Experience with Predicate Abstraction *

Satyaki Das[1], David L. Dill[1], and Seungjoon Park[2]

[1] Computer Systems Laboratory, Stanford University, Stanford, CA 94305
[2] RIACS, NASA Ames Research Center, Moffett Field, CA 94035

Abstract. This reports some experiences with a recently-implemented prototype system for verification using predicate abstraction, based on the method of Graf and Saïdi [9]. Systems are described using a language of iterated guarded commands, called *Murϕ*$^{--}$ (since it is a simplified version of our Murϕ protocol description language). The system makes use of two libraries: SVC [1] (an efficient decision procedure for quantifier-free first-order logic) and the CMU BDD library. The use of these libraries increases the scope of problems that can be handled by predicate abstraction through increased efficiency, especially in SVC, which is typically called thousands of times. The verification system also provides limited support for quantifiers in formulas. The system has been applied successfully to two nontrivial examples: the Flash multiprocessor cache coherence protocol, and a concurrent garbage collection algorithm. Verification of the garbage collector algorithm required proving properties simple of graphs, which was also done using predicate abstraction.

1 Introduction

Abstraction is emerging as the key to formal verification of large designs, especially designs that are not finite-state. *Predicate abstraction,* first described by Graf and Saïdi [9], provides a means for combining theorem proving and model checking techniques by automatically mapping an unbounded system (called the *concrete system*) to a finite state system (called the *abstract system*). The states of the abstract system correspond to truth assignments to a set of predicates. The user must supply the predicates and properties to be proven. The system automatically model checks the properties on the abstract system defined by the predicates. The abstraction is *conservative*, meaning that if a property is shown to hold on the abstract system, there is a concrete version of the property that holds on the concrete system; however, if the property fails to hold on the abstract system, it may or may not hold on the concrete system.

We have recently implemented a prototype system for efficient verification of invariants by predicate abstraction, to discover how far predicate abstraction can take us towards the goal of formal verification of real systems. Results have

* This work was supported by DARPA/NASA contract DABT63-96-C-0097-P00002 and NASA contract NASI-98139. The content of this paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

been encouraging. Systems are described using a language of iterated guarded commands, which we call $Mur\phi^{--}$ (since it is a simplified version of our Murϕ protocol description language). The system makes use of two libraries: an efficient decision procedure for quantifier-free first-order logic, called SVC [1], and the CMU BDD library written by David Long. The use of these libraries increases the scope of problems that can be handled by predicate abstraction through increased efficiency, especially in SVC, which is typically called thousands of times. The prototype verifier is written in Common Lisp, and the libraries (which are written in C and C++) are called via the "foreign function" interface.

We have applied it successfully to two nontrivial examples: the Flash multiprocessor cache coherence protocol, and a concurrent garbage collection algorithm. In verification, discovering strategies for effective use of a tool is often as important as the design of the tool. We quickly found that we needed limited support for quantifiers, for expressing properties of unbounded numbers of processes and data. For the garbage collection algorithm, it was necessary to prove some properties of a recursive function. Interestingly, some recursive algorithms can be verified by translating them to $Mur\phi^{--}$ and using predicate abstraction.

The more detailed description below has programs written in a syntax other than $Mur\phi^{--}$, and logical formulas in a syntax other than SVC. The benefits of readability were deemed to outweigh the possibility of translation errors.

Related work

Our work is derived from the Graf/Saïdi abstraction scheme [9]. However, the original implementation represented the abstract state space as a set of *monomials* (a monomial is a product of Boolean variables and negated variables). Instead, we use BDDs, which usually represent Boolean functions more efficiently. However, Graf and Saidi also sacrificed some accuracy by representing the image of a monomial under a transition rule as a single monomial which must cover all of the states in the image of the transition rule. Our method has no such restriction. So, our verifier is more accurate, but may require more computation (which is performed more efficiently).

Our approach to handling parameterized systems uses quantified formulas, (similar to [17] and [13]), which differs from the method presented in [12]. They used linear systems of equations to deal with state transitions. The basic idea is that for each state there is an abstract variable which keeps track of the number of processes in that state. So if a process moves from q to q' then the value of X_q is decremented by one while $X_{q'}$ is incremented by one. We have handled reasoning about parameterized systems by introducing formulas quantified over the replicated processes as abstract state variables. This is similar to what was proposed in [8] and [7].

Another approach to generating abstract state graphs is to abstract the concrete rules [3]. This has the advantage of requiring fewer validity checks (as they are required when constructing the abstract transitions). However, abstracting the rules may also lose more information about the concrete system, and so might be unable to prove the invariant of interest.

2 Predicate abstraction

This section summarizes the theory of predicate abstraction and its implementation in the prototype verifier. The notation is somewhat different from Graf and Saïdi's, but everything is very similar until the details of the computation of the successors of a set of abstract states (the recursive decomposition).

The concrete and abstract descriptions

As with previous work in this area, the concrete system is modeled as a collection of iterated nondeterministic commands. There is a single global state variable X that represents the complete state of the system. Multiple state variables can be represented by making them fields of a variable of record type. The initial state of the concrete system is generated by an assignment $X := init(X)$ [1]

There is a set of transition rules. Each rule defines a transition function f which maps states to states (the input language has guarded commands, but the guards are not necessary since the transition functions can be defined to leave state variables unchanged when their guards are not satisfied).

An *execution* of the system is a sequence of states, $q_0, \ldots q_n, q_{n+1}, \ldots$, where $q_0 = init(q_{-1})$ for some arbitrary state q_{-1} (note that q_{-1} does not occur in the execution sequence) and $q_{n+1} = f(q_n)$ for some transition function f. A concrete state q is reachable if it appears in some execution sequence. We are interested in whether predicates on the state variables are *invariants*, meaning that they hold for every reachable state of a concrete system.

An abstract system is defined by a concrete system and a set of N predicates, $\phi_1, \phi_2, \ldots \phi_N$. Each state q_A of the abstract state space is a truth assignment to the indices 1 through N (so the set of states is finite). The predicates define an *abstraction function*, α, which maps concrete states to abstract states. In particular, $q_A = \alpha(q_C)$ whenever $\forall i : q_A(i) = \phi_i(q_C)$. An abstract state q_A is *reachable* if it is an abstraction of a reachable concrete state q_C.

The reachable state space can be used to check invariants. If the user knows what invariants he or she wants to prove, these invariants are supplied as some of the predicates ϕ_i (actually, the invariant may sometimes be decomposed into a conjunction of simpler properties). If $q_A(i)$ is true in all reachable abstract states, the invariant has been proven. In addition, a BDD describing the abstract reachable state space can be converted into an invariant for the concrete state space by *concretizing* it, as described below.

Approximating the abstract reachable state space

Sets of abstract or concrete states are represented using logical formulas. Abstract states are represented using BDDs, which can be regarded as propositional

[1] Initialization depends on the values of the state variables, which are unconstrained, so as to allow nondeterministic choice of start states. The initialization rule is also conveniently similar to the transition rules of the system.

formulas, by associating Boolean variables $B_1, \ldots, B_n$ with the truth values of the corresponding predicates. The concrete domain is not necessarily finite, so the concrete state space is represented using first-order formulas.

If s_C is a set of concrete states, $\alpha(s_C)$ will be taken to be $\{\alpha(q_C) \mid q_C \in s_C\}$. The *concretization function* γ is the inverse image of α: $\gamma(s_A) = \{q_C \mid \alpha(q_C) \in s_A\}$. Note that $\forall s_C : s_C \subseteq \gamma(\alpha(s_C))$. If ψ_A is a propositional formula (e.g., a BDD) over the variables B_i representing the set s_A, a first-order formula ψ_C representing $\gamma(s_A)$ can be computed by substituting each predicate ϕ_i for B_i in ψ_A.

An approximation of the reachable state space of the abstract system is computed by (the usual) breadth-first symbolic traversal. At any time, the algorithm has a BDD representing the current abstract reachable set. Initially, this formula represents an abstraction of the initial states of the concrete system. Then, the algorithm iteratively computes an over-approximation of the set of all successors of the current reachable set. At the end of the next iteration, the formula is the logical disjunction of the formula for the current reachable set and the formula for its successor set.

The key step in this procedure is how to find the formula for the set of successors. Given a BDD ψ_A which characterizes s_A, find a BDD ψ'_A characterizing the successors of s_A in the abstract system. It is sufficient to compute the successors contributed by each concrete transition function f, since the set of abstract successors is the union of the successors contributed by the individual functions. The formula for the initial abstract states is computed by finding the possible successors of the entire state space under the "transition function" *init* (in other words, finding the formula for the successors of *true* under *init*).

The abstract successors are computed by a method similar to that of Graf and Saïdi, but using recursive subdivision of the concrete state space. The first step computes $\psi_C = \gamma(\psi_A)$ by substitution (as described above). ψ_C represents the set of all states that could abstract to a state in s_A.

We assume that each transition function f can be written as a first-order term, which is also name f. Predicates $\phi'_i(x)$ that characterize the sets $\{q_C \mid \phi_i(f(q_C))\}$ can be pre-computed by substituting the term $f(X)$ for X in ϕ. Intuitively, $\phi'_i(x)$ means "x is a predecessor of a state that can satisfy ϕ_i."

We compute ψ'_A by recursive case splitting on each bit B_i in the abstract formula, in ascending order of i.

$$H(\psi, m) = \begin{cases} \begin{array}{c} B_m \wedge H(\psi \wedge \phi'_m, m+1) \\ \bigvee \neg B_m \wedge H(\psi \wedge \neg\phi'_m, m+1) \end{array} & \text{if } 0 < m \leq N \\ \textit{true} & \text{if } m = N+1 \wedge \psi \text{ is satisfiable} \\ \textit{false} & \text{if } m = N+1 \wedge \psi \text{ is unsatisfiable} \end{cases}$$

The formula ψ is a Boolean combination of predicates ϕ_i for $m \leq i \leq N+1$. If s is the set of concrete states represented by ψ, the function H, below, computes a logical formula representing the set of abstract states $\alpha(f(s))$. If $m \leq N$, it splits s into two parts, s' and s'', by conjoining ψ with ϕ_i and then $\neg\phi_i$; H is then called recursively to compute $\alpha(f(s'))$ and $\alpha(f(s''))$. When $m = N+1$, every ϕ_i has been assumed *true* or *false* in ψ, so ψ is equivalent to one of these values.

Several important optimizations are not shown. First, $H(\mathit{false}, m)$ is always *false*, so we check whether ψ is satisfiable at each step, using SVC. Second, $H(\psi, m)$ is saved in a table the first time it is computed; this table is checked to see if the needed value is available before computing H recursively. Finally, the propositional operations are performed using a BDD library, so common subexpressions are shared.

Dealing with indexed sets of transitions.

Murϕ^{--}, like Murϕ before it, allows the user to define a set of transition rules that vary over an index variable. There is a construct called a "ruleset," which declares a index variable that can be used in the code for transition rules contained in the ruleset. This feature is useful for describing collections of nearly identical processes.

Ruleset parameters are encoded as accesses to an infinite array, indexed by the natural numbers, whose entries are rule indices. The contents of the array are unconstrained, so it serves as a source of nondeterministic choices. The ith element of the array is looked up to determine the choice of the transition rule to execute in the ith step of a computation.

Stating properties of parameterized systems requires quantified formulas, but SVC can only decide quantifier-free formulas. The prototype verifier copes with quantifiers using some simple heuristics:

- In parameterized processes, the concrete variables associated with each process are frequently stored in an array, so quantified variables are instantiated with all array index expressions.
- Since SVC checks validity, variables that are universally quantified outside of the scope of an existential quantifier can be replaced by a fresh symbolic constant (which is distinct from all other names in the formula). Instantiation of quantifiers with these fresh variables is also useful.
- As a last resort, the system allows the user to supply hints about how to instantiate (and not instantiate) variables.

These measures are barely adequate; more sophisticated handling of quantifiers is required in the future.

3 FLASH cache coherence protocol example

One advantage of predicate abstraction is that it can be used to strengthen invariants, automatically. This is potentially valuable, since finding appropriate invariants is one of the most difficult aspects of verifying a design using a theorem prover.

This technique was evaluated on a protocol that was previously verified by several methods: the Stanford FLASH multiprocessor cache coherence protocol.

The model of the cache coherence protocol consists of a set of nodes, each of which contains a processor, caches, and a portion of global memory of the system. Each cache line-sized block in memory is associated with a directory header which keeps information about the line. The state of a cached copy is in either *invalid*, *shared* (readable), or *exclusive* (readable and writable). The distributed nodes communicate using asynchronous messages through a point-to-point network.

This protocol has been verified using an aggregation abstraction with help of a theorem prover. This proof required many lemmas that showed that various pairs of actions commute (produce the same state, regardless of execution order). However, the lemmas don't hold in arbitrary system states; instead, it is necessary to prove an invariant that characterizes the reachable states, then prove that the lemma holds given the invariant. Finding this invariant was the most difficult part of the proof. A more detailed description of the protocol and the proof can be found in [14].

To prove the invariants, it is necessary to strengthen them until they are inductive (strengthening them is equivalent to finding an induction hypothesis). In practice, strengthening an invariant is a trial-and-error process involving repeated failed proofs, from which new properties must be manually extracted. This usually requires many iterations, and each iteration is difficult.

Predicate abstraction makes invariant strengthening easier. The user supplies plausible properties that might be useful in strengthening the invariant, and the system automatically tries various Boolean combinations of these conditions until it is able to prove the property (or not). This saves the effort of trying Boolean combinations by hand. When the abstract reachability analysis generates a state where the candidate invariant does not hold, it is possible to report an abstract state, along with a concrete transition that enters the state. This information may suggest additional predicates that should be added.

To use predicate abstraction for invariant strengthening, the user starts with a description of the system and some (relatively simple) invariants that are sufficient conditions to prove the verification conditions of interest. For example, a desired property of FLASH was that there be at most one exclusive copy of a memory line in the system. To prove this, two predicates were supplied initially:

- There are no exclusive copies.
- There is a single exclusive copy

The invariants discovered using these properties are not strong enough, so two more properties were added about the PUTX message, which is a message from the directory to the cache that wants an exclusive copy.

- There are no PUTX replies in the network.
- There is a single PUTX reply in the network

The $Mur\phi^{--}$ description of the protocol used in this test was somewhat different from the PVS description used in the aggregation proof. The first simplification was modeling the memory as a separate node in the machine, when in fact memory is stored in processing nodes. This simplification was necessitated by the inefficient treatment of quantifiers in the current $Mur\phi^{--}$ prototype. The second simplification was the result of a limitation of $Mur\phi^{--}$: In the PVS description, the directory entry for a memory block maintained a count of sharers (read-only cached copies of the memory block). There was no easy way to count the number of actual sharers in $Mur\phi^{--}$, so this was changed to be the set of sharing nodes, instead of a count. [2] In spite of these compromises, we believe that the problem of invariant strengthening for the modified FLASH protocol is quite difficult, and the ability to solve it with $Mur\phi^{--}$ indicates that predicate abstraction is an effective approach to this problem.

One of the interesting challenges presented by the FLASH protocol is finding invariants for an unknown number of processes. As with the original description, the protocol description is parameterized for unknown number of processes. The caches are modeled as an unbounded array indexed by node indices. This tends to lead to predicates and properties to prove that are quantified over all process indices. For instance, the property that there should be no write-back request when there exists any exclusive copy of the memory line in the whole system can be specified with a universal quantifier as

$$\forall p : (\ cache[p].state = exclusive \Rightarrow net_{WB} = empty\ ).$$

As explained in Section 2, $Mur\phi^{--}$ is able to handle quantified predicates, albeit sub-optimally, by trying many instantiations without human interaction. This capability was critical for completing the proof with reasonable effort.

Overall, we estimate that finding the invariants with predicate abstraction was at least an order-of-magnitude easier than finding them by trial and error with PVS. It required no more than five days of user time and two hours of CPU time to strengthen the invariants.

4 Garbage collection example

The most ambitious example we have attempted is the on-the-fly garbage collection algorithm, which was first proposed by Dijkstra, et al. [4]. The algorithm is widely acknowledged to be difficult to get right, and difficult to prove. A more detailed discussion of the subtlety of this algorithm and subsequent variations can be found in a paper by Havelund and Shankar [11].

An extended version of this algorithm which can handle multiple concurrent mutators was used as the garbage collector of Concurrent Caml Light. The proof of the safety property required 58 invariants to be proved. Details of the modified algorithm and its proof are discussed in [6] and [5].

The original algorithm was simplified by Ben-Ari [2] to involve two colors instead of three. This also led to a simpler argument of correctness. Alternative

[2] This problem could possibly have been addressed by writing a recursive function to count the sharing nodes, then verifying some properties of it as in the proof of the garbage collection algorithm. We haven't tried this yet.

justifications of Ben-Ari's algorithm were also given by Van de Snepscheut [18] and Pixley [15]. However, these proofs were informal *pencil and paper* proofs.

Later, this modified algorithm was mechanically proved by Russinoff [16] using the Boyer-Moore theorem prover. A formulation of the same algorithm was also proved by Havelund and Shankar in PVS [10] and [11]. The proofs of both [10] and [11] were of approximately the same size. The proofs needed 19 invariant lemmas and 57 function lemmas and [11] took about two months. So far as we know, no one has mechanically proved the original algorithm of Dijkstra, et al.

In the garbage collection algorithm, the *collector* and the *mutator* (which models the behavior of the user program by nondeterministically changing pointers) run concurrently with both processes accessing a shared memory. The memory is abstractly modeled as a directed graph with each node having at most two outgoing edges. A subset of these nodes are called *roots*; they are special in the sense that they are always accessible (our proof of the algorithm assumes without loss of generality that there is only one root node). Any node that can be reached from one of the roots by following edges is also accessible. The mutator is allowed to choose an arbitrary node and redirect one of its edges to an arbitrarily chosen accessible node. Each memory node also has a *color* field which the collector uses to keep track of the accessible nodes. The collector adds nodes that are not accessible to the mutator, so-called *garbage* nodes, to a *free-list* for recycling.

The mutator, which is described in pseudo-code in Figure 1, first redirects an edge of an arbitrarily selected accessible node towards an arbitrary accessible node ($acc(j)$ says j is accessible). It then colors the second node gray if it was white, or otherwise does nothing. Part of the subtlety of the algorithm is that the collector can mark nodes between these two steps of the mutator.

The collector finds the nodes that are not reachable from the roots, so they can be added to the free list. It begins by coloring the root nodes gray ("coloring a node gray" is called *shading*, from now on). Then it iterates through all the nodes; whenever it finds a gray node, it shades its successors and colors the node black. After this the collector starts this iteration again. The collector algorithm is presented in Figure 1.

The basic property to prove is that the collector does not free an accessible node. An extra state variable called *error* was added to the collector, which is set to *true* if the collector ever frees an accessible node, reducing the desired property to an invariant that *error* is never true.

Most of the predicates were simply guards from the $Mur\phi^{--}$ description of the algorithm or derived directly from the invariant to be proved. Some required insight, however. Two predicates are needed because, when the collector is in the *marking phase*, the mutator can change the color of a node to gray, in which case there must already exist a gray node yet to be examined by the collector.

$$\forall x \in [i, M) : color[x] \neq gray$$
$$\exists y \in [0, M) : color[y] = gray$$

```
/* mutator */
while(true)
    choose n, k ∈ [0, M),
        s.t. acc(k) = true
    /* choose to change left or right */
    [   left[n] := k; q := k
    □  right[n] := k; q := k]
    if color[q] = white →
        color[q] := gray; fi
end /* while */
```

```
/* collector */
shade all roots;
error := false;
i := 0; k := M;
/* marking phase */
do (k > 0) →
   c := color[i];
   if c = gray  →
      k := M;
      shade left[i], right[i];
      color[i] := black;
   □c ≠ gray → k := k − 1
   fi;
   i := (i + 1) mod M
od
/* collecting phase */
j := 0;
do  (j < M) →
   c := color[j];
   if c = white →
      if acc(j) → error := true fi
      append j to free list
   □c ≠ white → color[j] := white
   fi;
   j := j + 1
od
```

Fig. 1. Mutator and Collector Algorithms

The correctness of the algorithm also depends on the invariant that a black node never has a white successor (except in the transitory case where the mutator is about to shade the white successor).

$$\forall x \in [0, M) : (\mathit{color}[x] = \mathit{black} \Rightarrow (\mathit{color}[\mathit{left}[x]] \neq \mathit{white} \vee q = \mathit{left}[x]))$$
$$\forall x \in [0, M) : (\mathit{color}[x] = \mathit{black} \Rightarrow (\mathit{color}[\mathit{right}[x]] \neq \mathit{white} \vee q = \mathit{right}[x]))$$

Verifying properties of graphs

A major difficulty with verifying the garbage collection algorithm using predicate abstraction is that its correctness depends on some simple properties of graphs that are not easy to prove by simple instantiation of quantifiers (induction is actually needed). These properties are given as axioms to the verifier when verifying the algorithm, and are proved by using predicate abstraction on "auxiliary" $Mur\phi^{--}$ programs that compute the graph properties.

For example, the following property about the function *acc* is necessary:

$$\begin{aligned}
&(color[0] = black) \\
&\wedge(\forall p \in [0, M) : color[p] = black \\
&\qquad\qquad\qquad \Rightarrow (color[left[p]] = black \wedge color[right[p]] = black)) \\
&\Rightarrow (\forall q \in [0, M) : acc(left, right)(q) \Rightarrow (color[q] = black))
\end{aligned} \tag{1}$$

(The function *acc* is actually a function of the graph structure of the nodes, so *left* and *right* are its arguments.)

Another axiom is says that redirecting an edge to point to an already accessible node never makes a previously inaccessible node accessible. In the following, $write(left, q, p)$ represents an array which is the same as *left* except that it has the value p at index q. There is a similar equation for redirecting the right side.

$$\begin{aligned}
\forall p, q, r \in [0, M) : (&acc(left, right)(p) \wedge acc(write(left, q, p), right)(r)) \\
&\Rightarrow acc(left, right)(r)
\end{aligned} \tag{2}$$

The most difficult property required some insight. It states that if the root node of the graph is gray in color and all other nodes are either gray or white then, for every accessible white node, there exists a path from a gray node to it, entirely through white nodes.

$$\begin{aligned}
&(color[0] = gray \wedge \forall x \in [0, M) : color[x] = white \wedge acc(left, right)(x)) \\
&\quad \Rightarrow \exists y \in [0, M) : color[y] = gray \wedge reachable_white(left, right)(y, x)
\end{aligned} \tag{3}$$

where *reachable_white* is a similarly recursive definition that says there is a path of all white nodes from *left* to *right*.

It is frequently possible to write an auxiliary $Mur\phi^{--}$program that computes a graph property, then verify some predicates on this algorithm. The verified properties are then used as axioms in the main verification effort. These auxiliary programs are not tricky to write, because they do not require concurrency. Although this method is currently *ad hoc*, it seems that the properties we encountered, and many others, could be written as simple recursive definitions and then translated by some provably correct algorithm to a $Mur\phi^{--}$program that computes the same property.

For example, starting with a simple recursive definition of accessibility,

$$acc(0) \wedge (\forall x \in [0, M) : acc(x) \Rightarrow (acc(left(x)) \wedge acc(right(x))),$$

it is simple to write a $Mur\phi^{--}$program that sets the entries of an array $acc[i]$ to true or false depending on whether node i is accessible.

To prove property 1, we assume that the array *color* is initialized so that

$$\begin{aligned}
&(color[0] = black) \\
&\wedge(\forall p \in [0, M) : color[p] = black \\
&\qquad\qquad\qquad \Rightarrow (color[left[p]] = black \wedge color[right[p]] = black))
\end{aligned}$$

and then check the abstract state space with the predicate $\forall x : acc[x] \Rightarrow color[x] = black$.

A similar approach was used to prove property 2. This property was slightly more complex, since the function needed to be computed twice: once on the original memory structure and once after the mutator has redirected an edge in the memory graph.

As might be expected from its complexity, property 3 was somewhat more difficult to prove. We provided an auxiliary $Mur\phi^{--}$program that, given a white accessible node, finds the witness to the existential quantifier in the consequent.

We were able to prove this algorithm correct in about seven days. The machine time required to prove the final version of the garbage collection algorithm is about three hours. Finding appropriate abstraction predicates took much of the time, and required an understanding of the algorithm. Typically we would start with some invariants which seemed should hold in the system as part of the abstract state. More often than not, the proof process would generate traces where the candidate invariant would fail. This mostly happened because of two reasons:

- We left out some "obvious" axiom about *acc*.
- The invariant does not hold under some situations and needed to be tweaked to get it right. This either needed changing the predicate or adding other predicates.

During the proof process we also discovered some bugs which were accidentally added while coding the algorithm. Of course, much of the human time was spent figuring out what the axioms should be and how to prove them.

5 Conclusions

Based on the experiences reported here, we believe that predicate abstraction can be a very cost-effective verification technique for non-finite problems such as parameterized systems.

Predicate abstraction could be regarded as an infinite-state alternative to model checking. However, we believe it would be most valuable in as a method for checking or strengthening invariants in a larger verification effort involving other tools, especially interactive theorem provers.

The $Mur\phi^{--}$ verifier is a prototype for evaluating ideas, not a polished tool. To be generally useful, every aspect of the $Mur\phi^{--}$ system needs additional work (including a name change). In particular, there is a need for better support for quantifiers, and more generally efficient and powerful decision procedures.

6 Acknowledgments

We are grateful to Mahadevan Subramaniam for suggesting the use of predicate abstraction, Hassen Saïdi for his help in understanding the abstraction methodology, Klaus Havelund for telling us about the concurrent garbage collection algorithm, Clark Barrett for his help in integrating SVC libraries, and Shankar Govindraju for his help with the CMU-BDD package.

References

1. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California, November 6–8.
2. M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.
3. M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, 1998.
4. E. W. Dijkstra, L. Lamport, A. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–75, November 1978.
5. D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. *Proc. ACM Symp. on Principles of Programming Languages*, January 1994.
6. D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. *Proc. ACM Symp. on Principles of Programming Languages*, January 1993.
7. E. A. Emerson and K. S. Namjoshi. Reasoning about rings. *Proc. ACM Symp. on Principles of Programming Languages*, 1995.
8. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3), July 1992.
9. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Conference on Computer Aided Verification*, volume 1254 of *Lecture notes in Computer Science*, pages 72–83. Springer-Verlag, 1997. June 1997, Haifa, Israel.
10. K. Havelund. Mechanical verification of a garbage collector. Unpublished manuscript, 1996.
11. K. Havelund and N. Shankar. A mechanized refinement proof for a garbage collector. Unpublished manuscript, 1997.
12. D. Lessens and H. Saïdi. Automatic verification of parameterized networks of processes by abstraction. *Electronic Notes of Theoretical Computer Science (ENTCS)*, 1997.
13. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.
14. S. Park and D. L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.
15. C. Pixley. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing*, 3(1):41–50, 1988.
16. D. M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6(4):359–390, 1994.
17. A. P. Sistla and S. M. German. Reasoning with many processes. *Symp. on Logic in Computer Science, Ithaca*, pages 138–152, June 1987.
18. J. van de Snepscheut. Algorithms for on-the-fly garbage collection revisited. *Information Processing Letters*, 24(4):211–16, March 1987.