Indexed Predicate Discovery for Unbounded System Verification[*]

Shuvendu K. Lahiri and Randal E. Bryant

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

Abstract. Predicate abstraction has been proved effective for verifying several infinite-state systems. In predicate abstraction, an abstract system is automatically constructed given a set of predicates. Predicate abstraction coupled with automatic predicate discovery provides for a completely automatic verification scheme. For systems with unbounded integer state variables (e.g. software), counterexample guided predicate discovery has been successful in identifying the necessary predicates.

For verifying systems with function state variables, which include systems with unbounded memories (microprocessors), arrays in programs, and parameterized systems, an extension to predicate abstraction has been suggested which uses predicates with *free* (index) variables. Unfortunately, counterexample guided predicate discovery is not applicable to this method. In this paper, we propose a simple heuristic for discovering indexed predicates. We illustrate the effectiveness of the approach for verifying safety properties of two systems: (i) a version of the Bakery mutual exclusion protocol, and (ii) a directory-based cache coherence protocol with unbounded FIFO channels per client.

1 Introduction

Predicate abstraction [15] has emerged as a successful technique for analyzing infinite-state systems. The infinite-state systems consist of both hardware and software systems, where the state variables can assume arbitrarily large sets of values. Predicate abstraction, which is a special instance of the more general theory of *abstract interpretation* [9], automatically constructs a finite-state abstract system from a potentially infinite state concrete system, given a set of *predicates* (where a predicate describes some property of the concrete system). The abstract system can be used to synthesize inductive invariants or perform model checking to verify properties of the concrete system.

For synthesizing inductive invariants, predicate abstraction can be viewed as a systematic way to compose a set of predicates $\mathcal{P}$ using the Boolean connectives ($\wedge$, $\vee$, $\neg$) to construct the strongest inductive invariant that can be expressed with these predicates. This process can be made efficient by using symbolic and Boolean techniques based on incremental SAT and BDD-based algorithms [21, 7]. Thus, predicate abstraction can construct complex invariants given a set of predicates.

[*] This research was supported in part by the Semiconductor Research Corporation, Contract RID 1029.001.

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 135–147, 2004.

For systems which do not require quantified invariants, it suffices to use simple atomic predicates (predicates do not contain $\vee, \wedge$ or $\neg$). The simplicity of the predicates make them amenable to automatic predicate discovery schemes [2, 6, 17]. All these methods use the framework of counterexample-guided abstraction refinement [19, 8] to add new predicates which eliminate spurious counterexample traces over the abstract system. Automatic predicate discovery coupled with the automatic abstraction provided by predicate abstraction makes the verification process fully automatic. This has been the cornerstone of many successful verification systems based on predicate abstraction [2, 6, 17].

To verify systems containing unbounded resources, such as buffers and memories of arbitrary size and systems with arbitrary number of identical, concurrent processes, the system model must support state variables that are mutable functions or predicates [25, 10, 5]. For example, a memory can be represented as a function mapping an address to the data stored at an address, while a buffer can be represented as a function mapping an integer index to the value stored at the specified buffer position. The state elements of a set of identical processes can be modeled as functions mapping an integer process identifier to the state element for the specified process.

To verify systems with function state variables, we require quantified predicates to describe global properties of state variables, such as "At most one process is in its critical section," as expressed by the formula $\forall i, j : \texttt{crit}(i) \wedge \texttt{crit}(j) \Rightarrow i = j$. Conventional predicate abstraction restricts the scope of a quantifier to within an individual predicate. System invariants often involve complex formulas with widely scoped quantifiers. The scoping restriction (the fact that quantifiers do not distribute over Boolean connectives) 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 [10], but it has not been successful for many of the systems that we consider.

Our earlier work [21, 20] and the work by Flanagan and Qadeer (in the context of unbounded arrays in software) [13] overcome this problem by allowing the predicates to include free variables from a set of *index* variables $\mathcal{X}$. We call these predicates as *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 $\texttt{crit(i)}$, $\texttt{crit(j)}$, and $\texttt{i} = \texttt{j}$, where $\texttt{i}$ and $\texttt{j}$ are the index variables.

One of the consequences of adding indexed predicates is that the state space defined over the predicates does not have a transition relation [20]. This is a consequence of the fact that the abstraction function α maps each concrete state to a *set* of abstract states, instead of a single abstract state as happens in predicate abstraction [15]. The lack of an abstract transition relation prevents us from generating an abstract trace and thus rules out the counterexample-guided refinement framework.

In this work, we look at a technique to generate the set of predicates iteratively. Our idea is based on generating predicates by computing the *weakest liberal precondition* [11], similar to Namjoshi and Kurshan [27] and Lakhnech et al. [23]. Our method differs from [27] in that we simply use the technique as a heuristic for discovering useful indexed predicates. We rely on predicate abstraction to construct invariants using these predicates. The method in [27] proposed computing the abstract transition relation on-the-fly using the weakest precondition. The methods in [23, 6] can be seen as generating new (quantifier-free) predicates using the counterexample-guided refinement framework with some acceleration techniques in [23].

The techniques have been integrated in UCLID [5] verifier, which supports a variety of different modeling and verification techniques for infinite-state systems. We describe the use of the predicate inference scheme for verifying the safety properties of two protocols: (i) A version of the N-process Bakery algorithm by Lamport [24], where the reads and writes are atomic and (ii) A extension of the cache-coherence protocol devised by Steven German of IBM [14], where each client communicates to the central process using unbounded channels. The protocols were previously verified by manually constructing predicates in [20]. In contrast, in this work the protocols are verified almost automatically with minimal intervention from the user.

Related Work. The method of invisible invariants [28, 1] 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 the inductive invariant. The method has been successfully used to verify German's protocol with single entry channels and a version of the Bakery algorithm, where all the tickets have an upper bound and a loop is abstracted with an atomic test. However, the class of system handled by the method is restricted; it can't be applied to the extension of the cache-coherence protocol we consider in this paper or an out-of-order processor that is considered in our method [22].

McMillan uses compositional model checking [25] with various built in abstractions and symmetry reduction to reduce an infinite-state system to a finite state version, which can be model checked using Boolean methods. Since the abstraction mechanisms are built into the system, they can often be very 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. The proof of safety of the Bakery protocol required non-trivial lemmas in the compositional model checking framework [26].

Regular model checking [18, 4] 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, several examples can't be modeled in this framework; the out-of-order processor or the Peterson's mutual exclusion (which can be modeled in our framework) are few such examples. Even though the Bakery algorithm can be verified in this framework, it requires user ingenuity to encode the protocol in a regular language.

Emerson and Kahlon [12] have verified the version of German's cache coherence protocol with single entry channels by reducing it to a snoopy protocol, which can in turn be verified automatically by considering finite instances of the parameterized problem. 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.

Predicate abstraction with locally quantified predicates [10, 3] require complex quantified predicates to construct the inductive assertions, as mentioned in the introduction. These predicates are often as complex as invariants themselves. The method in [3] verified (both safety and liveness) a version of the cache coherence protocol with single entry channels, with complex manually provided predicates. In comparison, our method constructs an inductive invariant automatically to prove cache coherence. Till date, automatic predicate discovery methods for quantified predicates [10] have not been demonstrated on the examples we consider in this paper.

2 Preliminaries

The concrete system is defined in terms of a decidable subset of first-order logic. Our implementation is based on the CLU logic [5], supporting expressions containing uninterpreted functions and predicates, equality and ordering tests, and addition by integer constants. The logic supports Booleans, integers, functions mapping integers to integers, and predicates mapping integers to Booleans.

2.1 Notation

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

For a set of symbols $\mathcal{A}$, let $\sigma_{\mathcal{A}}$ denote an *interpretation* of these symbols, assigning to each symbol $\mathrm{x} \in \mathcal{A}$ a value $\sigma_{\mathcal{A}}(\mathrm{x})$ 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 $\mathcal{A}$. Let $\sigma_{\mathcal{A}} \cdot \sigma_{\mathcal{B}}$ be the result of combining interpretations $\sigma_{\mathcal{A}}$ and $\sigma_{\mathcal{B}}$ over disjoint set of symbols $\mathcal{A}$ and $\mathcal{B}$.

For symbol set $\mathcal{A}$, let $E(\mathcal{A})$ denote the set of all expressions in the logic over $\mathcal{A}$. For any expression $e \in E(\mathcal{A})$ and interpretation $\sigma_{\mathcal{A}} \in \Sigma_{\mathcal{A}}$, let $\langle e \rangle_{\sigma_{\mathcal{A}}}$ be the value obtained by evaluating e when each symbol $\mathrm{x} \in \mathcal{A}$ is replaced by its interpretation $\sigma_{\mathcal{A}}(\mathrm{x})$. For a set of expressions $\mathbf{v}$, such that $v_{\mathrm{x}} \in E(\mathcal{B})$, we extend $\langle \mathbf{v} \rangle_{\sigma_{\mathcal{B}}}$ to denote the (named) set of values obtained by applying $\sigma_{\mathcal{B}}$ to each element v_{x} of the set.

A *substitution* $\boldsymbol{\pi}$ for a set of symbols $\mathcal{A}$ is a named set of expressions, such that for each $\mathrm{x} \in \mathcal{A}$, there is an expression π_{x} in $\boldsymbol{\pi}$. For an expression e we let $e\,[\boldsymbol{\pi}/\mathcal{A}]$ denote the expression resulting when we (simultaneously) replace each occurrence of every symbol $\mathrm{x} \in \mathcal{A}$ with the expression π_{x}.

2.2 System Description and Concrete System

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.

The overall system operation is described by an *initial-state* expression set $\mathbf{q}^0$ and a *next-state* expression set $\boldsymbol{\delta}$. That is, for each state element $\mathtt{x} \in \mathcal{V}$, the expressions $q^0_{\mathtt{x}} \in E(\mathcal{J})$ and $\delta_{\mathtt{x}} \in E(\mathcal{V} \cup \mathcal{I})$ denote the initial state expression and the next state expression for $\mathtt{x}$.

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

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

3 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 $\boldsymbol{\phi}$, where for each $\mathrm{p} \in \mathcal{P}$, predicate ϕ_{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. We can denote a set of abstract states by a Boolean formula $\psi \in E(\mathcal{P})$. This expression defines a set of states $\langle \psi \rangle \doteq \{\sigma_{\mathcal{P}} | \langle \psi \rangle_{\sigma_{\mathcal{P}}} = \mathbf{true}\}$.

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: $\alpha(s) \doteq \left\{\langle \boldsymbol{\phi} \rangle_{s \cdot \sigma_{\mathcal{X}}} | \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}}\right\}$. We then extend the abstraction function to apply to sets of concrete states in the usual way: $\alpha(S_C) \doteq \bigcup_{s \in S_C} \alpha(s)$.

We define the concretization function γ for a set of abstract states $S_A \subseteq \Sigma_{\mathcal{P}}$: $\gamma(S_A) \doteq \left\{s | \forall \sigma_{\mathcal{X}} \in \Sigma_{\mathcal{X}} : \langle \boldsymbol{\phi} \rangle_{s \cdot \sigma_{\mathcal{X}}} \in S_A\right\}$, to require universal quantification over the index symbols.

The universal quantifier in this definition has the consequence that the concretization function does not distribute over set union. In particular, we cannot view the concretization function as operating on individual abstract states, but rather as generating each concrete state from multiple abstract states.

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 $Q^0_A \doteq \alpha(Q^0_C)$ and a next-state function operating on sets: $N_A(S_A) \doteq \alpha(N_C(\gamma(S_A)))$.

It is important to note that there is no transition relation associated with this next-state function, since γ cannot be viewed as operating on individual abstract states. In previous work [20], we provide examples where a pair of abstract states s_a and s'_a each has an empty set of abstract successors, but the set of successors of $\{s_a,s'_a\}$ is non-empty.

We perform reachability analysis on the abstract system using N_A as the next-state function: $R^0_A = Q^0_A$ and $R^{i+1}_A = R^i_A \cup N_A(R^i_A)$. Since the abstract system is finite, there must be some n such that $R^n_A = R^{n+1}_A$. The set of all reachable abstract states R_A is then R^n_A. Let $\rho_A \in E(\mathcal{P})$ be the expression representing R_A. The corresponding set of concrete states is given by $\gamma(R_A)$, and can be represented by the expression $\forall \mathcal{X} : \rho_A\, [\phi/\mathcal{P}]$. In previous work [20], we showed that the concretization of R_A (or equivalently $\forall \mathcal{X} : \rho_A\, [\phi/\mathcal{P}]$) is the strongest universally quantified inductive invariant that can be constructed from the set of predicates.

Since there is no complete way for handling quantifiers in first order logic with equality and uninterpreted functions [16], we resort to sound quantifier instantiation techniques to compute an overapproximation of the abstract state space. The quantifier instantiation method uses heuristics to choose a finite set of terms from the possible infinite range of values and replaces the universal quantifier by a finite conjunction over the terms. More details of the technique can be found in [20] and details of the quantifier instantiation heuristic can be found in [22]. The method has sufficed for all the examples we have seen so far. The predicate abstraction is carried out efficiently by using Boolean techniques [21].

The inductive invariant is then used to prove the property of interest by the decision procedure inside UCLID. If the assertion holds, then the property is proved. We have used this method to verify safety properties of cache-coherence protocols, mutual exclusion algorithms, out-of-order microprocessors and sequential software programs with unbounded arrays [20]. However, most predicates were derived manually by looking at failures or adding predicates that appear in the transition function.

4 Indexed Predicate Discovery

This section presents a syntactic method for generating indexed predicates, based on *weakest liberal precondition* [11] transformer. A similar idea has been used in [23], but they do not consider indexed predicates. As a result, they can use methods based on analyzing abstract counterexample traces to refine the set of predicates. In our case, we only use it as a syntactic heuristic for generating new predicates. An inexpensive syntactic heuristic is also more suited to our approach since computing the inductive invariant is an expensive process [20] for large number of predicates (> 30), even with the recent advances in symbolic methods for predicate abstraction [21]. More importantly, the simple heuristic has sufficed for automating the verification of non-trivial problems.

The *weakest precondition* of a set of states S_C is the largest set of states T_C, such that for any state $t_c \in T_C$, the successor states lie in S_C. If Ψ_C is an expression representing the set of states S_C, then the expression which represents the WP(Ψ_C) is $\forall \mathcal{I} : \Psi_C \, [\delta/\mathcal{V}]$. To obtain this expression in terms of the state variables, one would have to perform quantifier elimination to eliminate the input symbols $\mathcal{I}$. In general, eliminating quantifiers over integer symbols in the presence of uninterpreted functions in $\Psi_C \, [\delta/\mathcal{V}]$ is undecidable [16].

Let us see the intuition (without any rigorous formal basis, since its only a heuristic) for using WP for predicate discovery. Consider a predicate ϕ without any index variables. A predicate represents a property of the concrete system, since it is a Boolean formula over the state variables. Thus, if WP(ϕ) (WP($\neg\phi$)) is true at a state, then ϕ (respectively $\neg\phi$) will be true in the next state. Therefore the predicates which appear in WP(ϕ) are important when tracking the truth of the predicate ϕ accurately. Since computing WP(ϕ) as a predicate over $\mathcal{V}$ is undecidable in general, we choose predicates from $\Psi_C \, [\delta/\mathcal{V}]$ without explicitly eliminating the quantifiers over $\mathcal{I}$. We later provide a strategy to deal with predicates which involve input symbols. This intuition can be naturally extended to indexed predicates. In this case, our aim is to generate predicates which involve the index symbols. For a predicate ϕ over $\mathcal{V} \cup \mathcal{X}$, the predicates in $\phi \, [\delta/\mathcal{V}]$ involve $\mathcal{X}$ and is a good source for mining additional indexed predicates.

We start with the set of predicates in the property to be proved. If the final property to be proved is $\forall \mathcal{X} : \Psi(\mathcal{V}, \mathcal{X})$, we extract the indexed predicates that appear in $\Psi(\mathcal{V}, \mathcal{X})$. At each predicate discovery step, we generate new predicates from the weakest precondition of the existing predicates. An inductive invariant over the combined set of predicates is constructed by predicate abstraction. If the invariant implies the property, we are done. Otherwise, we iterate the process. This process can be repeated until no more predicates are discovered or we exhaust resources.

There are several enhancements over the simple idea that were required to generate meaningful predicates. The problems encountered and our solutions are as follows:

If-then-Else Constructs. To generate atomic predicates, the *if-then-else* (*ITE*) constructs are eliminated from the WP expression by two rewrite rules. First, we distribute a function application over an *ITE* term to both the branches i.e. $f(\mathit{ITE}(G,\, T_1,\, E_1)) \longrightarrow \mathit{ITE}(G,\, f(T_1),\, f(E_1))$. Second, we distribute the comparisons over *ITE* to both the branches, i.e. $\mathit{ITE}(G_1,\, T_1,\, E_1) \bowtie T_2 \longrightarrow (G_1 \wedge T_1 \bowtie T_2) \vee (\neg G_1 \wedge E_1 \bowtie T_2)$, where $\bowtie \in \{<, =\}$. Since the only arithmetic supported in our modeling formalism is addition by constants [5], the final predicates (after all the rewrites are applied) are of the form $T_1 \bowtie T_2 + c$, where T_i is an integer symbol or a function application.

Input Symbols. Since predicates relate state variables, input symbols in $\mathcal{I}$ should not be considered as part of predicates (the input symbols are universally quantified out while computing the WP). Boolean valued input variables can be safely ignored. However, integer input variables are different.

The principal source of integer inputs is the arbitrary index that is generated for choosing a process or an instruction to execute non-deterministically. Ignoring predicates containing inputs can often result in loosing useful predicates. Consider a fragment of a code for modeling a simple protocol in UCLID, where pc is a function state variable, representing the state of each process and cid is an arbitrary process identifier (input) generated to at each step.

```
next[pc] :=                (* next state of 'pc' state variable *)
Lambda (i).  case          (* next state for the ith index      *)
  i != cid     : pc(i) ;   (* if i != cid, remain unchanged     *)
  pc(cid) = A  : B ;       (* else if pc(i) = A, update to B    *)
  pc(cid) = B  : C ;       (* else if pc(i) = B, update to C    *)
  default      : pc(i);    (* else remain unchanged             *)
esac;
```

Let us assume that we are generating the set of predicates for $\mathtt{pc}(x) = C$, where $x \in \mathcal{X}$. The set of predicates generated from this code fragment is $\{x = \mathtt{cid}, \mathtt{pc}(\mathtt{cid}) = A, \ldots, \}$. Ignoring predicates containing cid returns an empty set of new predicates.

To circumvent this problem, we check if an input variable inp appears in any predicate $\mathtt{inp} = x$, where $x \in \mathcal{X}$. In such a case, we say x is a *match* for inp. We repeat this for each input symbol in $\mathcal{I}$. We can have multiple input variables to index into a multidimensional array. Let $\mathcal{X}^{\mathcal{I}}$ be the named set of index variables, such that $\mathcal{X}^{\mathcal{I}}_{\mathtt{a}}$ represents the match for input a.

We first ignore predicates that contain input variables without any matches. For any other predicate ϕ, we generate the predicate $\phi\left[\mathcal{X}^{\mathcal{I}}/\mathcal{I}\right]$. For the above example, the predicates generated are: $\{x = x, \mathtt{pc}(x) = A, \ldots, \}$, which generates important predicates. Trivial predicates such as $x = x$ are ignored.

Arithmetic Predicates. In the presence of even simple arithmetic in the model, e.g. $\delta(\mathtt{v}_1) \doteq \mathtt{v}_1 + 1$, predicate abstraction generates too many predicates, often generating towers such as $\mathtt{v} \bowtie \mathtt{v}_2 + 1, \ldots, \mathtt{v} \bowtie \mathtt{v}_2 + k$. Most often these predicates are not essential for the property to be proved.

To prevent such predicates, we only generate predicates of the form $T_1 = T_2$ or $T_1 < T_2$, where T_i is a integer state variable or a function application. The first time we see a predicate $T_1 \bowtie T_2 + c$ (where $c \neq 0$), we generate the predicates $T_1 = T_2$ and $T_1 < T_2$. We ignore the predicate $T_1 \bowtie T_2 + c$ for the next step of predicate generation. This step is automatically performed. If predicates involving $+c$ are required, the user is expected to provide them. This step can be seen as a lightweight acceleration approach.

Nested Function Applications. While generating predicates for a given index variable, say i, we add new index variables to be placeholders for nested function applications. For instance, if the predicate discovered is $\mathtt{F}(\mathtt{G}(i)) = T_1$, where F and G are state variables, we introduce a new index variable j, add the predicate $j = \mathtt{G}(i)$ and replace the predicate (with the nested function application) with a new predicate $\mathtt{F}(j) = T_1$. All nested occurrences of $\mathtt{G}(i)$ are replaced with j for subsequent predicates. At present, the user determines if new index variables have to be added, as the addition of too many index variables often overwhelm the predicate abstraction engine. We are currently automating this step.

Generalizing from a Fixed Index. Suppose we have a state variable v and we have a predicate that involves $\mathtt{P}(\mathtt{v})$, (where P is a function or predicate state variable). If at some point in the future, we see the predicate $x = \mathtt{v}$, where $x \in \mathcal{X}$, then we rewrite all the predicates which contains v as an argument to a function or predicate state variable by substituting x for v. For example, the predicate $\mathtt{P}(\mathtt{v}) = 5$ is rewritten as $\mathtt{P}(x) = 5$. This rule has the effect of generalizing a predicate. It also removes the predicate $\mathtt{P}(\mathtt{v}) = 5$, since it can be constructed as a combination of $x = \mathtt{v}$ and $\mathtt{P}(x) = 5$. We have found this rule crucial for generating important predicates and also keep the size of the predicate set small.

5 Case Studies

5.1 Bakery Protocol

In this section, we describe the verification of mutual exclusion property for a version of N-process Bakery algorithm proposed by Lamport [24]. For this verification, we only model atomic reads and writes. A version with non-atomic reads and writes has been verified with manually supplied predicates [20].

Each process has a Boolean variable choosing, a ticket number and a loop index j. Each process can be in one of 6 program locations $\{L_0, \ldots, L_5\}$. A program counter variable pc holds the current program location a process is in. All the statements for a particular program location are executed atomically. The test $(\mathtt{a}_1, \mathtt{a}_2) \le (\mathtt{b}_1, \mathtt{b}_2)$ stands for $\mathtt{a}_1 < \mathtt{b}_1 \lor \mathtt{a}_1 = \mathtt{b}_1 \land \mathtt{a}_2 \le \mathtt{b}_2$.

The variables are initialized as:

```
choosing := Lambda i. false ;  j := Lambda i. 0 ;
number := Lambda i. 0 ; pc := Lambda i. L0 ;
```

and the overall protocol for the process at index i is described in the following pseudo code:

```
L0 : choosing[i] := true; goto L1 ;
L1 : number[i] := max(number[0],..,number[N-1]) + 1;
     choosing[i] := false; j[i] := 0; goto L2 ;
L2 : if (!choosing(j[i])) then  {goto L3;}
     else {goto L2;}
L3 : if (number(j[i]) = 0 ||
        (number[i],i) <= (number[j[i]],j)) then {goto L4;}
     else {goto L3;}
L4 : if (j[i] < N) then {j[i] := j[i]+1; goto L2;}
     else {goto L5;}
L5 : Critical-Section; number[i] := 0 ; goto L0;
```

We model the computation of the maximum value of $\mathtt{number}(0), \ldots, \mathtt{number}(\mathtt{N}-1)$, in program location L_1 as an atomic operation. This is modeled by introducing a state variable max which takes on arbitrary values at each step, and an axiom that says: $\forall i : \mathtt{max} \ge \mathtt{number}(i)$. At each step, at most one process (at index cid) is scheduled to execute.

Proving Mutual Exclusion. The property we want to prove is that of mutual exclusion, namely $\forall i, j : \mathtt{pc}(i) = L_5 \land \mathtt{pc}(j) = L_5 \Rightarrow (i = j)$. We start out with a single index variable i. We start with the predicates in the property, namely $\{\mathtt{pc}(i) = L_5\}$.

The predicates discovered after the first round are: $P^1 \doteq \{\ \mathtt{j}(i) = \mathtt{N}, \mathtt{N} < \mathtt{j}(i), \mathtt{pc}(i) = L_4, i = \mathtt{j}(i), i < \mathtt{j}(i), l = \mathtt{j}(i), \mathtt{number}(i) = \mathtt{number}(l), \mathtt{number}(i) < \mathtt{number}(l), \mathtt{number}(l) = 0, \mathtt{pc}(i) = L_3, \mathtt{choosing}(l), \mathtt{pc}(i) = L_2, \mathtt{pc}(i) = L_1, \mathtt{pc}(i) = L_0, i = \mathtt{N}, \mathtt{N} < i, i < 0\ \}$. We introduced a second index symbol l when the predicate $\mathtt{number}(i) = \mathtt{number}(\mathtt{j}(i))$ with nested function application was encountered. We replaced this predicate with $\{l = \mathtt{j}(i), \mathtt{number}(i) = \mathtt{number}(l)\}$. Similarly the predicates $\mathtt{number}(i) = \mathtt{number}(\mathtt{j}(i))$ and $\mathtt{choosing}(\mathtt{j}(i))$ were replaced with $\mathtt{number}(i) = \mathtt{number}(l)$ and $\mathtt{choosing}(l)$ respectively. We ignored one redundant predicate $\mathtt{j}(i) > \mathtt{N}$, since both $\mathtt{j}(i) = \mathtt{N}$ and $\mathtt{j}(i) < \mathtt{N}$ are present. We also encountered the predicate $i = \mathtt{cid}$, and thus we will replace all future occurrences of cid with i in subsequent predicates. It took 0.81 seconds to discover that the initial set of predicates was not sufficient and to discover these predicates.

It took 55.8 seconds to find a counterexample with this set of predicates. The next set of predicates generated are: $P^2 \doteq \{\ i = 0, l = 0, l < \mathtt{j}(i), \mathtt{number}(i) = 0, \mathtt{pc}(l) = L_5, \mathtt{pc}(l) = L_1, l = i, l = \mathtt{N}, \mathtt{N} < l, l < 0, \mathtt{number}(i) < 0, 0 < \mathtt{number}(l), \mathtt{pc}(l) = L_0, 0 < N, N < 0\ \}$. We filtered out quite a number of predicates using our rules. First the predicate $\mathtt{j}(i) + 1 = \mathtt{N}$ gave rise to $\{\mathtt{j}(i) = \mathtt{N}, \mathtt{j}(i) < \mathtt{N}\}$, both of which are already present. Similarly $\mathtt{j}(i) + 1 < \mathtt{N}$ did not give rise to any new predicates. Likewise, the predicates generated from $i = \mathtt{j}(i) + 1$ are also present in P^1. We ignored predicates which involve max since it is merely a input. The predicate $l < \mathtt{j}(i)$ was generated from $l = \mathtt{j}(i) + 1$ ($l = \mathtt{j}(i)$ is already present). Finally $l = i$ was generated from $l = \mathtt{cid}$ by substituting i for cid in the predicate. All the filtering happened automatically inside the tool, we describe them to point out the effectiveness of the various rules.

With this set of 33 predicates, we were able to derive an inductive invariant in 471 seconds and 18 iterations of the abstract reachability. The inductive invariant implies the mutual exclusion property.

5.2 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).

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 ch_2. The home grants exclusive access to a client only when there are no clients sharing a line, i.e. $\forall \mathtt{i} : \mathtt{sharer\_list}(\mathtt{i}) = \textbf{false}$. The home maintains variables for the current client (current_client) and the current request (current_command). It also maintains a bit exclusive_granted to indicate that some client has exclusive access. The cache lines acknowledge invalidation requests with a *invalidate_ack* along another channel ch_3. At each step an input 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 [28]. 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 $\forall \texttt{i} : \texttt{sharer\_list(i)} = \textbf{false}$ alternately. We maintain a Boolean state variable empty_hsl, which assumes an arbitrary value at each step of operation. We then add an axiom to the system: $\texttt{empty\_hsl} \Leftrightarrow \forall \texttt{i} : \texttt{sharer\_list(i)} = \textbf{false}$ [1].

In our version of the protocol, each cache communicates to the home process through three directed unbounded FIFO channels, namely the channels ch_1, ch_2, 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 [28, 12, 3] among others and then consider the extended system *german-cache-fifo*. In both cases, the cache coherence property to prove is $\forall i, j : \texttt{cache}(i) = \textit{exclusive} \wedge i \neq j \Rightarrow \texttt{cache}(j) = \textit{invalid}$. Details of the verification of both *german-cache* and *german-cache-fifo* can be found at the authors homepage[3]. Here, we briefly sketch the highlights of the experiments. All the experiments are run on an 2.1GHz Pentium machine running Linux with 1GB of RAM. Let P^k denote the predicates discovered after k iterations of WP computation.

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 brevity, we will only describe the dual indexed invariant.

For the dual indexed invariant, we start off with the predicates in the property, namely $P^0 \doteq \{ \texttt{pc}(i) = \textit{exclusive}, \texttt{pc}(j) = \textit{invalid}, i = j \}$. The predicates discovered in subsequent iterations are listed below:

$P^1 \doteq \{ \texttt{ch2}(i) = \textit{grant\_exclusive}, \texttt{ch2}(i) = \textit{grant\_shared}, \texttt{ch3}(i) = \textit{empty}, \texttt{ch2}(i) = \textit{invalidate}, \texttt{ch2}(j) = \textit{grant\_exclusive}, \texttt{ch2}(j) = \textit{grant\_shared}, \texttt{ch3}(j) = \textit{empty}, \texttt{ch2}(j) = \textit{invalidate} \}$.

$P^2 \doteq \{ \texttt{ch2}(i) = \textit{empty}, \texttt{current\_command} = \textit{req\_exclusive}, i = \texttt{current\_client}, \texttt{invalidate\_list}(i), j = \texttt{current\_client}, \texttt{current\_command} = \textit{req\_shared}, \texttt{exclusive\_granted}, \texttt{ch3}(i) = \textit{invalidate\_ack}, \texttt{current\_command} = \textit{empty}, \texttt{ch2}(j) = \textit{empty}, \texttt{invalidate\_list}(j), \texttt{ch3}(j) = \textit{invalidate\_ack} \}$.

$P^3 \doteq \{ \texttt{sharer\_list}(i), \texttt{ch1}(i) = \textit{empty}, \texttt{ch1}(i) = \textit{req\_exclusive}, \texttt{ch1}(i) = \textit{req\_shared}, \texttt{sharer\_list}(j) \}$.

The inductive invariant which implies the cache-coherency was constructed using these 28 predicates in 1266 seconds using 15 steps of abstract reachability. The entire process took less than 2000 seconds of CPU time.

Invariant Generation for ***German-Cache-Fifo*****.** The addition of an unbounded number of FIFOs increases the complexity of the model. We constructed an inductive invariant (which implies cache coherence) with a process index i and an index j for the channels.

[1] Our current implementation only handles one direction of the axiom, $\forall \texttt{i} : \texttt{empty\_hsl} \Rightarrow \texttt{sharer\_list(i)} = \textbf{false}$, which is sufficient to ensure the safety property.

[2] The extension was suggested by Steven German himself.

[3] http://www.ece.cmu.edu/~shuvendu/papers/cav04a-submit.ps

We needed to add two predicates for the FIFOs manually, which contain constant offsets (e.g. $x < y - 1$). The time taken to construct the inductive invariant with 29 predicates was 3 hours.

6 Conclusions

In this work, we have demonstrated that predicate abstraction with indexed predicates coupled with simple heuristics for indexed predicate discovery can be very effective for automated or systematic verification of unbounded systems. The verification is carried out without knowledge about the operation of any of the protocols. The technique has also been applied for the systematic generation of invariants for an out-of-order microprocessor engine and a network protocol.

There is a lot of scope for improving the performance of invariant generation procedure by finding a minimal set of predicates. For instance, the inductive invariant for *german-cache-fifo* could be computed using 26 manually specified predicates in 581 seconds. Hence, there is a scope of almost 20X improvement that can be possibly obtained by a suitable selection of predicates. We are also experimenting with proof-based predicate discovery, where the proof that no concrete counterexample exists within a finite number of steps is used to discover new predicates.

References

1. T. Arons, A. Pnueli, S. Ruah, Y. Zhu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification (CAV '01)*, LNCS 2102, pages 221–234, 2001.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, Snowbird, Utah, June, 2001. *SIGPLAN Notices,* 36(5), May 2001.
3. K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized Verification of a Cache Coherence Protocol: Safety and Liveness. In A. Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation, VMCAI 2002*, LNCS 2294, pages 317–330, January 2002.
4. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In A. Emerson and P. Sistla, editors, *Computer-Aided Verification (CAV 2000)*, LNCS 1855, pages 403–418, July 2000.
5. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In E. Brinksma and K. G. Larsen, editors, *Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.
6. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *International Conference on Software Engineering (ICSE)*, pages 385–395, May 2003.
7. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate Abstraction of ANSI-C Programs using SAT. Technical Report CMU-CS-03-186, Carnegie Mellon University, 2003.
8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification (CAV '00)*, LNCS 1855, pages 154–169, 2000.
9. P. Cousot and R. Cousot. 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)*, 1977.

10. S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 19–32, 2002.
11. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
12. E. A. Emerson and V. Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods (CHARME '03)*, LNCS 2860, pages 247–262, 2003.
13. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In J. Launchbury and J. C. Mitchell, editors, *Symposium on Principles of programming languages (POPL '02)*, pages 191–202, 2002.
14. Steven German. Personal communication.
15. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer-Aided Verification (CAV '97)*, LNCS 1254, June 1997.
16. Y. Gurevich. The decision problem for standard classes. *The Journal of Symbolic Logic*, 41(2):460–464, June 1976.
17. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In J. Launchbury and J. C. Mitchell, editors, *Symposium on Principles of programming languages (POPL '02)*, pages 58–70, 2002.
18. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Computer-Aided Verification (CAV '97)*, LNCS 1254, pages 424–435, June 1997.
19. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1995.
20. S. K. Lahiri and R. E. Bryant. Constructing Quantified Invariants via Predicate Abstraction. In G. Levi and B. Steffen, editors, *Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, LNCS 2937, pages 267–281, 2004.
21. S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer-Aided Verification (CAV 2003)*, LNCS 2725, pages 141–153, 2003.
22. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In J. W. O'Leary M. Aagaard, editor, *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 142–159, Nov 2002.
23. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume LNCS 2031, pages 98–112, April 2001.
24. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17:453–455, August 1974.
25. K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification (CAV 1998)*, LNCS 1427, pages 110–121, June 1998.
26. K. McMillan, S. Qadeer, and J. Saxe. Induction in compositional model checking. In A. Emerson and P. Sistla, editors, *Computer-Aided Verification (CAV 2000)*, LNCS 1855, July 2000.
27. K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In A. Emerson and P. Sistla, editors, *Computer Aided Verification*, LNCS 1855, pages 435–449, 2000.
28. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems(TACAS'01)*, volume LNCS 2031, pages 82–97, 2001.