Successive Approximation of Abstract Transition Relations *

Satyaki Das and David L. Dill
Computer Systems Laboratory,
Stanford University, CA 94305
satyaki@theforce.stanford.edu, dill@cs.stanford.edu

Abstract

Recently we have improved the efficiency of the predicate abstraction scheme presented in [7]. As a result the number of validity checks needed to prove the necessary verification condition has been reduced. The key idea is to refine an approximate abstract transition relation based on the counter-example generated. The system starts with an approximate abstract transition relation on which the verification condition (in our case this is a safety property) is model checked. If the property holds then the proof is done. Otherwise the model checker returns an abstract counter-example trace. This trace is used to refine the abstract transition relation if possible and start anew. At the end of the process the system either proves the verification condition or comes up with an abstract counter-example trace which holds in the most accurate abstract transition relation possible (with the user provided predicates as a basis). If the verification condition fails in the abstract system then either the concrete system does not satisfy it or the abstraction predicates chosen are not strong enough. This algorithm has been used on a concurrent garbage collection algorithm and a secure contract signing protocol. This method improved the performance on the first problem significantly and allowed us to tackle the second problem which the previous method could not handle.

1 Introduction

Abstraction is emerging as the key to formal verification of large designs, especially those that are not finite-state. *Predicate Abstraction* provides the potential for combining the generality of theorem proving with the ease-of-use of model checking 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 *abstraction predicates*, which can be supplied by the user or derived from the problem using heuristics [4].

The user must supply a *verification condition* that is to be proved. Throughout this paper, the verification condition is assumed to be an invariant. Of course more complex safety properties can be checked by augmenting the system description with history variables, and specifying an invariant over the history variables. Either the system extracts appropriate predicates or uses user provided abstraction predicates to automatically construct an abstract system from the concrete system description.

Model checking techniques can then be used to check whether the abstract system satisfies the verification condition. 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 verification condition fails to hold on the abstract system, it may or may not hold on the concrete system.

The prototype system described here handles more complex system descriptions than methods previously described. It uses two existing libraries: SVC [2], an implementation of decision procedures for quantifier-free first-order logic, and Boolean Decision Diagrams (called BDDs), an efficient representation for Boolean functions. The use of these efficient libraries is crucial for the success of the system. For example, SVC is typically called tens of thousands of times during verification.

The prototype works in two phases: it first produces a representation of a finite-state machine that is a conservative abstraction of the concrete system. Creating a good abstract machine is expensive, so an over-approximation of the abstract transition relation is computed. In the second phase, the verification condition is checked on this machine using a variant of standard BDD-based model checking algorithms. If the verification condition holds then the proof is complete. Otherwise an abstract counter-example trace is generated. This counter-example is checked to see

*This work was supported by NASA contract NAS1-98139 and DARPA contract 00-C-8015. The content of this paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

whether it is an artifact of the approximation during the first phase. If it is, then the abstract transition relation is refined (by adding constraints to the transition relation) so as to eliminate the spurious counter-example and the verification condition is model checked once again. This process is repeated until the verification condition is proved or a valid abstract counter-example is generated. This *counter-example guided refinement* phase is essential to speed up the predicate abstraction process.

The technique has been applied to a concurrent garbage collection algorithm and a contract signing protocol. The new technique was able to verify the garbage collection algorithm much faster than the technique used by Das, Dill, and Park in 1999 [7], which was the first and still only attempt to verify it using predicate abstraction. The original method could not even prove the contract signing protocol because the proof obligations generated were too difficult for the decision procedure.

Related work

The use of automatic predicate abstraction for model checking infinite-state systems was first presented by Graf and Saïdi in 1997 [9]. Their method represented the abstract states as *monomials* (monomials are conjunctions of abstract state variables or their negations). Compared with the original method of Das, Dill, and Park, and the new method, the use of monomials may result in more false errors and failed proofs. However their method also requires fewer validity checks. The original Graf/Saïdi method computes the reachable state set as part of the abstraction process. Our work uses some of the ideas present in the Graf/Saïdi abstraction scheme [9] and [7].

The creation of the initial abstract transition relation is similar to the abstraction method presented by Saïdi and Shankar [15]. In that work the authors construct an accurate abstract transition relation that is used in model checking. If the desired invariant does not hold, then new predicates are added. In their paper, refinement is used to construct the new abstract transition relation from the original relation. Their method computes the exact abstract transition relation which can be expensive. In contrast our strategy of successive approximation is more efficient because it attempts to compute the *least* accurate approximation that gives a definitive answer.

Colón and Uribe have also described a method that first generates an abstract transition system, then model checks it [6]. The transition relation generated is less accurate than that presented here.

The idea of counter-example-guided refinement is a generally useful technique in model checking, which has been used before, by Kurshan *et al.* [13] for checking timed automata, Balarin *et al.* [1] for language containment and Clarke *et al* [5] in the context of verification using abstraction for different variables in a version of the SMV model checker. Counter-example guided refinement has even been used with predicate abstraction by Lakhnech *et al.* [18]. However, their method refines by discovering new predicates to add, an idea that is quite different from refining the *use* of a given set of predicates in the abstract system.

We believe that the present method can handle significantly larger problems than previous methods. So far as we know, the original method of Das, Dill and Park is able to handle more difficult problems than any of the other methods described above, and the new method is much more efficient.

2 Abstraction Method

This section summarizes the theory of *conservative* abstraction. Since the theory behind this is well known and descriptions of this can be found in previous papers on this subject (for instance in [9]), the important properties of the abstraction will mostly be stated without formal proof. In stating and proving the claims, we have found that using logical formulas uniformly, instead of a mix of set and logic notation, eliminates a certain amount of confusion. Hence initial states, transition relations and reachable state sets are represented as predicates.

The key idea in conservative abstraction is that the abstract state machine yields a superset of the reachable concrete states. This means that if the verification condition holds in the superset of the reachable concrete states then it will also hold in the concrete system.

The concrete transition system consists of initial states represented by the predicate I_C. $I_C(x)$ is $true$ iff x is an initial state. The transition system is represented by $R_C(x, y)$. $R_C(x, y)$ is $true$ iff y is a successor of x.

The concrete system is mapped to an abstract transition system. If there are N abstraction predicates, $\phi_1, \phi_2, \ldots \phi_N$, then the abstract state space is the subset of all bit-vectors of length N, which can be modeled as follows. If $P = \{x \in N \mid 0 < x \leq N\}$, then the type of these bit-vectors is $P \to \{0, 1\}$. In what follows 1 and 0 shall be interpreted as $true$ and *false* in the obvious way. The initial states and the transition relation for the abstract system are constructed later in the section.

The abstraction can be formalized as a standard Galois connection, having an abstraction function, α which maps concrete states to bit-vectors, and a concretization function, γ which is essentially the inverse image of α. Specifically, $\alpha(x)$ is a bit-vector whose i^{th} bit has the truth value $\phi_i(x)$ while $\gamma(s)$ is a predicate on concrete states that hold on x when for every $i \in P$ the i^{th} bit of s matches $\phi_i(x)$.

Definition 1 *The abstraction and concretization functions,*

$\alpha : C \to (P \to \{0,1\})$ *and* $\gamma : (P \to \{0,1\}) \to C$ *are defined as,*

$$\alpha(x)(i) = \phi_i(x)$$
$$\gamma(s)(x) = \bigwedge_{i \in P} \phi_i(x) \equiv s(i)$$

($\equiv$ is the biconditional)

The definition of α and γ can be extended to work on the predicates defined over the concrete states and abstract states respectively. These extended definitions are as follows:

Definition 2 *Given predicates, Q_C and Q_A over concrete and abstract states respectively, the abstraction and concretization functions are extended as follows:*

$$\begin{aligned} \alpha(Q_C)(s) &= \exists x.\, Q_C(x) \wedge \bigwedge_{i \in P} \phi_i(x) \equiv s(i) \\ \gamma(Q_A)(x) &= \exists s.\, Q_A(s) \wedge \bigwedge_{i \in P} \phi_i(x) \equiv s(i) \end{aligned}$$

Predicates are used to describe sets. So the set of all abstract states are defined by the predicate, $\exists x.\ \gamma(s)(x)$. Then for any arbitrary predicates S and X defined on the abstract and concrete states respectively it can be easily proved that,

$$X \to \gamma(\alpha(X))$$
$$(\exists x.\ \gamma(S)(x)) \to (S = \alpha(\gamma(S)))$$

These two results show that the abstraction scheme is indeed a Galois connection.

Definition 3 *The set of abstract initial states, I_A is defined to be $\alpha(I_C)$.*

Notice that α has been used on a concrete predicate and so the second definition of α is to be used. It may be shown that the concrete and abstract initial states satisfy the inclusion relation, $I_C \to \gamma(I_A)$

Definition 4 *The abstract transition relation is represented by a predicate R_A with two states, s and t as arguments. The transition relation is defined as,*

$$R_A(s,t) = \exists x, y.\ \ \gamma(s)(x) \wedge \gamma(t)(y) \wedge R_C(x,y)$$

The abstract transition system so defined is a conservative abstraction of the concrete system. Let the predicate $S_A^k(s)$ hold if s is an abstract state that is reachable from an initial state after k transitions. Similarly let the predicate $S_C^k(x)$ hold if x is a concrete state that is reachable from an initial state after k transitions. Assuming that

$$\forall x.\ \ S_C^k(x) \to \gamma(S_A^k)(x) \qquad (1)$$

holds it can easily be shown that

$$\forall x.\ \ S_C^{k+1}(x) \to \gamma(S_A^{k+1})(x)$$

where the reachable concrete and abstract states after $k+1$ transitions are given by

$$S_C^{k+1}(y) = S_C^k(y) \vee \exists x.\ S_C^k(x) \wedge R_C(x,y)$$
$$S_A^{k+1}(t) = S_A^k(t) \vee \exists s.\ S_A^k(s) \wedge R_A(s,t)$$

Then by induction it may be concluded that (1) holds for all k. Since the abstract system is finite, the fixed point of abstract reachable states exists and the concretization of the abstract reachable states must include all concrete reachable states. This shows that any invariant that holds in the concretization of the abstract reachable states must also hold in the concrete system. Thus the abstract system is a conservative abstraction of the concrete system.

3 Counter-Example Guided Refinement

Now that the abstract system has been defined, a method is presented to compute the abstract system efficiently and with the necessary accuracy. Usually, computing the exact abstract transition relation defined in the previous section requires excessive time for all but the most trivial of systems. Also typically the set of abstract reachable states is extremely sparse. So most of the abstract states are unreachable. Hence computing the full transition relation is not necessary.

Assume that the successive approximation process starts with an over-approximation, R_0, of the exact abstract transition relation. If a state t is a successor of s in the exact transition relation then t is also a successor of s in the over approximated transition relation as well. R_0 is used to model check the verification condition. If the verification condition holds then the proof is complete. Otherwise the model checker generates an *abstract counter-example trace* which violates the verification condition. The abstract counter-example trace is a finite sequence of abstract states, $s_0, s_1, \ldots s_n$ such that $I_A(s_0)$ holds and $R_0(s_i, s_{i+1})$ holds for every $i \in [0, n)$. Also s_n violates the verification condition. Now, for each pair of consecutive abstract states, (s_i, s_{i+1}), check if $R_A(s_i, s_{i+1})$ holds. In this case, a valid abstract counter-example has been found. Otherwise R_0, can be refined to eliminate the generated counter-example. This process of model checking followed by refinement is repeated till the verification condition is proved or a valid counter-example is found.

We now explain how the refinement process works. Suppose R is the an over approximated abstract transition relation and that the abstract counter-example trace found after model checking has two consecutive states, s_j and s_{j+1}, such that $R_A(s_j, s_{j+1})$ is *false*. The algorithm tries to find

```
PROVE_VERIFICATION_CONDITION(property)
begin
  I_A := Initial State predicate
  R_A := true
  while (true)
    R_orig := R_A
    trace := model check property in abstract system, (I_A, R_A)
    if empty(trace) then
      return PROPERTY_PROVED
    else
      for each pair of successive states s_j, s_{j+1} in trace do
        if γ(s_j)(x) ∧ γ(s_{j+1})(y) ∧ R_C(x, y) is unsatisfiable
        then
          R_orig := R_A
          R_A := R_A ∧ REFINE_TRANS_REL(s_j, s_{j+1})
          break
        endif
      end
      if R_A = R_orig return trace
    endif
  end
end

REFINE_TRANS_REL(s_j, s_{j+1})
/* The function returns the constraint C */
begin
  X := γ(s_j)(x) ∧ γ(s_{j+1})(y)
  for each conjunct, p in X do
    remove p from X
    if satisfiable(X ∧ R_C(x, y)) then
      add p back to X
    endif
  end
  return ¬α(X)
end
```

Figure 1. Abstract State Machine Refinement

a constraint, $C(s,t)$, such that $R_A(s,t) \rightarrow C(s,t)$ and $C(s_j, s_{j+1})$ is *false*. Then the abstract transition relation,

$$R'(s,t) = R(s,t) \wedge C(s,t)$$

is also a conservative abstract transition relation. Since $R_A(s_j, s_{j+1})$ is *false*, this means that $\gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \wedge R_C(x,y)$ is unsatisfiable for every x and every y. From the definition of γ, it follows that $\gamma(s_j)(x) \wedge \gamma(s_{j+1})(y)$ is a conjunction of abstraction predicates, $\phi_i(x)$ and $\phi_i(y)$ and their logical complements. We wish to find a minimal subset of these predicates that is unsatisfiable when conjoined with $R_C(x,y)$. The heuristic in the present system is a simple greedy algorithm. It is explained in Figure 1.

The following theorem shows that this construction results in a new conservative abstract transition relation. The key point to note is that at the end of the algorithm the conjunction of the remaining conjuncts and $R_C(x,y)$ is unsatisfiable. The bit-vectors c_j and c_{j+1} determine which conjuncts have been removed. Wherever $c_j(k)$ is *false*, the conjunct involving $\phi_k(x)$ has been removed from $\gamma(s_j)(x)$ in the added constraint, $C(s,t)$. Similarly, if $c_{j+1}(k)$ is false, then the conjunct involving $\phi_k(y)$ has been removed from $\gamma(s_{j+1})(y)$.

Theorem 1 *Let the initial abstract transition relation, R satisfy $\forall s,t.\ R_A(s,t) \rightarrow R(s,t)$ and s_j, s_{j+1} be abstract states and c_j and c_{j+1} are bit-vectors such that*

$$\begin{aligned} & \bigwedge_{i \in P} c_j(i) \rightarrow (s_j(i) \equiv \phi_i(x)) \\ \wedge \quad & \bigwedge_{i \in P} c_{j+1}(i) \rightarrow (s_{j+1}(i) \equiv \phi_i(y)) \wedge R_C(x,y) \end{aligned}$$

is unsatisfiable, then the new transition relation defined by,

$$\begin{aligned} R'(s,t) \quad = \quad & R(s,t) \wedge \\ & \neg[\bigwedge_{i \in P} c_j(i) \rightarrow (s(i) \ \equiv \ s_j(i)) \wedge \\ & \bigwedge_{i \in P} c_{j+1}(i) \rightarrow (t(i) \ \equiv \ s_{j+1}(i))] \end{aligned}$$

satisfies

$$\forall s,t.\ R_A(s,t) \rightarrow R'(s,t)$$

Proof To prove the theorem assume that $R_A(s,t)$ holds for some arbitrary s and t.

Since $R_A(s,t) \rightarrow R(s,t)$, it may be concluded that $R(s,t)$ holds as well. Also by definition of R_A,

$$\exists x,y.\ \gamma(s)(x) \wedge \gamma(t)(y) \wedge R_C(x,y)$$

Existential instantiation of the quantifier and using the definition of γ yields,

$$\bigwedge_{i \in P} s(i) \equiv \phi_i(x_0) \wedge \bigwedge_{i \in P} t(i) \equiv \phi_i(y_0) \wedge R_C(x_0,y_0) \qquad (2)$$

Because of the condition that c_j and c_{j+1} satisfies,

$$\begin{aligned} \neg[\exists x,y. \quad & \bigwedge_{i \in P} c_j(i) \rightarrow (s_j(i) \equiv \phi_i(x)) \wedge \\ & \bigwedge_{i \in P} c_{j+1}(i) \rightarrow (s_{j+1}(i) \equiv \phi_i(y)) \wedge R_C(x,y)] \end{aligned}$$

Simplifying the expression and then instantiating with x_0 and y_0 yields,

$$\begin{aligned} & [\bigvee_{i \in P} c_j(i) \wedge (s_j(i) \not\equiv \phi_i(x_0))] \\ \vee \quad & [\bigvee_{i \in P} c_{j+1}(i) \wedge (s_{j+1}(i) \not\equiv \phi_i(y_0))] \\ \vee \quad & \neg R_C(x_0,y_0) \end{aligned}$$

Using the expressions for $\phi_i(x_0)$ and $\phi_i(y_0)$ from (2) yields,

$$\begin{aligned} & \bigvee_{i \in P} c_j(i) \wedge (s_j(i) \not\equiv s(i)) \\ \vee \quad & \bigvee_{i \in P} c_{j+1}(i) \wedge (s_{j+1}(i) \not\equiv t(i)) \end{aligned} \tag{3}$$

Now from the definition of R',

$$\begin{aligned} R'(s,t) \quad = \quad & R(s,t) \wedge \\ & \neg[\bigwedge_{i \in P} c_j(i) \rightarrow (s(i) \equiv s_j(i)) \wedge \\ & \bigwedge_{i \in P} c_{j+1}(i) \rightarrow (t(i) \equiv s_{j+1}(i))] \end{aligned}$$

Simplifying the above definition and using that $R(s,t)$ holds,

$$\begin{aligned} R'(s,t) \quad = \quad & [\bigvee_{i \in P} c_j(i) \wedge (s(i) \not\equiv s_j(i)) \vee \\ & \bigvee_{i \in P} c_{j+1}(i) \wedge (t(i) \not\equiv s_{j+1}(i))] \end{aligned} \tag{4}$$

The combination of (4) and (3) shows that $R'(s,t)$ holds. This completes the proof of the theorem. □

As mentioned above, the approximate abstract system is model checked, and then refined if necessary. This process is repeated until one of the following happens:

1. The verification condition holds.

2. A counter-example trace in which for any two successive states, s_j and s_{j+1},
   $\exists x,y.\ \gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \wedge R_C(x,y)$
   holds.

It is easy to see that the process will necessarily terminate in one of these situations. Every refinement must remove at least one pair of abstract states from the transition relation. Since the abstract system is finite, the number of times the refinement can be carried out is bounded.

In the first scenario the desired invariant holds in an over-approximation of the exact abstract transition relation and so would also hold in the exact transition relation. Thus the desired invariant has been proved correct. In the second case the counter-example generated would also hold in the abstract machine with transition relation R_A. So further refinement of R_A would be useless. This is proved in the next theorem.

Theorem 2 *If an abstract transition system with transition relation, R such that $R_A \rightarrow R$ and initial state set, I_A has a counter-example trace, $s_0, s_1, \ldots s_n$ such that for each $j \in [0,n)$ there are concrete states x and y (not necessarily the same for different values of j) such that,*

$$\gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \wedge R_C(x,y)$$

is satisfiable, then $s_0, s_1, \ldots s_n$ is also a counter-example trace in the abstract transition system where the transition relation is R_A and the initial state set is I_A.

Proof Since $s_0, s_1, \ldots s_n$ is an execution trace in the approximate transition system,

$$I_A(s_0) \tag{5}$$

Now for every $j \in [0,n)$,

$$\begin{aligned} R_A(s_j, s_{j+1}) \quad = \exists x,y. \quad & \gamma(s_j)(x) \wedge \gamma(s_{j+1})(y) \\ & \wedge\ R_C(x,y) \end{aligned} \tag{6}$$

Existential instantiation of the precondition of the theorem yields,

$$\gamma(s_i)(x_0) \wedge \gamma(s_{i+1})(y_0) \wedge R_C(x_0,y_0)$$

Using this with (6) implies that $R_A(s_j, s_{j+1})$ is $true$ and so s_{j+1} is a successor of s_j. Using this fact in conjunction with (5) proves that $s_0, s_1, \ldots s_n$ is a counter-example trace in the exact abstract system. □

Thus, if a counter-example is generated, either the set of predicates provided are not rich enough to prove the desired verification condition or the invariant does not hold in the concrete system.

4 Prototype Implementation and Results

A prototype verifier based on the preceding ideas was implemented to evaluate efficiency on real problems. The decision procedure, SVC was used to do the satisfiability checks. Binary Decision Diagrams were used to represent the abstract transition relation and to model check the verfication condition on the abstract system. The user has to provide the predicates used to construct the abstract system.

An obvious choice for the initial approximate abstract transition relation is the completely unconstrained abstract transition relation. The decision procedure, SVC, did not perform well when this was the case, so the prototype produced an initial approximation by heuristically collecting small sets of predicates with many common variables, and building a abstract transition relation using only those predicates.

Unlike the preceding discussion, the prototype creates abstraction predicates on the next-state variables by substituting transition functions for current state variables in the abstraction functions (this is the method used in most previous papers on predicate abstraction).

We have used two examples to evaluate the successive approximation method presented here. The examples are:

- On-The-Fly Garbage Collection
- GJM Secure Contract Signing Protocol

On-The-Fly Garbage Collection

The on-the-fly garbage collection algorithm was proposed by Dijkstra, et al. [8]. This 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 [10].

The algorithm was simplified by Ben-Ari [3] to involve two colors instead of three. This also led to a simpler argument of correctness. Alternative justifications of Ben-Ari's algorithm were also given by Van de Snepscheut [17] and Pixley [12]. However it must be remembered that these proofs were informal *pencil and paper* proofs.

Later this modified algorithm was mechanically proved by Russinoff [14] using the Boyer-Moore theorem prover. A formulation of the same algorithm was also proved by Havelund and Shankar in PVS [10]. The authors give an estimation of the complexity and size of the proof. The proof needed 19 invariant lemmas and 57 function lemmas and 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 user program, the *mutator*, may be regarded as a concurrent system with both processes working on 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* and they are special in the sense that they are always accessible to the mutator. Also any node that can be reached from one of the roots by following edges is also accessible to the mutator. The mutator is allowed to choose an arbitrary node and redirect one of its edges towards another 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 also maintains a *free-list* which is a list of nodes that are not being used by the mutator. The mutator can request nodes from the collector which the collector satisfies from the free-list. The collector collects *garbage* nodes (that is nodes which are no longer accessible to the mutator) and adds them to the free-list.

The garbage collection algorithm must satisfy two properties for it to be correct. First it must guarantee that no node accessible to the mutator is ever added to the free-list. The second property is that if some node becomes inaccessible to the mutator it is eventually added to the free-list. The first property makes sure that no data which would be used by the user program is ever freed. The second property makes sure that there are no memory leaks in the system. We have proved that the first property holds for the algorithm using predicate abstraction. The proof of correctness needs some auxillary graph properties which are treated as axioms by the predicate abstraction tool.

GJM Abuse-Free Contract Signing Protocol

The abuse-free contract signing protocol provides a mechanism for signing contracts between two parties and guarantees some correctness properties. A *contract* can be thought of as reciprocal promises between the involved parties. For instance if Alice is buying a car from Bob then she promises to pay Bob the negotiated price while he promises to give her the car.

A very basic correctness condition is *fairness*. For a contract signing protocol to be fair it must be the case that after the protocol terminates either both parties have a contract or neither party has a contract. In the previous example if Alice promises to pay the price of the car she should have a promise from Bob that he would give her the car. Otherwise the protocol violates fairness.

Other correctness properties of the protocol are *accountability* and *abuse-freeness*. We have not proved these properties.

The protocol we have studied here was introduced in [11]. The protocol depends on a *trusted third party* to resolve conflicts. The protocol works in two phases. In the first phase the participants exchange messages and try to arrive at a contract. If something goes wrong (either because messages were lost or because of foul play) the trusted third party resolves the contract. The protocol has been exhaustively analyzed for weaknesses using a model checker [16] with a finite number of concurrent contract signings. A problem was discovered during this and was fixed. We have looked at the fixed protocol and proved that it maintains fairness with any number of concurrent contract signings.

Results

For each example, the execution times on a 800MHz Pentium processor are reported. In the table below the abstraction time is the time required to compute the initial approximate transition relation. The model checking time is the time required to repeatedly model check and refine the abstraction. The time required is compared to the approach presented in implicit predicate abstraction [7].

One reason that the current method works much better than implicit predicate abstraction is that it never has to check the satisfiability of similar expressions repeatedly. To see why this can be a problem with implicit predicate abstraction consider the following example. Assume that we have abstraction predicates $\phi_1 \equiv a > b$ and $\phi_2 \equiv b > a$ (where a and b are concrete state variables). It is obvious that both predicates can not be true at the same

	Abstraction time (in hr:min)	Model checking time (in min)
GC(implicit)	2:25	N/A
GC(current)	0:09	1
GJM(implicit)	24hr+	N/A
GJM(current)	0:13	4

time. In the implicit abstraction scheme, expressions, which are unsatisfiable because they are conjunctions containing $\phi_1(x) \wedge \phi_2(x)$, are checked for satisfiability repeatedly. In the current method this will be recognized the first time a counter-example has both predicates true. After that the abstract transition relation will be suitably modified so that a counter-example is never generated which has both predicates asserted simultaneosly.

Another interesting observation is that the set of reachable abstract states is usually extremely sparse. So the current method will perform much better than systems which naively compute an exact abstract transition system.

If the verification condition can be proved with the provided abstraction predicates then the current method will indeed be able to prove the verification condition. Thus if the proof fails then that means that the set of abstraction predicates is not enough to prove the verification condition. In systems which construct a weaker abstraction, a failed proof has to be investigated to determine if the proof failed because the abstraction predicates are insufficient or because the approximation lost information.

5 Conclusion

This paper demonstrates that using counter-example guided refinement with predicate abstraction can reduce the computational difficulty of formally verifying systems with unbounded numbers of states. However, we have only done a few examples of any size, and there are obviously many additional problems that would need to be solved before predicate abstraction could be used as routinely as model checking is currently.

The most obvious issue at this point is the need to find good candidate predicates automatically, instead of requiring the user to provide them. This problem has been addressed to some extent by others (as discussed in section 1), but it is not clear that the techniques would scale up to the size of problems in the previous section. Automatically deriving excessively complex predicates or too many irrelevant predicates could make the computational part of predicate abstraction too difficult. Another important issue is how to find good candidate predicates containing quantifiers, which are needed for the examples in the previous section.

Another difficult issue is how to discover when there are design errors. A good pragmatic step would be to model check a finite instance of the problem before applying predicate abstraction. But feasible finite instances may not exhibit the errors (which is the motivation for doing predicate abstraction in the first place). In the system described here, errors will result in valid abstract counter-examples, but there is no algorithmic way to determine if these correspond to a concrete counter-example, which is what the user really needs to determine whether the problem is a design error or an inadequate abstraction. Of course, the problem is undecidable, so there is no perfect solution, but there may be good heuristics for finding useful counter-examples.

References

[1] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *5th International Conference on Computer-Aided Verification*, pages 29–40, 1993.

[2] 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.

[3] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 6(3):333–344, July 1984.

[4] S. Bensalem, Y. Lakhnech, and S. Owre. Invest: A tool for the verification of invariants. In *10th International Conference on Computer-Aided Verification*, pages 505–510, 1998.

[5] E. *et al.*. Clarke. Counterexample-guided abstraction refinement. In *12th International Conference on Computer-Aided Verification*, pages 154–169, July 2000.

[6] 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.

[7] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *11th International Conference on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.

[8] 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.

[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 and N. Shankar. A mechanized refinement proof for a garbage collector. Available at http://ic-www.arc.nasa.gov/ic/projects/amphion/people/havelund, 1997.

[11] M. J. J. A. Garay and P. MacKenzie. Abuse-free optimistic contract signing. In *Proc. Advances in Cryptology - Crypto '99*, pages 449–466, 1999.

[12] C. Pixley. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing*, 3(1):41–50, 1988.

[13] R. K. R. Alur, A. Itai and M. Yannakakis. Timing verification by successive approximation. *Information and Computation 118(1)*, pages 142–157, 1995.

[14] D. M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6(4):359–390, 1994.

[15] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *11th International Conference on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.

[16] V. Shmatikov and J. C. Mitchell. Analysis of abuse-free contract signing. In *Financial Cryptography*, 2000. Anguilla.

[17] J. van de Snepscheut. Algorithms for on-the-fly garbage collection revisited. *Information Processing Letters*, 24(4):211–16, March 1987.

[18] S. B. Y. Lakhnech, S. Bensalem and S. Owre. Incremental verification by abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2001.