Predicate Abstraction with Indexed Predicates

SHUVENDU K. LAHIRI
RANDAL E. BRYANT
Carnegie Mellon University

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 containing first-order state variables, where the system state includes mutable functions and predicates. 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 first-order 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.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Invariants*

General Terms: Verification, Predicate Abstraction

Additional Key Words and Phrases: formal verification, invariant synthesis, infinite-state verification, abstract interpretation, cache-coherence protocols

1. INTRODUCTION

Graf and Saïdi introduced *predicate abstraction* [Graf and Saïdi 1997] 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 [Graf and Saïdi 1997; Saïdi and Shankar 1999; Das et al. 1999; Das and Dill 2001; Ball et al. 2001; Flanagan and Qadeer 2002; Chaki et al. 2003] 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

Authors address: Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213.
A shorter version of the paper [Lahiri and Bryant 2004a] appeared at *International Conference on Verification, Model Checking and Abstract Interpretation, (VMCAI) 2004*
The research was supported in part by the Semiconductor Research Corporation, contract RID 1029.001.

that both BDD and SAT-based Boolean methods can be applied to perform the analysis efficiently [Lahiri et al. 2003].

In most formulations of predicate abstraction, the predicates contain no free variables, and hence 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.

To verify systems containing unbounded resources, such as buffers and memories of arbitrary size and systems with arbitrary numbers of identical, concurrent processes, the system model must support *first-order* state variables, in which the state variables are themselves functions or predicates [Ip and Dill 1996; Bryant et al. 2002b]. 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 value of 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 [Burch and Dill 1994; McMillan 1998]. Our verifier allows a more general form of mutable function, where the updating operation is expressed using lambda notation.

In verifying systems with first-order 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 $\forall i, j : \mathtt{crit}(i) \wedge \mathtt{crit}(j) \Rightarrow i = j$. 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 (the fact that the universal quantifier does not distribute over disjunctions) 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 [Das and Dill 2002], but this is a formidable task.

In this paper we present an extension of predicate abstraction in which the predicates include free variables from a set of *index* variables $\mathcal{X}$ (and hence the name *indexed predicates*). 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 $\mathtt{crit}(\mathtt{i})$, $\mathtt{crit}(\mathtt{j})$, and $\mathtt{i} = \mathtt{j}$, where $\mathtt{i}$ and $\mathtt{j}$ are the index symbols. Encoding these predicates in the abstract system with Boolean variables $\mathtt{ci}$, $\mathtt{cj}$, and $\mathtt{eij}$, respectively, we can verify this property by using predicate abstraction to prove that $\mathtt{ci} \wedge \mathtt{cj} \Rightarrow \mathtt{eij}$ holds for every reachable state of the abstract system.

Flanagan and Qadeer use a method similar to ours [Flanagan and Qadeer 2002], and we briefly described our method in an earlier paper [Lahiri et al. 2003]. 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 abstraction interpretation of the concrete system [Cousot and Cousot 1977] and hence can be used to find invariant system properties.

Assuming a decision procedure that can determine the satisfiability of a formula with universal quantifiers, we prove the following completeness result: 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 indexed predicates that must evaluate to true on every step.

The ideas have been implemented in UCLID [Bryant et al. 2002b], a platform for modeling and verifying infinite-state systems. Although we demonstrate the ideas in the context of this tool and the logic (CLU) it supports, the ideas developed here are not strongly tied to this logic. 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 [German]. 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.

1.1 Related Work

Verifying systems with unbounded resources is in general undecidable. For instance, the problem of verifying if a system of N (N can be arbitrarily large) concurrent processes satisfies a property is undecidable [Apt and Kozen 1986]. Despite its complexity, the problem of verifying systems with arbitrarily large resources (e.g. parameterized systems with N processes, out-of-order processors with arbitrarily large reorder buffers, software programs with arbitrary large arrays) is of significant practical interest. Hence, in recent years, there has been a lot of interest in developing techniques based on model checking and deductive approaches for verifying such systems.

McMillan uses "compositional model checking" [McMillan 1998] with various built-in abstractions to reduce an infinite-state system to a finite state system, which can be model checked using Boolean methods. The abstraction mechanisms include *temporal case splitting, datatype reduction* [Clarke et al. 1992] and *symmetry reduction*. Temporal case splitting uses heuristics to slice the program space to only consider the resources necessary for proving a property. Datatype reduction uses abstract interpretation [Cousot and Cousot 1977] to abstract unbounded data and operations over them to operations over finite domains. For such finite domains, datatype reduction is subsumed by predicate abstraction. Symmetry is exploited to reduce the number of indices to consider for verifying unbounded arrays or network of processes. The method can prove both safety and liveness properties. Since the abstraction mechanisms are built into the system, they can often be coarse and may not suffice for proving a system. Besides, the user is often required to provide auxiliary lemmas or to decompose the proof to be discharged by symbolic model checkers. For instance, the proof of safety of the Bakery protocol [McMillan et al. 2000] or the proof of an out-of-order processor model [McMillan 1998] required non-trivial lemmas in the compositional model checking framework.

Regular model checking [Kesten et al. 1997; Bouajjani et al. 2000] uses regular languages to represent parameterized systems and computes the closure for the regular relations to construct the reachable state space. In general, the method is not guaranteed to be complete and requires various *acceleration* techniques (sometimes guided by the user) to ensure termination. Moreover, approaches based on regular language are not suited for representing data in the system. Several examples that we consider in this work can't be modeled in this framework; the out-of-order processor which contains data operations or the Peterson's mutual exclusion are few such examples. Even though the Bakery algorithm can be verified in this framework, it requires considerable user ingenuity to encode the protocol in a regular language.

Several researchers have investigated restrictions on the system description to make the parameterized verification problem decidable. Notable among them is the early work by German and Sistla [German and Sistla 1992] for verifying single-indexed properties for synchronously communicating systems. For restricted systems, finite "cut-off" based approaches [Emerson and Namjoshi 1995; Emerson and Kahlon 2000; 2003] reduce the problem to verifying networks of some fixed finite size. These bounds have been established for verifying restricted classes of ring networks and cache coherence protocols. Emerson and Kahlon [Emerson and Kahlon 2003] have verified the version of German's cache coherence protocol with single entry channels by manually reducing it to a snoopy protocol, for which finite cut-off exists. However, the reduction is manually performed and exploits details of operation of the protocol, and thus requires user ingenuity. It can't be easily extended to verify other unbounded systems including the Bakery algorithm or the out-of-order processors.

The method of "invisible invariants" [Pnueli et al. 2001; Arons et al. 2001] uses heuristics for constructing universally quantified invariants for parameterized systems automatically. The method computes the set of reachable states for finite (and small) instances of the parameters and then generalizes them to parameterized systems to construct a potential inductive invariant. They provide an algorithm for checking the verification conditions for a restricted class of systems called the *stratified* systems, which include German's protocol with single entry channels and Lamport's Bakery protocol [Lamport 1974]. However, the method simply becomes a heuristic for generating candidate invariants for non-stratified systems, which includes Peterson's mutual exclusion algorithm [Peterson 1981] and the Ad-hoc On-demand Distance Vector (AODV) [C.Perkins et al. 2002] network protocol. The class of *bounded-data* systems (where each variable is finite but parameterized) considered by this approach can't model the the out-of-order processor model [Lahiri et al. 2002] that we can verify using our method.

Predicate abstraction with locally quantified predicates [Das and Dill 2002; Baukus et al. 2002] requires complex quantified predicates to construct the inductive assertions, as mentioned in the introduction. These predicates are often as complex as invariants themselves. In fact, some of the invariants are used as predicates in [Baukus et al. 2002] to derive inductive invariants. The method in [Baukus et al. 2002] verified (both safety and liveness) a version of the cache coherence protocol with single entry channels, with complex manually provided predicates. Baukus et al. [Baukus et al. 2002] uses the the logic of *WSIS* (weak second order logic with one successor) [Buchi 1960; Thomas 1990], which does not allow function symbols and thus can't model the out-of-order processor model. The automatic predicate discovery methods for quantified predicates [Das and Dill 2002] have not been demonstrated on most examples (except the AODV model) we consider in this paper.

Flanagan and Qadeer [Flanagan and Qadeer 2002] use indexed predicates to synthesize loop invariants for sequential software programs that involve unbounded arrays. They also provide heuristics to extract some of the predicates from the program text automatically. The heuristics are specific to loops in sequential software and not suited for verifying more general unbounded systems that we handle in this paper. In this work, we explore formal properties of this formulation and apply it for verifying distributed systems. In a recent work [Lahiri and Bryant 2004b], we provide a weakest precondition transformer [Dijkstra 1975] based heuristic for discovering most of the predicates for many of the systems that we consider in this paper. We have proved some completeness results for the predicate discovery scheme in the first authors' thesis [Lahiri 2004].

2. NOTATION

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

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 $\mathtt{x} \in \mathcal{B}$.

Figure 1 displays the syntax of the Logic of Counter arithmetic with Lambda expressions and Uninterpreted functions (CLU), a fragment of first-order logic extended with equality, inequality, and counters. An *expression* in CLU can evaluate to truth values (*bool-expr*), integers (*int-expr*), functions (*function-expr*) or predicates (*predicate-expr*). Notice that we only allow restricted arithmetic on terms, namely that of addition or subtraction by constants. Notice that we restrict the parameters to a lambda expression to be integers, and not function or predicate expressions. There is no way in our logic to express any form of iteration or recursion.

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

Let 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 v defines an interpretation of the symbols in $\mathcal{A}$, which we denote $\langle v \rangle_{\sigma_{\mathcal{B}}}$.

$$\begin{array}{rcl}
\textit{bool-expr} & ::= & \mathbf{true} \mid \mathbf{false} \mid \textit{bool-symbol} \\
 & & \mid \neg\textit{bool-expr} \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \\
 & & \mid (\textit{int-expr} = \textit{int-expr}) \mid (\textit{int-expr} < \textit{int-expr}) \\
 & & \mid \textit{predicate-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{int-expr} & ::= & \textit{lambda-var} \mid \textit{int-symbol} \\
 & & \mid \textit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr}) \\
 & & \mid \textit{int-expr} + \textit{int-constant} \\
 & & \mid \textit{function-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{predicate-expr} & ::= & \textit{predicate-symbol} \mid \lambda\, \textit{lambda-var}, \ldots, \textit{lambda-var} \,.\, \textit{bool-expr} \\
\textit{function-expr} & ::= & \textit{function-symbol} \mid \lambda\, \textit{lambda-var}, \ldots, \textit{lambda-var} \,.\, \textit{int-expr}
\end{array}$$

Fig. 1. **CLU Expression Syntax.** Expressions can denote computations of Boolean values, integers, or functions yielding Boolean values or integers.

That is, $\langle v \rangle_{\sigma_{\mathcal{B}}}$ 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* π 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 $\pi_{\mathtt{x}} \in E(\mathcal{B})$. We assume that the expression $\pi_{\mathtt{x}}$ has the same type as the symbol $\mathtt{x} \in \mathcal{A}$. For an expression $\psi \in E(\mathcal{A} \cup \mathcal{C})$, we let $\psi\,[\pi/\mathcal{A}]$ denote the expression $\psi' \in E(\mathcal{B} \cup \mathcal{C})$ resulting when we replace each occurrence of each symbol $\mathtt{x} \in \mathcal{A}$ with the expression $\pi_{\mathtt{x}}$. These replacements are all performed simultaneously.

PROPOSITION 2.1. *Let ψ be an expression in $E(\mathcal{A} \cup \mathcal{C})$ and π be a substitution having $\pi_{\mathtt{x}} \in E(\mathcal{B})$ for each $\mathtt{x} \in \mathcal{A}$. For interpretations $\sigma_{\mathcal{B}}$ and $\sigma_{\mathcal{C}}$, if we let $\sigma_{\mathcal{A}}$ be the interpretation defined as $\sigma_{\mathcal{A}} = \langle \pi \rangle_{\sigma_{\mathcal{B}}}$, then $\langle \psi \rangle_{\sigma_{\mathcal{A}} \cdot \sigma_{\mathcal{C}}} = \langle \psi\,[\pi/\mathcal{A}] \rangle_{\sigma_{\mathcal{B}} \cdot \sigma_{\mathcal{C}}}$.*

This proposition captures a fundamental relation between syntactic substitution and expression evaluation, sometimes referred to as *referential transparency*. We can interchangeably use a subexpression $\pi_{\mathtt{x}}$ or the result of evaluating this subexpression $\sigma_{\mathcal{A}}(\mathtt{x})$ in evaluating a formula containing this subexpression.

3. 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 verifier 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 q^0 and a *next-state* expression set δ. The initial state consists of an expression for each state element, with the initial value of state element 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 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 F, i.e. $\mathcal{V} = \{\mathtt{F}\}$. An input symbol i determines which element of F is updated. Initially, F is the identify function:

$$q^0_{\mathtt{F}} = \lambda\, u \,.\, u.$$

On each step, the value of the function for argument 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 first one evaluates to true and the third otherwise. For the above example, $\mathcal{J} = \{\}$ and $\mathcal{I} = \{\mathtt{i}\}$.

3.1 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 $N_C : \mathscr{P}(\Sigma_{\mathcal{V}}) \rightarrow \mathscr{P}(\Sigma_{\mathcal{V}})$. The initial state set is defined as:

$$Q^0_C \doteq \{\langle q^0 \rangle_{\sigma_{\mathcal{J}}} | \sigma_{\mathcal{J}} \in \Sigma_{\mathcal{J}}\},$$

i.e., the set of all possible valuations of the initial state expressions. The next-state function N_C is defined for a single state s as:

$$N_C(s) \doteq \{\langle \delta \rangle_{s \cdot \sigma_{\mathcal{I}}} | \sigma_{\mathcal{I}} \in \Sigma_{\mathcal{I}}\},$$

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 $s_0 \in Q^0_C$, $s_n = s$, and $s_{i+1} \in N_C(s_i)$ for all values of i such that $0 \le i < n$. 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 $f(u+1) \ge f(u) \ge u$ for all u and $f(u) = u$ for all but finitely many values of u.

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

Table I. **Example abstract state sets and their concretizations** Abstract state elements are represented by their interpretations of $\mathtt{p}$ and $\mathtt{q}$.

4. PREDICATE ABSTRACTION WITH INDEXED PREDICATES

We use *indexed* 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 $\phi_{\mathtt{p}} \doteq \mathtt{F(x)} \geq 0$ 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 I. 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.

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:

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

$$= \bigcup_{\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}} \left\{\langle\phi\rangle_{s\cdot\sigma_{\mathcal{X}}}\right\} \quad (2)$$

Note that (2) is simply a restatement of (1) using set union notation.

Since there are multiple interpretations $\sigma_{\mathcal{X}}$, a single concrete state will generally map to multiple abstract states. Figure 2 illustrates this fact. The abstraction function α maps a single concrete state s to a set of abstract states — each abstract state ($\langle\phi\rangle_{s\cdot\sigma_{\mathcal{X}}}$) resulting from some interpretation $\sigma_{\mathcal{X}}$. This feature is not found in most uses of predicate abstraction, but it is the key idea for handling indexed predicates.

Working with our example system, consider the concrete state given by the function $\lambda\, u\, .\, u$, in Figure 3. 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

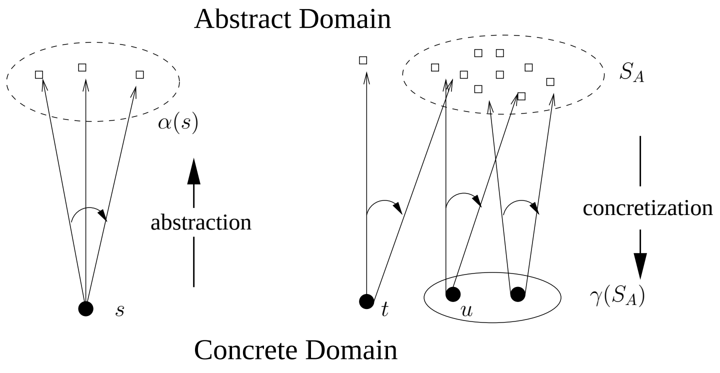

Fig. 2. **Abstraction and Concretization.**

characterized by the formula $\mathtt{p} \Leftrightarrow \mathtt{q}$.

We then extend the abstraction function to apply to sets of concrete states in the usual way:

$$\alpha(S_C) \doteq \bigcup_{s \in S_C} \alpha(s). \tag{3}$$

$$= \bigcup_{\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}} \bigcup_{s \in S_C} \{\langle\phi\rangle_{s \cdot \sigma_{\mathcal{X}}}\} \tag{4}$$

Note that (4) follows by combining (2) with (3), and then reordering the unions.

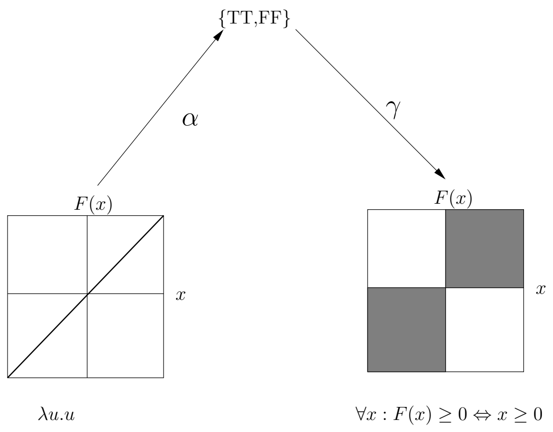

Fig. 3. **Abstraction and Concretization for the initial state for the example.**

PROPOSITION 4.1. *For any pair of concrete state sets S_C and T_C:*

(1) *If $S_C \subseteq T_C$, then $\alpha(S_C) \subseteq \alpha(T_C)$.*
(2) *$\alpha(S_C) \cup \alpha(T_C) = \alpha(S_C \cup T_C)$.*

These properties follow directly from the way we extended α from a single concrete state to a set of concrete states.

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

$$\gamma(S_A) \quad \doteq \quad \left\{s | \forall \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} : \langle \phi \rangle_{s \cdot \sigma_{\mathcal{X}}} \in S_A \right\} \tag{5}$$

Consider the Figure 2, where a set of abstract states S_A has been concretized to a set of concrete states $\gamma(S_A)$. It shows a concrete state t that is not included in $\gamma(S_A)$ because one of the states it abstracts to lies outside S_A. On the other hand, the concrete state u is contained in $\gamma(S_A)$ because $\alpha(u) \subseteq S_A$. One can provide an alternate definition of γ as follows:

$$\gamma(S_A) \quad \doteq \quad \{s | \alpha(s) \subseteq S_A\} \tag{6}$$

The universal quantifier in the definition of γ 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.

PROPOSITION 4.2. *For any pair of abstract state sets S_A and T_A:*

(*1*) *If $S_A \subseteq T_A$, then $\gamma(S_A) \subseteq \gamma(T_A)$.*
(*2*) *$\gamma(S_A) \cup \gamma(T_A) \subseteq \gamma(S_A \cup T_A)$.*

The first property follows from (5), while the second follows from the first.

Consider our example system with predicates $\phi_{\mathtt{p}}$ and $\phi_{\mathtt{q}}$. Table I 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 `x` to be either always negative or always nonnegative. 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 $\{\text{TF}\}$ and $\{\text{TT}\}$ have empty concretizations, but $\{\text{TF}, \text{TT}\}$ concretizes to the set of all nonnegative functions.

THEOREM 4.3. *The functions (α, γ) form a Galois connection, i.e., for any sets of concrete states S_C and abstract states S_A:*

$$\alpha(S_C) \subseteq S_A \;\Leftrightarrow\; S_C \subseteq \gamma(S_A) \tag{7}$$

PROOF. (This is one of several logically equivalent formulations of a Galois connection [Cousot and Cousot 1977].) The proof follows by observing that both the left and the right-hand sides of (7) hold precisely when for every $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ and every $s \in S_C$, we have $\langle \phi \rangle_{s \cdot \sigma_{\mathcal{X}}} \in S_A$. Let us prove the two directions:

(1) *If* : Let $\alpha(S_C) \subseteq S_A$. By the definition of α in (1), this implies that for every $s \in S_C$ and for interpretation $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$, $\langle \phi \rangle_{s \cdot \sigma_{\mathcal{X}}} \in S_A$. By the definition of γ in (5),

$\gamma(S_A)$ contains precisely those concrete states s' for which $\langle\phi\rangle_{s'\cdot\sigma_{\mathcal{X}}} \in S_A$, for every interpretation $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$. Thus, for every $s \in S_C$, $s \in \gamma(S_A)$ and consequently, $S_C \subseteq \gamma(S_A)$.

(2) *Only if* : Let $S_C \subseteq \gamma(S_A)$. By (5), for every $s \in S_C$, $\langle\phi\rangle_{s\cdot\sigma_{\mathcal{X}}} \in S_A$, for every interpretation $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$. By the definition of α in (1), $\alpha(s) \in S_A$. Further, by extending α for the entire set S_C by (3), we get $\alpha(S_C) \subseteq S_A$.

□

Alternately, the functions (α, γ) form a Galois connection if they satisfy the following properties for any sets of concrete states S_C and abstract states S_A:

$$S_C \subseteq \gamma(\alpha(S_C)). \tag{8}$$

$$\alpha(\gamma(S_A)) \subseteq S_A. \tag{9}$$

These properties can be derived from (7). Similarly, (7) can be derived from (8) and (9). The containment relation in both (8) and (9) 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 $f(u) \geq 0 \Leftrightarrow u \geq 0$, for any argument u. This is clearly demonstrated in Fig 3. On the other hand, consider the set of abstract states represented by $\mathtt{p} \wedge \mathtt{q}$. This set of abstract states has an empty concretization (see Table I), and thereby satisfies $\alpha(\gamma(S_A)) \subset S_A$.

5. 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* [Cousot and Cousot 1977] of the concrete system:

(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 (with a simulation relation defined by α): $\alpha(N_C(S_C)) \subseteq N_A(\alpha(S_C))$.

THEOREM 5.1. *N_A provides an* abstract interpretation *of the concrete transition system N_C.*

PROOF. Let us prove the three properties mentioned above:

(1) This follows from the definition of N_A and the fact that $\gamma(\emptyset) = \emptyset$, $N_C(\emptyset) = \emptyset$ and $\alpha(\emptyset) = \emptyset$.

(2) By the definition of N_A, and using the fact that γ, α and N_C are monotonic. N_C is monotonic since it distributes over the elements of a set of concrete states, i.e. $N_C(S_C) = \bigcup_{s \in S_C} N_C(s)$.

(3) From (8), we know that $S_C \subseteq \gamma(\alpha(S_C))$. By the monotonicity of N_C, $N_C(S_C) \subseteq N_C(\gamma(\alpha(S_C)))$. Since α is monotonic, we have $\alpha(N_C(S_C)) \subseteq \alpha(N_C(\gamma(\alpha(S_C))))$. Now applying the definition of N_A, we get the desired result.

□

6. REACHABILITY ANALYSIS

Given the set of initial abstract states Q_A^0, and the abstract transformer N_A, we can define the set of states R_A^i reachable after i steps of the reachability analysis as:

$$R_A^0 = Q_A^0 \tag{10}$$

$$R_A^{i+1} = R_A^i \cup N_A(R_A^i) \tag{11}$$

$$= R_A^i \cup \bigcup_{s \in \gamma(R_A^i)} \; \bigcup_{t \in N_C(s)} \alpha(t) \tag{12}$$

PROPOSITION 6.1. *If s is a reachable state in the concrete system such that $depth(s) \leq n$, then $\alpha(s) \subseteq R_A^n$.*

PROOF. We prove this by induction on n. For $n = 0$, the only concrete states having depth 0 are those in Q_C^0, and by (10), these states are all included in R_A^0.

For a state t having depth $k < n$, our induction hypothesis shows that $\alpha(t) \subseteq R_A^{n-1}$. Since $R_A^{n-1} \subseteq R_A^n$, we therefore have $\alpha(t) \subseteq R_A^n$.

Otherwise, suppose state t has depth n. Then there must be some state s having depth $n-1$ such that $t \in N_C(s)$. By the induction hypothesis, we must have $\alpha(s) \subseteq R_A^{n-1}$. By (8), we have $s \in \gamma(\alpha(s))$, and Proposition 4.2 then implies that $s \in \gamma(R_A^{n-1})$. By (12), we can therefore see that $\alpha(t) \subseteq R_A^n$. □

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.

PROPOSITION 6.2. *The abstract system computes an overapproximation of the set of reachable concrete states, i.e.,*

$$\alpha(R_C) \subseteq R_A \tag{13}$$

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 on the abstract system.

Remark 6.3. 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 (12). 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:

(1) 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 $f(u) \geq 0 \Leftrightarrow u \geq 0$. This is illustrated in Fig 3.

(2) 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 $u \geq 0$. This is illustrated in Fig 4. Applying the abstraction function, we get R_A^1 characterized by the formula $\mathtt{p} \vee \neg\mathtt{q}$ (see Table I.)

(3) 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.

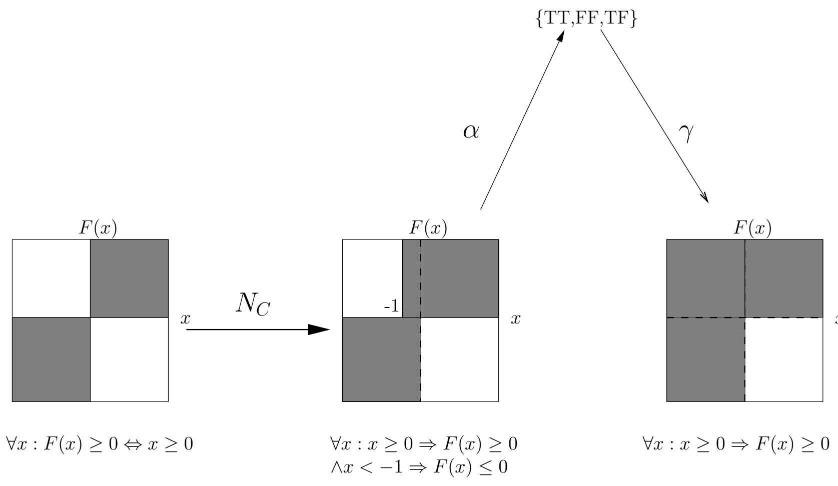
{TT,FF,TF}
α
γ
$F(x)$
$F(x)$
$F(x)$
x
N_C
-1
x
x
$\forall x : F(x) \geq 0 \Leftrightarrow x \geq 0$
$\forall x : x \geq 0 \Rightarrow F(x) \geq 0$
$\wedge x < -1 \Rightarrow F(x) \leq 0$
$\forall x : x \geq 0 \Rightarrow F(x) \geq 0$

Fig. 4. **Reachability after 1 iteration for the example.**

7. VERIFYING SAFETY PROPERTIES

A Boolean formula $\psi \in E(\mathcal{P})$ can be viewed as defining a *property* of the abstract state space. Such a 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 ϕ as a substitution, we have $\psi^* \doteq \psi\,[\phi/\mathcal{P}]$.

PROPOSITION 7.1. *For any formula $\psi \in E(\mathcal{P})$, any concrete state s, and interpretation $\sigma_\mathcal{X} \in \Sigma_\mathcal{X}$, if $\sigma_\mathcal{P} = \langle\phi\rangle_{s \cdot \sigma_\mathcal{X}}$, then $\langle\psi^*\rangle_{s \cdot \sigma_\mathcal{X}} = \langle\psi\rangle_{\sigma_\mathcal{P}}$.*

This is a particular instance of Proposition 2.1.

We can view the formula ψ^* as defining a property $\forall \mathcal{X} \psi^*$ of the concrete state space. This property is said to hold 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 is said to hold for the concrete system when $\forall \mathcal{X} \psi^*(s)$ holds for every reachable concrete state $s \in R_C$.

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 as a property of state variable $\mathtt{F}$ that $\forall \mathtt{x} : \mathtt{x} \geq 0 \Rightarrow \mathtt{F}(\mathtt{x}) \geq 0$.

PROPOSITION 7.2. *Property $\forall \mathcal{X} \psi^*(s)$ holds for concrete state s if and only if $\langle \psi \rangle_{\sigma_{\mathcal{P}}} = \mathbf{true}$ for every $\sigma_{\mathcal{P}} \in \alpha(s)$.*

This property follows from the definition of α (Equation 1) and Proposition 7.1.

Alternately, a Boolean formula $\psi \in E(\mathcal{P})$ can also be viewed as characterizing a set of abstract states $\langle \psi \rangle \doteq \{\sigma_{\mathcal{P}} \mid \langle \psi \rangle_{\sigma_{\mathcal{P}}} = \mathbf{true}\}$. Similarly, we can interpret the formula $\forall \mathcal{X} \psi^*$ as characterizing the set of concrete states $\langle \forall \mathcal{X} \psi^* \rangle \doteq \{s \mid \langle \forall \mathcal{X} \psi^* \rangle_s = \mathbf{true}\}$.

PROPOSITION 7.3. *If $S_C \doteq \langle \forall \mathcal{X} \psi^* \rangle$ and $S_A \doteq \langle \psi \rangle$, then $S_C = \gamma(S_A)$.*

PROOF. Expanding the definition of S_C, we get

$$
\begin{aligned}
S_C \ &= \ \{s \mid \forall \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} \ : \ \langle \psi^* \rangle_{s \cdot \sigma_{\mathcal{X}}} = \mathbf{true}\} && (14)\\
&= \ \{s \mid \forall \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} \ : \ \langle \psi \rangle_{\sigma_{\mathcal{P}}} = \mathbf{true} \text{ where } \sigma_{\mathcal{P}} \doteq \langle \phi \rangle_{s \cdot \sigma_{\mathcal{X}}}\} && (15)\\
&= \ \{s \mid \forall \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} \ : \ \langle \phi \rangle_{s \cdot \sigma_{\mathcal{X}}} \in S_A\} && (16)
\end{aligned}
$$

Observe that (15) follows from (14) by expanding the definition of ψ^*, and (16) follows from (15) by using Proposition 7.1. □

The purpose of indexed predicate abstraction is to provide a way to verify that a property $\forall \mathcal{X} \psi^*(s)$ holds for the concrete system based on the set of reachable abstract states.

THEOREM 7.4. *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.*

PROOF. Consider an arbitrary concrete state $s \in R_C$ and an arbitrary interpretation $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$. If we let $\sigma_{\mathcal{P}} = \langle \phi \rangle_{s \cdot \sigma_{\mathcal{X}}}$, then by the definition of α (Equation 1), we must have $\sigma_{\mathcal{P}} \in \alpha(s)$. By Propositions 4.1 and 6.2, we therefore have

$$\sigma_{\mathcal{P}} \quad \in \quad \alpha(s) \quad \subseteq \quad \alpha(R_C) \quad \subseteq \quad R_A$$

By the premise of the theorem we have $\langle \psi \rangle_{\sigma_{\mathcal{P}}} = \mathbf{true}$, and by Proposition 7.1, we have $\langle \psi^* \rangle_{s \cdot \sigma_{\mathcal{X}}} = \langle \psi \rangle_{\sigma_{\mathcal{P}}} = \mathbf{true}$. This is precisely the condition required for the property $\forall \mathcal{X} \psi^*$ to hold for the concrete system. □

Thus, the abstract reachability analysis on our example system does indeed prove the property that any value f of state variable $\mathtt{F}$ satisfies $\forall x : x \geq 0 \Rightarrow f(x) \geq 0$.

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.

For example, any reachable state f of our example system satisfies $\forall x : f(x) < 0 \Rightarrow f(-x) \geq -x$, but our reachability analysis cannot show this.

We can, however, precisely characterize the class of properties for which this form of predicate analysis is both sound and complete. 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 every pair of concrete states (s, t), such that $t \in N_C(s)$, if $\forall \mathcal{X} \psi^*(s)$ holds, then so does $\forall \mathcal{X} \psi^*(t)$.

PROPOSITION 7.5. *If $\forall \mathcal{X} \psi^*$ is inductive, then $\forall \mathcal{X} \psi^*$ holds for the concrete system.*

This proposition follows by induction on the state sequence leading to each reachable state.

Let ρ_A be a formula that exactly characterizes the set of reachable abstract states. That is, $\langle \rho_A \rangle = R_A$.

LEMMA 7.6. *$\forall \mathcal{X} \rho_A^*$ is inductive.*

PROOF. By definition, $\langle \rho_A \rangle_{\sigma_\mathcal{P}} = \mathbf{true}$ if and only if $\sigma_\mathcal{P} \in R_A$, and so by Proposition 7.2, $\forall \mathcal{X} \rho_A^*(s)$ holds for concrete state s if and only if $\alpha(s) \subseteq R_A$.

We can see that the first requirement is satisfied for any $s \in Q_C^0$, since $\alpha(s) \subseteq \alpha(Q_C^0) \subseteq R_A$ and therefore $\forall \mathcal{X} \rho_A^*(s)$ holds by Proposition 7.2.

Now suppose there is a state $t \in N_C(s)$ and $\forall \mathcal{X} \rho_A^*(s)$ holds. Then we must have $\alpha(s) \subseteq R_A^i$ for some $i \geq 0$. From (8), we have $s \in \gamma(\alpha(s)) \subseteq \gamma(R_A^i)$, and therefore, by (12), $\alpha(t) \subseteq R_A^{i+1} \subseteq R_A$. Thus, the second requirement is satisfied. □

LEMMA 7.7. *If $\forall \mathcal{X} \psi^*$ is inductive, then ψ holds for the abstract system.*

PROOF. We will prove by induction on i that $\langle \psi \rangle_{\sigma_\mathcal{P}} = \mathbf{true}$ for every $\sigma_\mathcal{P} \in R_A^i$. From the definition of R_A, it then follows that $\langle \psi \rangle_{\sigma_\mathcal{P}} = \mathbf{true}$ for every $\sigma_\mathcal{P} \in R_A$, and therefore ψ holds for the abstract system.

For the case of $i = 0$, (10) indicates that $R_A^0 = \alpha(Q_C^0)$. Thus, by the definition of α (Equation 1) for every $\sigma_\mathcal{P} \in R_A^0$, there must be a state s and an interpretation $\sigma_\mathcal{X} \in \Sigma_\mathcal{X}$ such that $\sigma_\mathcal{P} = \langle \phi \rangle_{s \cdot \sigma_\mathcal{X}}$. By the first property of an inductive predicate and by Proposition 7.1, we have $\langle \psi \rangle_{\sigma_\mathcal{P}} = \langle \psi^* \rangle_{s \cdot \sigma_\mathcal{X}} = \mathbf{true}$.

Now suppose that $\langle \psi \rangle_{\sigma_\mathcal{P}} = \mathbf{true}$ for all $\sigma_\mathcal{P} \in R_A^i$. Consider an element $\tau_\mathcal{P} \in R_A^{i+1}$. If $\tau_\mathcal{P} \in R_A^i$, then our induction hypothesis shows that $\langle \psi \rangle_{\tau_\mathcal{P}} = \mathbf{true}$. Otherwise, by (12), and the definitions of α (Equation 1), the transition relation N_C, and γ (Equation 5), there must be concrete states s and t satisfying:

(1) $\tau_\mathcal{P} \in \alpha(t)$. That is, $\tau_\mathcal{P} = \langle \phi \rangle_{t \cdot \tau_\mathcal{X}}$ for some $\tau_\mathcal{X} \in \Sigma_\mathcal{X}$.
(2) $t \in N_C(s)$.
(3) $s \in \gamma(R_A^i)$. That is, for all $\sigma_\mathcal{X} \in \Sigma_\mathcal{X}$, if $\sigma_\mathcal{P} \doteq \langle \phi \rangle_{s \cdot \sigma_\mathcal{X}}$, then $\sigma_\mathcal{P} \in R_A^i$.

By Proposition 7.1 we have $\langle \psi^* \rangle_{s \cdot \sigma_{\mathcal{X}}} = \langle \psi \rangle_{\sigma_{\mathcal{P}}} = \mathbf{true}$, and therefore $\forall \mathcal{X} \psi^*(s)$ holds. By the second property of an inductive predicate, $\forall \mathcal{X} \psi^*(t)$ must also hold. Applying Proposition 7.1 once again, we therefore have $\langle \psi \rangle_{\tau_{\mathcal{P}}} = \langle \psi^* \rangle_{t \cdot \tau_{\mathcal{X}}} = \mathbf{true}$. This completes our induction. □

This lemma simply shows that if we present our predicate abstraction engine with a fully formed induction hypothesis, then it will be able to perform the induction proof. But, it has important consequences.

For a formula $\psi \in E(\mathcal{P})$ and a predicate set ϕ, the property $\forall \mathcal{X} \psi^*$ is said to *have an induction proof over* ϕ 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 7.8. *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 ϕ.*

PROOF. Suppose there is a formula χ such that $\forall \mathcal{X} \chi^*$ is inductive. Then by Lemma 7.7, we know that χ holds in the abstract system, and when $\chi \Rightarrow \psi$, we can infer that ψ holds in the abstract system.

On the other hand, suppose that ψ holds in the abstract system. Then the formula ρ_A (characterizing the set of all reachable abstract states) satisfies $\rho_A \Rightarrow \psi$ and $\forall \mathcal{X} \rho_A^*$ is inductive. Hence $\forall \mathcal{X} \psi^*$ has an induction proof over ϕ. □

This theorem precisely characterizes the capability of our formulation of predicate abstraction — it can prove any property that can be strengthened into an induction hypothesis using some combination 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 out of which the predicate abstraction engine could construct a universally quantified induction hypothesis.

COROLLARY 7.9. *The property $\forall \mathcal{X} \rho_A^*$ is the strongest inductive invariant for the concrete system of the form $\forall \mathcal{X} \chi^*$, where $\chi \in E(\mathcal{P})$. Alternately, for any other inductive property $\forall \mathcal{X} \chi^*$, where $\chi \in E(\mathcal{P})$, $\forall \mathcal{X} \rho_A^* \Rightarrow \forall \mathcal{X} \chi^*$.*

PROOF. The proof follows easily from Theorem 7.8, the fact that $\rho_A \Rightarrow \chi$ whenever χ is a property of the abstract state space, Proposition 7.3 and Proposition 4.2. □

Remark 7.10. To fully automate the process of generating invariants, we need to further discover the predicates automatically. Other predicate abstraction tools [Ball et al. 2001; Henzinger et al. 2002; Chaki et al. 2003; Das and Dill 2002] generate new predicates based on ruling out spurious counterexample traces from the abstract model. This approach cannot be used directly in our context, since our 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. In this paper, we do not address the issue of discovering the indexed predicates: we provide a syntactic heuristic based on the weakest precondition transformer in a separate work [Lahiri and Bryant 2004b].

8. QUANTIFIER INSTANTIATION

For many subsets of first-order logic, there is no complete method for handling the universal quantifier introduced in function γ (Equation 5). For example, in a logic with uninterpreted functions and equality, determining whether a universally quantified formula is satisfiable is undecidable [Börger et al. 1997]. Instead, we concretize abstract states by considering some limited subset of the interpretations of the index symbols, each of which is defined 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. Details of the quantifier instantiation heuristic can be found in an earlier work [Lahiri et al. 2002]. These subexpressions can contain symbols in $\mathcal{V}$, $\mathcal{X}$, and $\mathcal{I}$. These instantiated versions of the formulas enable the verifier to detect specific cases where the predicates can be applied.

More precisely, let π 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 $\phi_{\mathtt{p}}[\pi/\mathcal{X}]$ 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 define the concretization function γ_{Π} as:

$$\gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}) \quad \doteq \quad \left\{ s | \forall \pi \in \Pi : \langle \boldsymbol{\phi}[\boldsymbol{\pi}/\mathcal{X}] \rangle_{s \cdot \sigma_{\mathcal{X}} \cdot \sigma_{\mathcal{I}}} \in S_A \right\} \tag{17}$$

PROPOSITION 8.1. *For any abstract state set S_A and interpretations $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ and $\sigma_{\mathcal{I}} \in \Sigma_{\mathcal{I}}$:*

(1) $\gamma(S_A) \subseteq \gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}})$ *for any set of substitutions* Π.

(2) $\gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}) \subseteq \gamma_{\Pi'}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}})$ *for any pair of substitution sets* Π *and* Π' *satisfying* $\Pi \supseteq \Pi'$.

(3) *For any abstract state set T_A, if $S_A \subseteq T_A$, then $\gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}) \subseteq \gamma_{\Pi}(T_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}})$, for any set of substitutions* Π.

These properties follow directly from the definitions of γ and γ_{Π} and Proposition 2.1.

PROPOSITION 8.2. *For any concrete state set S_C, set of substitutions Π, and interpretations $\sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}$ and $\sigma_{\mathcal{I}} \in \Sigma_{\mathcal{I}}$:*

$$S_C \quad \subseteq \quad \gamma_{\Pi}(\alpha(S_C), \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}). \tag{18}$$

This property follows directly from Theorem 4.3 and Proposition 8.1. It shows that for a given interpretation $\sigma_{\mathcal{X}}$ and $\sigma_{\mathcal{I}}$, the functions (α, γ_{Π}) satisfy one of the properties of a Galois connection (Equation 8), but they need not satisfy the other (Equation 9). For example, when $\Pi = \emptyset$, the quantified condition of (17) becomes vacuous, and hence $\gamma_{\Pi}(S_A, \sigma_{\mathcal{X}}, \sigma_{\mathcal{I}}) = \Sigma_{\mathcal{V}}$.

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:

$$N_\Pi(S_A) \;=\; \{\langle\phi\,[\delta/\mathcal{V}]\rangle_{s\cdot\sigma_\mathcal{X}\cdot\sigma_\mathcal{I}} \,|\sigma_\mathcal{X} \in \Sigma_\mathcal{X}, \sigma_\mathcal{I} \in \Sigma_\mathcal{I}, s \in \gamma_\Pi(S_A, \sigma_\mathcal{X}, \sigma_\mathcal{I})\} \tag{19}$$

$$= \bigcup_{\sigma_\mathcal{X}\in\Sigma_\mathcal{X}} \; \bigcup_{\sigma_\mathcal{I}\in\Sigma_\mathcal{I}} \; \bigcup_{s\in\gamma_\pi(S_A,\sigma_\mathcal{X},\sigma_\mathcal{I})} \{\langle\phi\,[\delta/\mathcal{V}]\rangle_{s\cdot\sigma_\mathcal{X}\cdot\sigma_\mathcal{I}}\} \tag{20}$$

Observe in this equation that $\phi_\mathrm{p}\,[\delta/\mathcal{V}]$ is an expression describing the evaluation of predicate ϕ_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:

(1) $N_\Pi(\emptyset) = \emptyset$, if Π is nonempty.
(2) N_Π is monotonic: This follows from the definition of N_Π in (20) and Proposition 8.1.
(3) $\alpha(N_C(S_C)) \subseteq N_\Pi(\alpha(S_C))$: This follows from the fact that $\alpha(N_C(S_C)) \subseteq N_A(\alpha(S_C))$ and $N_A(S_A) \subseteq N_\Pi(S_A)$.

We can therefore perform reachability analysis:

$$R^0_\Pi \;=\; Q^0_A \tag{21}$$

$$R^{i+1}_\Pi \;=\; R^i_\Pi \cup N_\Pi(R^i_\Pi) \tag{22}$$

These iterations will converge to a set R_Π.

PROPOSITION 8.3.

(*1*) *$R_A \subseteq R_\Pi$ for any set of substitutions Π.*
(*2*) *$R_\Pi \subseteq R_{\Pi'}$ for any pair of substitution sets Π and Π' satisfying $\Pi \supseteq \Pi'$.*

To see the first property, consider the following way of expressing the equation for R^{i+1}_A (12) using the alternative equation for α (4), and rearranging the order of the union operations:

$$R^{i+1}_A \;=\; R^i_A \,\cup \bigcup_{\sigma_\mathcal{X}\in\Sigma_\mathcal{X}} \; \bigcup_{\sigma_\mathcal{I}\in\Sigma_\mathcal{I}} \; \bigcup_{s\in\gamma(R^i_A)} \{\langle\phi\,[\delta/\mathcal{V}]\rangle_{s\cdot\sigma_\mathcal{X}\cdot\sigma_\mathcal{I}}\}$$

The property then follows by Proposition 8.1, using induction on i. The second property also follows by Proposition 8.1 using induction on i.

THEOREM 8.4. *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.*

PROOF. Since $\langle\psi\rangle_{\sigma_\mathcal{P}} = \mathbf{true}$ for every $\sigma_\mathcal{P} \in R_\Pi$ and $R_A \subseteq R_\Pi$ (by Proposition 8.3), $\langle\psi\rangle_{\sigma_\mathcal{P}} = \mathbf{true}$ for every $\sigma_\mathcal{P} \in R_A$. Hence by Theorem 7.4, the 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 quantifier instantiation.

9. 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 a quantified CLU formula. We will then see how this can be converted into a problem of finding satisfying solutions to a Boolean formula.

On each step, we generate a Boolean formula ρ^i_Π, that characterizes R^i_Π. That is $\langle \rho^i_\Pi \rangle = R^i_\Pi$. The formulas directly encode the approximate reachability computations of (21) and (22).

Observe that by composing the predicate expressions with the initial state expressions, $\phi\left[q^0/\mathcal{V}\right]$, 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

$$\rho^0_\Pi(\mathcal{P}) \doteq \exists \mathcal{X} \exists \mathcal{J} \bigwedge_{\mathtt{p} \in \mathcal{P}} \mathtt{p} \Leftrightarrow \phi_{\mathtt{p}}\left[q^0/\mathcal{V}\right] \tag{23}$$

PROPOSITION 9.1. $\langle \rho^0_\Pi \rangle = Q^0_A$

Let us understand the expression ρ^0_Π by showing why it represents Q^0_A. Expanding the definition of Q^0_A, we get:

$$Q^0_A = \bigcup_{\sigma_\mathcal{X} \in \Sigma_\mathcal{X}} \bigcup_{s \in Q^0_C} \{\langle \phi \rangle_{s \cdot \sigma_\mathcal{X}}\} \tag{24}$$

Again, $Q^0_C = \bigcup_{\sigma_\mathcal{J} \in \Sigma_\mathcal{J}} \left\{\langle q^0 \rangle_{\sigma_\mathcal{J}}\right\}$. Using Proposition 2.1, we can rewrite (24) as:

$$Q^0_A = \bigcup_{\sigma_\mathcal{X} \in \Sigma_\mathcal{X}} \bigcup_{\sigma_\mathcal{J} \in \Sigma_\mathcal{J}} \left\{\langle \phi\left[q^0/\mathcal{V}\right] \rangle_{\sigma_\mathcal{J} \cdot \sigma_\mathcal{X}}\right\} \tag{25}$$

To generate a formula for the next-state computation, we first generate a formula for $\gamma_\pi(R^i_\Pi, \sigma_\mathcal{X}, \sigma_\mathcal{I})$ by forming a conjunction over each substitution in Π, where we compose the current-state formula with the predicate expressions and with each substitution π: $\bigwedge_{\pi \in \Pi} \left(\rho^i_\Pi\left[\phi/\mathcal{P}\right]\right)\left[\pi/\mathcal{X}\right]$.

The formula for the next-state computation combines the alternate definition of N_Π (20) and the formula for γ_Π above:

$$\begin{aligned} \rho^{i+1}_\Pi(\mathcal{P}) \doteq{}& \rho^i_\Pi(\mathcal{P}) \vee \\ & \exists \mathcal{V} \exists \mathcal{X} \exists \mathcal{I} \left(\bigwedge_{\pi \in \Pi} \left(\rho^i_\Pi\left[\phi/\mathcal{P}\right]\right)\left[\pi/\mathcal{X}\right] \;\wedge\; \bigwedge_{\mathtt{p} \in \mathcal{P}} \mathtt{p} \Leftrightarrow \phi_{\mathtt{p}}\left[\delta/\mathcal{V}\right] \right). \end{aligned} \tag{26}$$

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

PROPOSITION 9.2. $\langle \rho_\Pi^{i+1} \rangle = R_\Pi^{i+1}$

Let us see how this symbolic formulation would perform reachability analysis for our example system. Recall that our system has two predicates $\phi_\mathtt{p} \doteq \mathtt{F(x)} \geq 0$ and $\phi_\mathtt{q} \doteq \mathtt{x} \geq 0$. In the initial state, F is the function $\lambda\, u\,.\,u$, and therefore $\phi_\mathtt{p}\left[q^0/\mathcal{V}\right]$ simply becomes $\mathtt{x} \geq 0$. Equation (23) 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 π and π' with $\pi_\mathtt{x} = \mathtt{x}$ and $\pi'_\mathtt{x} = \mathtt{i}+1$. For $\rho_\Pi^0(\mathtt{p},\mathtt{q}) = \mathtt{p} \Leftrightarrow \mathtt{q}$, the left-hand term of (26) instantiates to $(\mathtt{F(x)} \geq 0 \Leftrightarrow \mathtt{x} \geq 0) \wedge (\mathtt{F(i+1)} \geq 0 \Leftrightarrow \mathtt{i+1} \geq 0)$. Substituting $\lambda\, u\,.\mathit{ITE}(u = \mathtt{i},\, \mathtt{F(i+1)},\, \mathtt{F}(u))$ for F in $\phi_\mathtt{p}$ gives $(\mathtt{x{=}i} \wedge \mathtt{F(i+1)} \geq 0) \vee (\mathtt{x} \neq \mathtt{i} \wedge \mathtt{F(x)} \geq 0)$. The quantified portion of (26) for $\rho_\Pi^1(\mathtt{p},\mathtt{q})$ then becomes:

$$\exists\, \mathtt{F},\mathtt{x},\mathtt{i} : \left(\begin{array}{l} \quad \mathtt{F(x)} \geq 0 \Leftrightarrow \mathtt{x} \geq 0 \;\; \wedge \;\; \mathtt{F(i+1)} \geq 0 \Leftrightarrow \mathtt{i+1} \geq 0 \\ \wedge\;\; \mathtt{p} \Leftrightarrow [(\mathtt{x{=}i} \wedge \mathtt{F(i+1)} \geq 0) \vee (\mathtt{x} \neq \mathtt{i} \wedge \mathtt{F(x)} \geq 0)] \\ \wedge\;\; \mathtt{q} \Leftrightarrow \mathtt{x} \geq 0 \end{array}\right) \tag{27}$$

The only values of p and q where this formula cannot be satisfied is when p is false and q is true.

As shown in [Lahiri et al. 2003], we can generate the set of solutions to (23) and (26) by first transforming the formulas into equivalent quantified Boolean formulas, and then performing 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.

10. USING A SAT SOLVER TO PERFORM REACHABILITY ANALYSIS

Observe that (26) has a general form $\chi'(\mathcal{P}) = \chi(\mathcal{P}) \vee \exists \mathcal{A}\, \theta(\mathcal{A},\mathcal{P})$, where θ is a quantifier-free CLU formula, $\mathcal{A}$ contains Boolean, integer, function, and predicate symbols, and $\mathcal{P}$ contains only Boolean symbols. Several methods (including those in [Bryant et al. 2002b; Strichman et al. 2002; Bryant et al. 2002a]) have been developed to transform a quantifier-free CLU formula $\theta(\mathcal{A},\mathcal{P})$ into a Boolean formula $\tilde{\theta}(\tilde{\mathcal{A}},\mathcal{P})$, where $\tilde{\mathcal{A}}$ is now a set of Boolean variables, in a way that preserves satisfiability.

By taking care [Lahiri et al. 2003], this transformation can be performed in a way that preserves the set of satisfying solutions for the symbols in $\mathcal{P}$. That is:

$$\{\sigma_\mathcal{P} | \exists \sigma_\mathcal{A} : \langle\theta\rangle_{\sigma_\mathcal{A}\cdot\sigma_\mathcal{P}} = \mathbf{true}\} \;=\; \{\sigma_\mathcal{P} | \exists \sigma_{\tilde{\mathcal{A}}} : \left\langle\tilde{\theta}\right\rangle_{\sigma_{\tilde{\mathcal{A}}}\cdot\sigma_\mathcal{P}} = \mathbf{true}\} \tag{28}$$

Based on such a transformation, we can generate a Boolean formula for χ' by repeatedly calling a Boolean SAT solver, yielding one solution with each call. In this presentation, we consider an interpretation $\sigma_\mathcal{P}$ to represent a Boolean formula consisting of a conjunction of literals: p when $\sigma_\mathcal{P}(\mathtt{p}) = \mathbf{true}$ and $\neg\mathtt{p}$ when $\sigma_\mathcal{P}(\mathtt{p}) = \mathbf{false}$. Starting with $\chi' = \chi$, and

$\tilde{\theta}' = \tilde{\theta} \wedge \neg\chi$, we perform iterations:

$$\begin{aligned} \sigma_{\mathcal{A}}, \sigma_{\mathcal{P}} &\leftarrow SATSolve(\tilde{\theta}') \\ \chi' &\leftarrow \chi' \vee \sigma_{\mathcal{P}} \\ \tilde{\theta}' &\leftarrow \tilde{\theta}' \wedge \neg\sigma_{\mathcal{P}} \end{aligned}$$

until $\tilde{\theta}'$ is unsatisfiable.

To illustrate this process, let us solve (27) to perform the first iteration of reachability analysis on our example system. We can translate the right-hand term into Boolean form by introducing Boolean variables $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$, $\mathtt{d}$ and $\mathtt{e}$ encoding the predicates $\mathtt{F}(\mathtt{x}) \geq 0$, $\mathtt{x} \geq 0$, $\mathtt{F}(\mathtt{i}+1) \geq 0$, $\mathtt{i}+1 \geq 0$, and $\mathtt{x} = \mathtt{i}$, respectively.

The portion of (27) within square brackets then becomes

$$\mathtt{a} \Leftrightarrow \mathtt{b} \wedge \mathtt{c} \Leftrightarrow \mathtt{d} \wedge (\mathtt{p} \Leftrightarrow [(\mathtt{e} \wedge \mathtt{c}) \vee (\neg\mathtt{e} \wedge \mathtt{a})]) \wedge (\mathtt{q} \Leftrightarrow \mathtt{b}).$$

To this, let us add the consistency constraint: $\mathtt{e} \wedge \mathtt{b} \Rightarrow \mathtt{d}$ (encoding the property that $\mathtt{x} = \mathtt{i} \wedge \mathtt{x} \geq 0 \Rightarrow \mathtt{i}+1 \geq 0$). Although the translation schemes will add a lot more constraints (e.g., those involving uninterpreted function symbol), the above constraint is sufficient to preserve the property described in (28). For simplicity, we will not describe the other constraints that would be added by the algorithms in [Lahiri et al. 2003]. Finally, all the symbols apart from $\mathtt{p}$ and $\mathtt{q}$ are existentially quantified out.

It is easy to verify that the equation above with the consistency constraint is unsatisfiable only for the assignment when $\mathtt{p}$ is false and $\mathtt{q}$ is true.

11. 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 $\mathtt{f}(\mathtt{x}) > 0$ 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 $\Sigma_{\mathcal{P}}^{\mathcal{Q}}$ 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:

$$\alpha(s) \doteq \left\{ \langle\phi\rangle_{s\cdot\sigma_{\mathcal{X}}} \, | \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} \right\} \cap \Sigma_{\mathcal{P}}^{\mathcal{Q}} \tag{29}$$

and then using this definition in the extension of α to sets (Equation 3), the formulation of the reachability analysis (Equations 10 and 12), and the approximate reachability analysis (Equations 21 and 22).

The symbolic formulation of the approximate reachability analysis then becomes:

$$\begin{aligned}
\rho^0_\Pi(\mathcal{P}) &= \exists\mathcal{X}\exists\mathcal{J}\left(\bigwedge_{\mathtt{p}\in\mathcal{P}-\mathcal{Q}} \mathtt{p} \Leftrightarrow \phi_{\mathtt{p}}\left[q^0/\mathcal{V}\right] \;\wedge\; \bigwedge_{\mathtt{p}\in\mathcal{Q}} \phi_{\mathtt{p}}\left[q^0/\mathcal{V}\right]\right)\\
\rho^{i+1}_\Pi(\mathcal{P}) &= \rho^i_\Pi(\mathcal{P}) \vee \\
&\quad \exists\mathcal{V}\exists\mathcal{X}\exists\mathcal{I}\left(\bigwedge_{\pi\in\Pi}\left(\rho^i_\Pi\left[\phi/\mathcal{P}\right]\right)\left[\pi/\mathcal{X}\right] \;\wedge\; \bigwedge_{\mathtt{p}\in\mathcal{P}-\mathcal{Q}} \mathtt{p} \Leftrightarrow \phi_{\mathtt{p}}\left[\delta/\mathcal{V}\right] \;\wedge\; \bigwedge_{\mathtt{p}\in\mathcal{Q}} \phi_{\mathtt{p}}\left[\delta/\mathcal{V}\right]\right).
\end{aligned}$$

12. APPLICATIONS

We have integrated the method described in this paper into UCLID [Bryant et al. 2002b], a tool for modeling and verifying infinite-state systems. 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:

(1) A microprocessor out-of-order execution unit with an unbounded retirement buffer. Prior verification of this unit required manually generating 13 invariants [Lahiri et al. 2002]. The verification did not require any auxiliary invariants from the user and the proof script (which consists of the 24 simple predicates) is more compact than other verification efforts of similar models based on compositional model checking [McMillan 1998] or theorem proving methods [Arons and Pnueli 1999; Hosabettu et al. 1999].

(2) A directory-based cache protocol with unbounded channels, devised by Steven German of IBM [German], as discussed below.

(3) Versions of Lamport's bakery algorithm [Lamport 1974] that allow arbitrary number of processes to be active at each step or allow non-atomic reads and writes.

(4) Selection sort algorithm for sorting an arbitrary large array. We prove the property that upon termination, the algorithm produces an ordered array.

(5) A model of the Ad-hoc On-demand Distance Vector (AODV) routing protocol [C.Perkins et al. 2002]. This model was obtained from an earlier work [Das and Dill 2002], where the protocol was verified using quantified predicates.

(6) A crucial invariant (similar to the one proved in [Arons et al. 2001]) for proving the mutual exclusion for the Peterson's [Peterson 1981] mutual exclusion algorithm.

12.1 Directory-based Cache Coherence Protocol

For the directory-based German's cache-coherence protocol, an unbounded number of clients (`cache`), communicate with a central *home* process to gain *exclusive* or *shared* access to a memory line. The state of each `cache` can be {*invalid*, *shared*, *exclusive*}. The home maintains explicit representations of two lists of clients: those sharing the cache line (`sharer_list`) and those for which the home has sent an invalidation request but has not received an acknowledgment (`invalidate_list`) — this prevents sending duplicate invalidation messages.

The client places requests {*req_shared*, *req_exclusive*} on a channel `ch_1` and the home grants {*grant_shared*, *grant_exclusive*} on channel `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. $\forall \texttt{i} : \texttt{sharer\_list}(\texttt{i}) = \textbf{false}$. The home maintains variables for the current client ($\texttt{current\_client}$) and the current request ($\texttt{current\_command}$). It also maintains a bit $\texttt{exclusive\_granted}$ to indicate that some client has exclusive access. The cache lines acknowledge invalidation requests with a *invalidate_ack* along another channel $\texttt{ch\_3}$. At each step an input $\texttt{cid}$ is generated to denote the process that is chosen at that step. Details of the protocol operation with single-entry channels can be found in many previous works including [Pnueli et al. 2001]. We will refer to this version as *german-cache*.

Since the modeling language of UCLID does not permit explicit quantifiers in the system, we model the check for the absence of any sharers $\forall \texttt{i} : \texttt{sharer\_list}(\texttt{i}) = \textbf{false}$ alternately. We maintain a Boolean state variable $\texttt{empty\_hsl}$, which assumes an arbitrary value at each step of operation. We then add an axiom to the system: $\texttt{empty\_hsl} \Leftrightarrow \forall \texttt{i} : \texttt{sharer\_list}(\texttt{i}) = \textbf{false}$ [1]. The quantified test $\forall \texttt{i} : \texttt{sharer\_list}(\texttt{i}) = \textbf{false}$ in the model is replaced by $\texttt{empty\_hsl}$.

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. We refer to this version of the protocol as *german-cache-fifo*.

Proving Cache Coherence We first consider the version *german-cache* which has been widely used in many previous works [Pnueli et al. 2001; Emerson and Kahlon 2003; Baukus et al. 2002] among others and then consider the extended system *german-cache-fifo*. In both cases, the cache coherence property to prove is $\forall i, j : \texttt{cache}(i) = \textit{exclusive} \wedge i \neq j \Rightarrow \texttt{cache}(j) = \textit{invalid}$. All the experiments are performed on an 2.1GHz Pentium machine running Linux with 1GB of RAM.

12.1.1 *Invariant Generation for german-cache.* For this version, we derived two inductive invariants, one which involves a single process index i and other which involves two process indices i and j.

For single index invariant, we needed to add an auxiliary variable $\texttt{last\_granted}$ which tracks the last variable which has been granted exclusive access [Pnueli et al. 2001]. The inductive invariant which implies the cache coherence property was constructed using the following set of predicates:

$$\begin{aligned}\mathcal{P} = \{\ & \texttt{empty\_hsl}, \texttt{exclusive\_granted}, \texttt{current\_command} = \textit{req\_exclusive},\\ & \texttt{current\_command} = \textit{req\_shared}, i = \texttt{last\_granted}, \texttt{invalidate\_list}(i),\\ & \texttt{sharer\_list}(i), \texttt{cache}(i) = \textit{exclusive}, \texttt{cache}(i) = \textit{invalid}, \texttt{ch\_2}(i) = \textit{grant\_shared},\\ & \texttt{ch\_2}(i) = \textit{grant\_exclusive}, \texttt{ch\_2}(i) = \textit{invalidate}, \texttt{ch\_3}(i) = \textit{invalidate\_ack}\ \}.\end{aligned}$$

These predicates naturally appear in the system description. First, the predicates $\texttt{empty\_hsl}$ and $\texttt{exclusive\_granted}$ are Boolean state variables. Next, for each enumerated state

[1] Our current implementation only handles one direction of the axiom, $\forall \texttt{i} : \texttt{empty\_hsl} \Rightarrow \texttt{sharer\_list}(\texttt{i}) = \textbf{false}$, which is suffi cient to ensure the safety property.
[2] The extension was suggested by Steven German himself

variable $\mathtt{x}$, with range $\{e_1, \ldots, e_m\}$, we add the predicates $\mathtt{x} = e_1$, …, $\mathtt{x} = e_{m-1}$, leaving the redundant predicate $\mathtt{x} = e_m$. This explains $\mathtt{current\_command} = \textit{req\_shared}$ and $\mathtt{current\_command} = \textit{req\_exclusive}$. Next, we consider the values of the function and predicate state variables at a particular index i. In this example, such state variables are the `sharer_list`, `invalidate_list`, `cache`, `ch_1`, `ch_2` and `ch_3`. We did not need to add any predicate for the `ch_1` since the content of this channel does not affect the correctness condition. Finally, the predicate $i = \mathtt{last\_granted}$ was added for the auxiliary state variable `last_granted`.

With this set of 13 indexed predicates, the abstract reachability computation converged after 9 iterations in 14 seconds. Most of the time (about 8 seconds) was spent in eliminating quantifiers from the formula in (23) and (26) using the SAT-based quantifier elimination method.

For the dual index invariant, addition of the second index variable j makes the process computationally more expensive. However, the verification does not require any auxiliary variable to prove the correctness. The set of predicates used is:

$\mathcal{P} = \{\ \mathtt{cache}(i) = \textit{exclusive}, \mathtt{cache}(j) = \textit{invalid}, i = j, \mathtt{ch2}(i) = \textit{grant\_exclusive}, \mathtt{ch2}(i) = \textit{grant\_shared}, \mathtt{ch2}(i) = \textit{invalidate}, \mathtt{ch3}(i) = \textit{empty}, \mathtt{ch2}(j) = \textit{grant\_exclusive}, \mathtt{ch2}(j) = \textit{grant\_shared}, \mathtt{ch2}(j) = \textit{invalidate}, \mathtt{ch3}(j) = \textit{empty}, \mathtt{invalidate\_list}(i), \mathtt{current\_command} = \textit{req\_exclusive}, \mathtt{current\_command} = \textit{req\_shared}, \mathtt{exclusive\_granted}, \mathtt{sharer\_list}(i)\ \}.$

The inductive invariant which implies the cache-coherency was constructed using these 16 predicates in 41 seconds using 12 steps of abstract reachability. The portion of time spent on eliminating quantifiers was around 15 seconds.

12.1.2 *Invariant Generation for german-cache-fifo.* For this version, each of the channels, namely `ch1`, `ch2` and `ch3` are modeled as unbounded FIFO buffers. Each channel has a head (e.g. `ch1_hd`), which is the position of the earliest element in the queue and a tail pointer (e.g. `ch1_tl`), which is the position of the first *free* entry for the queue, where the next element is inserted. These pointers are modeled as function state variables, which maps process `i` to the value of the head or tail pointer of a channel for that process. For instance, `ch2_hd(i)` denotes the position of the head pointer for the process `i`. The channel itself is modeled as a two-dimensional array, where `ch2(i, j)` denotes the content of the channel at index `j` for the process `i`. We aim to derive an invariant over a single process index i and an index j for an arbitrary element of the channels. Hence we add the auxiliary variable `last_granted`. The set of predicates required for this model is:

$\mathcal{P} = \{\ \mathtt{cache}(i) = \textit{exclusive}, \mathtt{cache}(i) = \textit{invalid}, \mathtt{current\_command} = \textit{req\_shared}, \mathtt{current\_command} = \textit{req\_exclusive}, \mathtt{exclusive\_granted}, i = \mathtt{last\_granted}, \mathtt{invalidate\_list}(i), \mathtt{sharer\_list}(i), j = \mathtt{ch2\_hd}(i), j = \mathtt{ch3\_hd}(i), j \leq \mathtt{ch2\_hd}(i), j < \mathtt{ch2\_tl}(i), j \leq \mathtt{ch3\_hd}(i), j < \mathtt{ch3\_tl}(i), j = \mathtt{ch2\_tl}(i){-}1, \mathtt{ch1\_hd}(i) < \mathtt{ch1\_tl}(i), \mathtt{ch1\_hd}(i) = \mathtt{ch1\_tl}(i), \mathtt{ch2\_hd}(i) < \mathtt{ch2\_tl}(i), \mathtt{ch2\_hd}(i) = \mathtt{ch2\_tl}(i), \mathtt{ch2}(i,j) = \textit{grant\_exclusive}, \mathtt{ch2}(i,j) = \textit{grant\_shared}, \mathtt{ch2}(i,j) = \textit{invalidate}, \mathtt{ch3\_hd}(i) < \mathtt{ch3\_tl}(i), \mathtt{ch3\_hd}(i) = \mathtt{ch3\_tl}(i), \mathtt{ch3\_tl}(i) = \mathtt{ch3\_hd}(i)+1, \mathtt{ch3}(i,j) = \textit{invalidate\_ack}\ \}.$

Apart from the predicates required for *german-cache*, we require predicates involving entries in the various channels for a particular cache entry i. Predicates like $\texttt{ch1\_hd}(i) < \texttt{ch1\_tl}(i)$ and $\texttt{ch1\_hd}(i) = \texttt{ch1\_tl}(i)$ are used to determine if the particular channel is non-empty. To reason about *active* entries in a FIFO, i.e., those lying between the head (inclusive) and the tail, we need predicates like $j \leq \texttt{ch2\_hd}(i)$ and $j < \texttt{ch2\_tl}(i)$. The content of the channel at a location j is given by the predicates like $\texttt{ch2}(i, j) =$ *grant_exclusive* and $\texttt{ch3}(i, j) =$ *invalidate_ack*. Finally, a couple of predicates like $\texttt{ch3\_tl}(i) = \texttt{ch3\_hd}(i) + 1$ and $j = \texttt{ch2\_tl}(i) - 1$ are added by looking at failures to prove the cache coherence property.

Our tool constructs an inductive invariant with these 26 predicates which implies the cache coherence property. The abstract reachability took 17 iterations to converge in 1435 seconds. The quantifier elimination process took 1227 seconds.

A. SYNTAX AND SEMANTICS OF CLU

A.1 Syntax

Expressions in CLU (Figure 1) describe a means of computing four different types of values. *Boolean* expressions (*bool-expr*) yield **true** or **false**. We also refer to Boolean expressions as *formulas*. *Integer* expressions (*int-expr*), yield integer values. *Predicate* expressions (*predicate-expr*), denote functions from integers to Boolean values. *Function* expressions (*function-expr*), on the other hand, denote functions from integers to integers.

The simplest truth expressions are the values **true** and **false**. Boolean expressions can also be formed by comparing two term expressions for equality (referred to as an *equation*) or for ordering (referred to as an *inequality*), by applying a predicate expression to a list of term expressions, and by combining Boolean expressions using Boolean connectives.

Integer expressions can be integer variables, used only as the formal arguments of lambda expressions. They can also be formed by applying a function expression (including addition by constants) to a set of integer expressions, or by applying the *ITE* (for "if-then-else") operator. The *ITE* operator chooses between two values based on a Boolean control value, i.e., *ITE*$(\mathbf{true}, x_1, x_2)$ yields x_1, while *ITE*$(\mathbf{false}, x_1, x_2)$ yields x_2.

Function expressions can be either function symbols, representing uninterpreted functions, or lambda expressions, defining the value of the function as an integer expression containing references to a set of argument variables. Function symbols of arity 0 are also called *int-symbol*, symbolic constants of type integers. Since these symbols are instantiated without any arguments, we will omit the parentheses, writing a instead of $a()$.

Similarly, predicate expressions can be either predicate symbols, representing uninterpreted predicates, or lambda expressions, defining the value of the predicate as a Boolean expression containing references to a set of argument variables. Predicate symbols of arity 0 are also called *bool-symbol*, symbolic constants of type Booleans. They denote arbitrary Boolean values. We will also omit the parentheses following the instantiation of such a predicate.

Notice that we restrict the parameters to a lambda expression to be integers, and not function or predicate expressions. There is no way in our logic to express any form of iteration or recursion.

A.2 Semantics

For symbol set $\mathcal{A}$, 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 $\mathcal{Z}$ denote the set of integers. Interpretation $\sigma_{\mathcal{A}}$ assigns to each function symbol (in $\mathcal{A}$) of arity k, a function from $\mathcal{Z}^k$ to $\mathcal{Z}$, and to each predicate symbol (in $\mathcal{A}$) of arity k a function from $\mathcal{Z}^k$ to $\{\mathbf{true}, \mathbf{false}\}$. Let $\Sigma_{\mathcal{A}}$ denote the set of all interpretations $\sigma_{\mathcal{A}}$ over the symbol set $\mathcal{A}$.

For symbol set $\mathcal{A}$, let $E(\mathcal{A})$ denote the set of all CLU expressions over $\mathcal{A}$. For any expression $\phi \in E(\mathcal{A})$ and interpretation $\sigma_{\mathcal{A}} \in \Sigma_{\mathcal{A}}$, let the *valuation of ϕ with respect to $\sigma_{\mathcal{A}}$*, denoted $\langle\phi\rangle_{\sigma_{\mathcal{A}}}$ be the (Boolean, integer, function, or predicate) value obtained by evaluating ϕ when each symbol $\mathtt{x} \in \mathcal{A}$ is replaced by its interpretation $\sigma_{\mathcal{A}}(\mathtt{x})$. Figure 5 describes the evaluation inductively on the structure of any expression.

Form of Expression ϕ	Valuation $\langle\phi\rangle_{\sigma_{\mathcal{A}}}$
true	**true**
false	**false**
$\neg F$	$\neg \langle F\rangle_{\sigma_{\mathcal{A}}}$
$F_1 \wedge F_2$	$\langle F_1\rangle_{\sigma_{\mathcal{A}}} \wedge \langle F_2\rangle_{\sigma_{\mathcal{A}}}$
$F_1 \vee F_2$	$\langle F_1\rangle_{\sigma_{\mathcal{A}}} \vee \langle F_2\rangle_{\sigma_{\mathcal{A}}}$
$T_1 = T_2$	$\langle T_1\rangle_{\sigma_{\mathcal{A}}} = \langle T_2\rangle_{\sigma_{\mathcal{A}}}$
$T_1 < T_2$	$\langle T_1\rangle_{\sigma_{\mathcal{A}}} < \langle T_2\rangle_{\sigma_{\mathcal{A}}}$
predicate-expr$(T_1, \ldots, T_k)$	$\langle$*predicate-expr*$\rangle_{\sigma_{\mathcal{A}}} (\langle T_1\rangle_{\sigma_{\mathcal{A}}}, \ldots, \langle T_k\rangle_{\sigma_{\mathcal{A}}})$
ITE(F, T_1, T_2)	if $\langle F\rangle_{\sigma_{\mathcal{A}}} = \mathbf{true}$ then $\langle T_1\rangle_{\sigma_{\mathcal{A}}}$ else $\langle T_2\rangle_{\sigma_{\mathcal{A}}}$
function-expr$(T_1, \ldots, T_k)$	$\langle$*function-expr*$\rangle_{\sigma_{\mathcal{A}}} (\langle T_1\rangle_{\sigma_{\mathcal{A}}}, \ldots, \langle T_k\rangle_{\sigma_{\mathcal{A}}})$
$T_1 +$ *int-constant*	$\langle T_1\rangle_{\sigma_{\mathcal{A}}} +$ *int-constant*
f	$\langle f\rangle_{\sigma_{\mathcal{A}}}$
p	$\langle p\rangle_{\sigma_{\mathcal{A}}}$
$(\lambda\, v_1, \ldots, v_n \,.\, \tau)$	$\left\langle \tau_{[x_1/v_1, \ldots, x_n/v_n]}\right\rangle_{\sigma_{\mathcal{A}}}$, when applied to $(x_1, \ldots, x_n)$

Fig. 5. **Semantics of CLU**

Acknowledgments

We wish to thank Ching-Tsun Chou for his detailed comments on an early draft of this paper. We are also grateful to the reviewers for their insightful comments on the paper.

REFERENCES

APT, K. R. AND KOZEN, D. 1986. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters 22,* 5, 307–309.

ARONS, T. AND PNUELI, A. 1999. Verifying Tomasulo's algorithm by Refinement. In *Proc. VLSI Design Conference (VLSI '99).*

ARONS, T., PNUELI, A., RUAH, S., ZHU, Y., AND ZUCK, L. 2001. Parameterized verification with automatically computed inductive assertions. In *Computer-Aided Verification (CAV '01),* G. Berry, H. Comon, and A. Finkel, Eds. LNCS 2102. 221–234.

BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. K. 2001. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01).* Snowbird, Utah. *SIGPLAN Notices,* 36(5), May 2001.

BAUKUS, K., LAKHNECH, Y., AND STAHL, K. 2002. Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2002*, A. Cortesi, Ed. LNCS 2294. 317–330.

BÖRGER, E., GRÄDEL, E., AND GUREVICH, Y. 1997. *The Classical Decision Problem*. Springer-Verlag.

BOUAJJANI, A., JONSSON, B., NILSSON, M., AND TOUILI, T. 2000. Regular model checking. In *Computer-Aided Verification (CAV 2000)*, A. Emerson and P. Sistla, Eds. LNCS 1855. Springer-Verlag, 403–418.

BRYANT, R. E., LAHIRI, S. K., AND SESHIA, S. A. 2002a. Deciding CLU Logic formulas via Boolean and Pseudo-Boolean encodings. In *Proc. Intl. Workshop on Constraints in Formal Verification (CFV'02)*.

BRYANT, R. E., LAHIRI, S. K., AND SESHIA, S. A. 2002b. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Computer-Aided Verification (CAV'02)*, E. Brinksma and K. G. Larsen, Eds. LNCS 2404. 78–92.

BUCHI, J. R. 1960. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math. 6*, 66–92.

BURCH, J. R. AND DILL, D. L. 1994. Automated verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, D. Dill, Ed. LNCS 818. 68–80.

CHAKI, S., CLARKE, E. M., GROCE, A., JHA, S., AND VEITH, H. 2003. Modular Verification of Software Components in C. In *International Conference on Software Engineering (ICSE '03)*. IEEE Computer Society 2003, 385–395.

CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. 1992. Model checking and abstraction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. 342–354.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation : A Unified Lattice Model for the Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages (POPL '77)*. ACM Press.

C.PERKINS, ROYER, E., AND DAS, S. 2002. Ad hoc on demand distance vector (aodv) routing. In *IETF Draft, Available at http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-10.txt*. North-Holland, Amsterdam.

DAS, S. AND DILL, D. 2001. Successive approximation of abstract transition relations. In *IEEE Symposium of Logic in Computer Science(LICS '01)*. IEEE Computer Society.

DAS, S., DILL, D., AND PARK, S. 1999. Experience with predicate abstraction. In *Computer-Aided Verification (CAV '99)*. LNCS 1633. Springer-Verlag.

DAS, S. AND DILL, D. L. 2002. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, M. D. Aagaard and J. W. O'Leary, Eds. LNCS 2517. 19–32.

DIJKSTRA, E. W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18*, 453–457.

EMERSON, E. A. AND KAHLON, V. 2000. Reducing model checking of the many to the few. In *International Conference on Automated Deduction*, D. A. McAllester, Ed. 1831. 236–254.

EMERSON, E. A. AND KAHLON, V. 2003. Exact and efficient verification of parameterized cache coherence protocols. In *Correct Hardware Design and Verification Methods (CHARME '03)*, D. Geist and E. Tronci, Eds. LNCS 2860. 247–262.

EMERSON, E. A. AND NAMJOSHI, K. S. 1995. Reasoning about rings. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. 85–94.

FLANAGAN, C. AND QADEER, S. 2002. Predicate abstraction for software verification. In *Symposium on Principles of programming languages (POPL '02)*, J. Launchbury and J. C. Mitchell, Eds. ACM Press, 191–202.

GERMAN, S. Personal communication.

GERMAN, S. M. AND SISTLA, A. P. 1992. Reasoning about systems with many processes. *Journal of the ACM 39*, 3, 675–735.

GRAF, S. AND SAÏDI, H. 1997. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, O. Grumberg, Ed. LNCS 1254. Springer-Verlag.

HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy Abstraction. In *Symposium on Principles of programming languages (POPL '02)*, J. Launchbury and J. C. Mitchell, Eds. ACM Press, 58–70.

HOSABETTU, R., GOPALAKRISHNAN, G., AND SRIVAS, M. 1999. Proof of correctness of a processor with reorder buffer using the completion function approach. In *Computer-Aided Verification (CAV 1999)*. LNCS.

IP, C. N. AND DILL, D. L. 1996. Verifying systems with replicated components in Murφ. In *Computer-Aided Verification (CAV '96)*, R. Alur and T. A. Henzinger, Eds. LNCS 1102. Springer-Verlag, 147–158.

KESTEN, Y., MALER, O., MARCUS, M., PNUELI, A., AND SHAHAR, E. 1997. Symbolic model checking with rich assertional languages. In *Computer-Aided Verification (CAV '97)*, O. Grumberg, Ed. LNCS 1254. Springer-Verlag, 424–435.

LAHIRI, S. K. 2004. *Unbounded System Verification Using Decision Procedures and Predicate Abstraction*. Carnegie Mellon University.

LAHIRI, S. K. AND BRYANT, R. E. 2004a. Constructing Quantifi ed Invariants via Predicate Abstraction. In *Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, G. Levi and B. Steffen, Eds. LNCS 2937. 267–281.

LAHIRI, S. K. AND BRYANT, R. E. 2004b. Indexed Predicate Discovery for Unbounded System Verifi cation. In *Computer Aided Verification (CAV '04) (to appear)*.

LAHIRI, S. K., BRYANT, R. E., AND COOK, B. 2003. A symbolic approach to predicate abstraction. In *Computer-Aided Verification (CAV 2003)*, W. A. Hunt, Jr. and F. Somenzi, Eds. LNCS 2725. Springer-Verlag, 141–153.

LAHIRI, S. K., SESHIA, S. A., AND BRYANT, R. E. 2002. Modeling and verifi cation of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, J. W. O. M. Aagaard, Ed. LNCS 2517. Springer-Verlag, 142–159.

LAMPORT, L. 1974. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM 17*, 453–455.

MCMILLAN, K. 1998. Verifi cation of an implementation of Tomasulo's algorithm by compositional model checking. In *Computer-Aided Verification (CAV 1998)*, A. J. Hu and M. Y. Vardi, Eds. LNCS 1427. 110–121.

MCMILLAN, K., QADEER, S., AND SAXE, J. 2000. Induction in compositional model checking. In *Computer-Aided Verification (CAV 2000)*, A. Emerson and P. Sistla, Eds. LNCS 1855. Springer-Verlag.

PETERSON, G. L. 1981. Myths about the mutual exclusion problem. *Information Processing Letters 12,* 3, 115–116.

PNUELI, A., RUAH, S., AND ZUCK, L. 2001. Automatic deductive verifi cation with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems(TACAS'01)*, T. Margaria and W. Yi, Eds. Vol. LNCS 2031. 82–97.

SAÏDI, H. AND SHANKAR, N. 1999. Abstract and model check while you prove. In *Computer-Aided Verification*, N. Halbwachs and D. Peled, Eds. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, 443–454.

STRICHMAN, O., SESHIA, S. A., AND BRYANT, R. E. 2002. Deciding Separation Formulas with SAT. In *Proc. Computer-Aided Verification (CAV'02)*, E. Brinksma and K. G. Larsen, Eds. LNCS 2404. 209–222.

THOMAS, W. 1990. Automata on infi nite objects. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*.

Received July 2004; revised March 2006; accepted March 2006