sets: $N_A(S_A) \doteq \alpha(N_C(\gamma(S_A)))$. Note that there is no transition relation associated with this next-state function, since γ cannot be viewed as operating on individual abstract states.

It can be seen that N_A provides an *abstract interpretation* of the concrete system [6, 5]:

1. N_A is null-preserving: $N_A(\emptyset) = \emptyset$
2. N_A is monotonic: $S_A \subseteq T_A \;\Rightarrow\; N_A(S_A) \subseteq N_A(T_A)$
3. N_A simulates N_C (a simulation relation defined by α): $\alpha(N_C(S_C)) \subseteq N_A(\alpha(S_C))$

We perform reachability analysis on the abstract system using N_A as the next-state function:

$$R_A^0 = Q_A^0 \tag{5}$$

$$R_A^{i+1} = R_A^i \cup N_A(R_A^i) \tag{6}$$

Since the abstract system is finite, there must be some n such that $R_A^n = R_A^{n+1}$. The set of all reachable abstract states R_A is then R_A^n. By induction on n, it can be shown that if s is a reachable state in the concrete system with depth $\leq n$, then $\alpha(s) \subseteq R_A^n$. From this it follows that $\alpha(s) \subseteq R_A$ for any concrete reachable state s, and therefore that $\alpha(R_C) \subseteq R_A$. Thus, even though determining the set of reachable concrete states would require examining paths of unbounded length, we can compute a conservative approximation to this set by performing a bounded reachability analysis of the abstract system.

It is worth noting that we cannot use the standard “frontier set” optimization in our reachability analysis. This optimization, commonly used in symbolic model checking, considers only the newly reached states in computing the next set of reachable states. In our context, this would mean using the computation $R_A^{i+1} = R_A^i \cup N_A(R_A^i - R_A^{i-1})$ rather than that of (6). This optimization is not valid, due to the fact that γ, and therefore N_A, does not distribute over set union.

As an illustration, let us perform reachability analysis on our example system. In the initial state, state element F is the identity function, which we have seen abstracts to the set represented by the formula $\mathtt{p} \Leftrightarrow \mathtt{q}$. This abstract state set concretizes to the set of functions f satisfying $f(u) \geq 0 \Leftrightarrow u \geq 0$. Let h denote the value of F in the next state. If input i is -1, we would $h(-1) = f(0) \geq 0$, but we can still guarantee that $h(u) \geq 0$ for $u \geq 0$. Applying the abstraction function, we get R_A^1 characterized by the formula $\mathtt{p} \vee \neg\mathtt{q}$ (see Table 1.) For the second iteration, the abstract state set characterized by the formula $\mathtt{p} \vee \neg\mathtt{q}$ concretizes to the set of functions f satisfying $f(u) \geq 0$ when $u \geq 0$, and this condition must hold in the next state as well. Applying the abstraction function to this set, we then get $R_A^2 = R_A^1$, and hence the process has converged.

5 Verifying Safety Properties

A Boolean formula $\psi \in E(\mathcal{P})$ defines a *property* of the abstract state space. The property is said to hold for the abstract system when it holds for every reachable abstract state. That is, $\langle\psi\rangle_{\sigma_\mathcal{P}} = \mathbf{true}$ for all $\sigma_\mathcal{P} \in R_A$.

For Boolean formula $\psi \in E(\mathcal{P})$, define the formula $\psi^* \in E(\mathcal{V} \cup \mathcal{X})$ to be the result of substituting the predicate expression $\phi_\mathtt{p}$ for each predicate symbol $\mathtt{p} \in \mathcal{P}$. That is, viewing $\boldsymbol{\phi}$ as a substitution, we have $\psi^* \doteq \psi\,[\boldsymbol{\phi}/\mathcal{P}]$. Formula ψ^* defines a property $\forall\mathcal{X}\psi^*$ of the concrete states. The property holds for concrete state s, written $\forall\mathcal{X}\psi^*(s)$, when $\langle\psi^*\rangle_{s\cdot\sigma_\mathcal{X}} = \mathbf{true}$ for every $\sigma_\mathcal{X} \in \Sigma_\mathcal{X}$. The property holds for the concrete system when $\forall\mathcal{X}\psi^*(s)$ holds for every reachable concrete state $s \in R_C$. Table 1 shows the concrete system properties given by different abstract state formulas ψ.

Theorem 2. *For a formula $\psi \in E(\mathcal{P})$, if property ψ holds for the abstract system, then property $\forall\mathcal{X}\psi^*$ holds for the concrete system.*

This follows by the definition of α and the fact that $\alpha(R_C) \subseteq R_A$.

With our example system, letting formula $\psi \doteq \mathtt{p}\vee\neg\mathtt{q}$, and noting that $\mathtt{p}\vee\neg\mathtt{q} \equiv \mathtt{q} \Rightarrow \mathtt{p}$, we get the property $\forall\mathtt{x} : \mathtt{x} \geq 0 \Rightarrow \mathtt{F}(\mathtt{x}) \geq 0$.

Using predicate abstraction, we can possibly get a *false negative* result, where we fail to verify a property $\forall\mathcal{X}\psi^*$, even though it holds for the concrete system, because the given set of predicates does not adequately capture the characteristics of the system that ensure the desired property. Thus, this method of verifying properties is sound, but possibly incomplete.

We can precisely characterize the class of properties for which the predicate abstraction is both sound and complete, assuming we have a decision procedure that can determine whether a universally quantified formula in the underlying logic is satisfiable. A property $\forall\mathcal{X}\psi^*$ is said to be *inductive* for the concrete system when it satisfies the following two properties:

1. Every initial state $s \in Q_C^0$ satisfies $\forall\mathcal{X}\psi^*(s)$.
2. For all concrete states s and $t \in N_C(s)$, if $\forall\mathcal{X}\psi^*(s)$ holds, then so does $\forall\mathcal{X}\psi^*(t)$.

Clearly an inductive property must hold for every reachable concrete state and therefore for the concrete system. It can also be shown that if $\forall \mathcal{X} \psi^*$ is inductive, then ψ holds for the abstract system. That is, if we present the predicate abstraction engine with a fully formed induction hypothesis, it can prove that it holds.

For formula $\psi \in E(\mathcal{P})$ and predicate set $\boldsymbol{\phi}$, the property $\forall \mathcal{X} \psi^*$ is said to *have an induction proof over* $\boldsymbol{\phi}$ when there is some formula $\chi \in E(\mathcal{P})$, such that $\chi \Rightarrow \psi$ and $\forall \mathcal{X} \chi^*$ is inductive. That is, there is some way to strengthen ψ into a formula χ that can be used to prove the property by induction.

Theorem 3. *A formula $\psi \in E(\mathcal{P})$ is a property of the abstract system if and only if the concrete property $\forall \mathcal{X} \psi^*$ has an induction proof over the predicate set $\boldsymbol{\phi}$.*

This theorem precisely characterizes the capability of our formulation of predicate abstraction—it can prove any property that can be proved by induction using an induction hypothesis expressed in terms of the predicates. Thus, if we fail to verify a system using this form of predicate abstraction, we can conclude that either 1) the system does not satisfy the property, or 2) we did not provide an adequate set of predicates to construct an universally quantifi ed induction hypothesis, provided one exists.

6 Quantifier Instantiation

For many subsets of fi rst-order logic, there is no complete method for handling the universal quantifi er introduced in function γ (Equation 2). For example, in a logic with uninterpreted functions and equality, determining whether a universally quantifi ed formula is satisfi able is undecidable [3]. Instead, we concretize abstract states by considering some limited subset of the interpretations of the index symbols, each of which is defi ned by a substitution for the symbols in $\mathcal{X}$. Our tool automatically generates candidate substitutions based on the subexpressions that appear in the predicate and next-state expressions [13]. These subexpressions can contain symbols in $\mathcal{V}$, $\mathcal{X}$, and $\mathcal{I}$. These instantiated versions of the formulas enable to verifi er to detect specifi c cases where the predicates can be applied. Flanagan and Qadeer use a similar technique [8].

More precisely, let $\boldsymbol{\pi}$ be a substitution assigning an expression $\pi_{\mathtt{x}} \in E(\mathcal{V} \cup \mathcal{X} \cup \mathcal{I})$ for each $\mathtt{x} \in \mathcal{X}$. Then $\phi_{\mathtt{p}}\,[\boldsymbol{\pi}/\mathcal{X}]$ will be a Boolean expression over symbols $\mathcal{V}$, $\mathcal{X}$, and $\mathcal{I}$ that represents some instantiation of predicate $\phi_{\mathtt{p}}$. For a set of substitutions Π and interpretations $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ and $\sigma_{\mathcal{I}} \in \Sigma_{\mathcal{I}}$, we defi ne the concretization function γ_{Π} as:

$$\gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}) \quad \doteq \quad \left\{ s | \forall \boldsymbol{\pi} \in \Pi : \langle \boldsymbol{\phi}\,[\boldsymbol{\pi}/\mathcal{X}] \rangle_{s \cdot \sigma_{\mathcal{X}} \cdot \sigma_{\mathcal{I}}} \in S_A \right\} \tag{7}$$

It can be seen that γ_{Π} is an overapproximation of γ, i.e., that $\gamma(S_A) \subseteq \gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}})$ for any abstract state S_A, set of substitutions Π, and interpretations $\sigma_{\mathcal{X}}$ and $\sigma_{\mathcal{I}}$. From (4), it then follows that

$$S_C \quad \subseteq \quad \gamma_{\Pi}(\alpha(S_C), \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}). \tag{8}$$

and hence the functions (α, γ_{Π}) satisfy property (4) of a Galois connection, even though they are not a true Galois connection.

We can use γ_{Π} as an approximation to γ in defining the behavior of the abstract system. That is, define N_{Π} over sets of abstract states as:

$$N_{\Pi}(S_A) = \left\{ \langle \boldsymbol{\phi}\,[\boldsymbol{\delta}/\mathcal{V}] \rangle_{s \cdot \sigma_{\mathcal{X}} \cdot \sigma_{\mathcal{I}}} \,|\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}, \sigma_{\mathcal{I}} \in \Sigma_{\mathcal{I}}, s \in \gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}) \right\} \tag{9}$$

Observe in this equation that $\phi_{\mathtt{p}}\,[\boldsymbol{\delta}/\mathcal{V}]$ is an expression describing the evaluation of predicate $\phi_{\mathtt{p}}$ in the next state. It can be seen that $N_{\Pi}(S_A) \supseteq N_A(S_A)$ for any set of abstract states S_A. As long as Π is nonempty (required to guarantee that N_{Π} is null-preserving), it can be shown that the system defined by N_{Π} is an abstract interpretation of the concrete system. We can therefore perform reachability analysis:

$$R_{\Pi}^{0} = Q_A^0 \tag{10}$$
$$R_{\Pi}^{i+1} = R_{\Pi}^{i} \cup N_{\Pi}(R_{\Pi}^{i}) \tag{11}$$

These iterations will converge to a set R_{Π}. For every step i, we can see that $R_{\Pi}^{i} \supseteq R_A^i$, and therefore we must have $R_{\Pi} \supseteq R_A$.

Theorem 4. *For a formula $\psi \in E(\mathcal{P})$, if $\langle \psi \rangle_{\sigma_{\mathcal{P}}} = \mathbf{true}$ for every $\sigma_{\mathcal{P}} \in R_{\Pi}$, then property $\forall \mathcal{X} \psi^*$ holds for the concrete system.*

This demonstrates that using quantifier instantiation during reachability analysis yields a sound verification technique. However, when the tool fails to verify a property, it could mean, in addition to the two possibilities listed earlier, that 3) it used an inadequate set of instantiations, or 4) that the property cannot be proved by any bounded set of quantifier instantiations.

7 Symbolic Formulation of Reachability Analysis

We are now ready to express the reachability computation symbolically, where each step involves finding the set of satisfying solutions to an existentially quantified formula. On each step, we generate a Boolean formula ρ_{Π}^{i}, that characterizes R_{Π}^{i}. That is $\langle \rho_{\Pi}^{i} \rangle = R_{\Pi}^{i}$. The formulas directly encode the approximate reachability computations of (10) and (11).

Observe that by composing the predicate expressions with the initial state expressions, $\boldsymbol{\phi}\,[\mathbf{q}^0/\mathcal{V}]$, we get a set of predicates over the initial state symbols $\mathcal{J}$ indicating the conditions under which the predicates hold in the initial state. We can therefore start the reachability analysis by finding solutions to the formula

$$\rho_{\Pi}^{0}(\mathcal{P}) = \exists \mathcal{J} \exists \mathcal{X} \bigwedge_{\mathtt{p} \in \mathcal{P}} \mathtt{p} \Leftrightarrow \phi_{\mathtt{p}}\,[\mathbf{q}^0/\mathcal{V}] \tag{12}$$

The formula for the next-state computation combines the definitions of N_{Π} (9) and γ_{Π} (7):

$$\rho_{\Pi}^{i+1}(\mathcal{P}) = \rho_{\Pi}^{i}(\mathcal{P}) \vee \exists \mathcal{V} \exists \mathcal{X} \exists \mathcal{I} \left(\bigwedge_{\boldsymbol{\pi} \in \Pi} \left(\rho_{\Pi}^{i}\,[\boldsymbol{\phi}/\mathcal{P}] \right) [\boldsymbol{\pi}/\mathcal{X}] \;\; \wedge \;\; \bigwedge_{\mathtt{p} \in \mathcal{P}} \mathtt{p} \Leftrightarrow \phi_{\mathtt{p}}\,[\boldsymbol{\delta}/\mathcal{V}] \right). \tag{13}$$

To understand the quantified term in this equation, note that the left-hand term is the formula for $\gamma_{\Pi}(\rho^i_{\Pi}, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}})$, while the right-hand term expresses the conditions under which each abstract state variable $\mathtt{p}$ will match the value of the corresponding predicate in the next state.

Let us see how this symbolic formulation would perform reachability analysis for our example system. Recall that our system has two predicates $\phi_{\mathtt{p}} \doteq \mathtt{F}(\mathtt{x}) \geq 0$ and $\phi_{\mathtt{q}} \doteq \mathtt{x} \geq 0$. In the initial state, $\mathtt{F}$ is the function $\lambda\, u \,.\, u$, and therefore $\phi_{\mathtt{p}}\left[\mathbf{q}^0/\mathcal{V}\right]$ simply becomes $\mathtt{x} \geq 0$. Equation (12) then becomes $\exists \mathtt{x}\, [(\mathtt{p} \Leftrightarrow \mathtt{x} \geq 0) \wedge (\mathtt{q} \Leftrightarrow \mathtt{x} \geq 0)]$, which reduces to $\mathtt{p} \Leftrightarrow \mathtt{q}$.

Now let us perform the first iteration. For our instantiations we require two substitutions $\boldsymbol{\pi}$ and $\boldsymbol{\pi}'$ with $\pi_{\mathtt{x}} = \mathtt{x}$ and $\pi'_{\mathtt{x}} = \mathtt{i}+1$. For $\rho^0_{\Pi}(\mathtt{p}, \mathtt{q}) = \mathtt{p} \Leftrightarrow \mathtt{q}$, the left-hand term of (13) instantiates to $(\mathtt{F}(\mathtt{x}) \geq 0 \Leftrightarrow \mathtt{x} \geq 0) \wedge (\mathtt{F}(\mathtt{i}+1) \geq 0 \Leftrightarrow \mathtt{i}+1 \geq 0)$. Substituting $\lambda\, u\,.\mathit{ITE}(u = \mathtt{i},\, \mathtt{F}(\mathtt{i}{+}1),\, \mathtt{F}(u))$ for $\mathtt{F}$ in $\phi_{\mathtt{p}}$ gives $(\mathtt{x}{=}\mathtt{i} \wedge \mathtt{F}(\mathtt{i}{+}1) \geq 0) \vee (\mathtt{x} {\neq} \mathtt{i} \wedge \mathtt{F}(\mathtt{x}) \geq 0)$.

The quantified portion of (13) for $\rho^1_{\Pi}(\mathtt{p}, \mathtt{q})$ then becomes

$$\exists\, \mathtt{F}, \mathtt{x}, \mathtt{i} : \begin{pmatrix} \quad \mathtt{F}(\mathtt{x}) \geq 0 \Leftrightarrow \mathtt{x} \geq 0 \quad \wedge \quad \mathtt{F}(\mathtt{i}+1) \geq 0 \Leftrightarrow \mathtt{i}+1 \geq 0 \\ \wedge\ \mathtt{p} \Leftrightarrow [(\mathtt{x} = \mathtt{i} \wedge \mathtt{F}(\mathtt{i}+1) \geq 0) \vee (\mathtt{x} \neq \mathtt{i} \wedge \mathtt{F}(\mathtt{x}) \geq 0)] \\ \wedge\ \mathtt{q} \Leftrightarrow \mathtt{x} \geq 0 \end{pmatrix}$$

The only values of $\mathtt{p}$ and $\mathtt{q}$ where this formula cannot be satisfied is when $\mathtt{p}$ is false and $\mathtt{q}$ is true.

As shown in [12], we can generate the set of solutions to (12) and (13) by first transforming the formulas into equivalent Boolean formulas and then performing Boolean quantifier elimination to remove all Boolean variables other than those in $\mathcal{P}$. This quantifier elimination is similar to the relational product operation used in symbolic model checking and can be solved using either BDD or SAT-based methods.

8 Axioms

As a special class of predicates, we may have some that are to hold at all times. For example, we could have an axiom $\mathtt{f}(\mathtt{w}) > 0$ to indicate that function $\mathtt{f}$ is always positive, or $\mathtt{f}(\mathtt{y}, \mathtt{z}) = \mathtt{f}(\mathtt{z}, \mathtt{y})$ to indicate that $\mathtt{f}$ is commutative. Typically, we want these predicates to be individually quantified, but we can ensure this by defining each of them over a unique set of index symbols, as we have done in the above examples.

We can add this feature to our analysis by identifying a subset $\mathcal{Q}$ of the predicate symbols $\mathcal{P}$ to be axioms. We then want to restrict the analysis to states where the axiomatic predicates hold. Let $\Sigma^{\mathcal{Q}}_{\mathcal{P}}$ denote the set of abstract states $\sigma_{\mathcal{P}}$ where $\sigma_{\mathcal{P}}(\mathtt{p}) = \mathbf{true}$ for every $\mathtt{p} \in \mathcal{Q}$. Then we can apply this restriction by redefining $\alpha(s)$ (Equation 1) for concrete state s to be:

$$\alpha(s) \doteq \left\{ \langle \boldsymbol{\phi} \rangle_{s \cdot \sigma_{\mathcal{X}}} | \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} \right\} \cap \Sigma^{\mathcal{Q}}_{\mathcal{P}} \tag{14}$$

and then using this definition in the extension of α to sets, the formulation of the reachability analysis (Equations 5 and 6), and the approximate reachability analysis (Equations 10 and 11).

9 Applications

We have used our predicate abstraction tool to verify safety properties of a variety of models and protocols. Some of the more interesting ones include:

- A microprocessor out-of-order execution unit with an unbounded retirement buffer. Prior verification of this unit required manually generating 13 invariants [13].
- A directory-based cache protocol with unbounded channels, devised by Steven German of IBM [9], as discussed below.
- A version of Lamport's bakery algorithm [14] that allows arbitrary number of processes and nonatomic reads and writes.
- Selection sort algorithm for sorting an arbitrary large array. We prove the property that upon termination, the algorithm produces a sorted array.

For the directory-based German's cache-coherence protocol, an unbounded number of clients ($\texttt{cache}$), communicate with a central *home* process to gain *exclusive* or *shared* access to a memory line. The state of each $\texttt{cache}$ can be $\{$INVALID, SHARED, EXCLUSIVE$\}$. The home maintains explicit representations of two lists of clients: those sharing the cache line ($\texttt{home_sharer_list}$) and those for which the home has sent an invalidation request but has not received an acknowledgement ($\texttt{home_invalidate_list}$).

The client places requests $\{$REQ_SHARED, REQ_EXCLUSIVE$\}$ on a channel $\texttt{ch\_1}$ and the home grants $\{$GRANT_SHARED, GRANT_EXCLUSIVE$\}$ on channel $\texttt{ch\_2}$. The home also sends invalidation messages INVALIDATE along $\texttt{ch\_2}$. The home grants exclusive access to a client only when there are no clients sharing a line, i.e. $\forall \texttt{i} : \texttt{home\_sharer\_list}(\texttt{i}) = \textbf{false}$. The home maintains variables for the current client ($\texttt{home\_current\_client}$) and the request it is currently processing ($\texttt{home\_current\_command}$). It also maintains a bit $\texttt{home\_exclusive\_granted}$ to indicate that some client has exclusive access. The cache lines acknowledge invalidation requests with a INVALIDATE_ACK along another channel $\texttt{ch\_3}$. Details of the protocol operation with single-entry channels can be found in many previous works including [15].

In our version of the protocol, each $\texttt{cache}$ communicates to the home process through three directed unbounded FIFO channels, namely the channels $\texttt{ch\_1}$, $\texttt{ch\_2}$, $\texttt{ch\_3}$. Thus, there are an unbounded number of unbounded channels, three for each client[2]. It can be shown that a client can generate an unbounded number of requests before getting a response from the home.

To model the protocol in CLU, we need to change the predicate state variable representation of $\texttt{home\_sharer\_list}$. Since the transition functions are expressed over quantifier-free logic, we cannot support a universal quantifier in the model. Instead, we model $\texttt{home\_sharer\_list}$ as a *set*, using (1) a queue $\texttt{hsl\_q} \doteq \langle \texttt{q}, \texttt{hd}, \texttt{tl} \rangle$ to store all cache indices $\texttt{i}$ for which $\texttt{home\_sharer\_list}(\texttt{i}) = \textbf{true}$ and (2) an array $\texttt{hsl\_pos}$ to map a cache index $\texttt{i}$ to the position in the queue, if $\texttt{i} \in \texttt{hsl\_q}$. This representation can support addition, deletion, membership-check and emptiness-check, which are the operations required for this protocol. In addition, this representation also allows us to *enumerate* the cache indices for which $\texttt{home\_sharer\_list}(\texttt{i}) = \textbf{true}$.

[2] The extension was suggested by Steven German himself

We had previously verified the cache-coherence property of the protocol with 31 non-trivial, manually constructed invariants. In contrast, the predicate abstraction constructs the strongest inductive invariant automatically with 29 predicates, all of which are simple and do not involve any Boolean connectives. There are 2 index variables in $\mathcal{X}$ to specify the predicates. The abstract reachability took 19 iterations and 263 minutes to produce the inductive invariant [3]. For the simpler version which has single-entry channels for communication, our method finds the inductive invariant in 85s using 17 predicates in 9 iterations. All experiments were performed on a 2.1 GHz Linux machine with 1GB of RAM. The main difficulty of making the channels unbounded is the presence of two-dimensional arrays in the model, and additional state variables for the head and tail pointers for each of the unbounded queues.

For space considerations, we will only describe the nature of predicates used for the model with single-entry channels. A few predicates did not require any index symbol. These include: $\mathtt{home\_exclusive\_granted}$, $\mathtt{home\_current\_command} = \text{REQ\_SHARED}$, $\mathtt{home\_current\_command} = \text{REQ\_EXCLUSIVE}$, $\mathtt{hd} < \mathtt{tl}$ and $\mathtt{hd} = \mathtt{tl}$. For most predicates, we required a single index variable $\mathtt{i} \in \mathcal{X}$, to denote an arbitrary cache index. They include: $\mathtt{home\_invalidate\_list(i)}$, $\mathtt{cache(i)} = \text{EXCLUSIVE}$, $\mathtt{cache(i)} = \text{SHARED}$, $\mathtt{ch\_2(i)} = \text{GRANT\_EXCLUSIVE}$, $\mathtt{ch\_2(i)} = \text{GRANT\_SHARED}$, $\mathtt{ch\_2(i)} = \text{INVALIDATE}$, $\mathtt{ch\_3(i)} = \text{INVALIDATE\_ACK}$ and $\mathtt{i} = \mathtt{q(hd)}$. We also required another index variable $\mathtt{j} \in \mathcal{X}$ to range over the entries of the queue $\mathtt{hsl\_q}$. The predicates over $\mathtt{j}$ are $\mathtt{hd} \leq \mathtt{j}$ and $\mathtt{j} < \mathtt{tl}$. Finally, to relate the entries in $\mathtt{hsl\_q}$ and $\mathtt{hsl\_pos}$, we needed the predicates $\mathtt{i} = \mathtt{q(j)}$ and $\mathtt{j} = \mathtt{hsl\_pos(i)}$.

Most of the predicates are fairly easy to find from the model and from counterexamples. Predicate abstraction constructs an inductive invariant of the form $\forall \mathtt{i}, \mathtt{j} : \psi^*(\mathtt{i}, \mathtt{j})$, which implies the cache-coherence property. This implication is checked automatically with a sound decision procedure in UCLID [4], using quantifier instantiation.

Previous attempts at using predicate abstraction (with locally quantified predicates), for a version of the protocol with single-entry channels required complex quantified predicates [7, 2], sometimes as complex as an invariant. However, Baukus et al. [2] proved the liveness of the protocol in addition to the cache-coherence property. Pnueli et al. [15] have used the method of *invisible* invariants to derive the inductive invariant for the model with single-entry channels, but it is not clear if their formalism can model the version with unbounded channels per client.

10 Conclusions

We have found quantified invariants to be essential in expressing the properties of systems with function state variables. The ability of our tool to automatically generate quantified invariants based on small and simple predicates allows us to deal with much more complex systems in a more automated fashion than previous work. A next step would be to automatically generate the set of predicates used by the predicate abstraction tool. Other tools generate new predicates based on the counterexample traces from the abstract model [1, 7]. This approach cannot be used directly in our context, since our

[3] There is a lot of scope for optimizing the performance of our procedure.

abstract system cannot be viewed as a state transition system, and so there is no way to characterize a counterexample by a single state sequence. We are currently looking at techniques to extract relevant predicates from the proof of unsatisfi able formulas which represent that an error state can't be reached after any fi nite number of steps.

Acknowledgments We wish to thank Ching-Tsun Chou for his detailed comments on an early draft of this paper.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.
2. K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In A. Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation, (VMCAI '02)*, LNCS 2294, pages 317–330, January 2002.
3. E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.
4. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Computer-Aided Verification (CAV '02)*, LNCS 2404, pages 78–92, 2002.
5. C. T. Chou. The mathematical foundation fo symbolic trajectory evaluation. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification (CAV'99)*, LNCS 1633, pages 196–207, 1999.
6. P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL '77)*, pages 238–252, 1977.
7. S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 19–32, 2002.
8. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In J. Launchbury and J. C. Mitchell, editors, *Principles of Programming Languages (POPL '02)*, pages 191–202, 2002.
9. S. German. Personal communication.
10. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer-Aided Verification (CAV '97)*, LNCS 1254, pages 72–83, 1997.
11. C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murφ. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification (CAV '96)*, LNCS 1102, pages 147–158, 1996.
12. S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer-Aided Verification (CAV '03)*, LNCS 2725, pages 141–153, 2003.
13. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 142–159, 2002.
14. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17:453–455, August 1974.
15. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, LNCS 2031, pages 82–97, 2001.