

Counter-Example Based Predicate Discovery in Predicate Abstraction ^{*}

Satyaki Das and David L. Dill

Computer Systems Laboratory
Stanford University
satyakid@stanford.edu, dill@cs.stanford.edu

Abstract. The application of predicate abstraction to parameterized systems requires the use of quantified predicates. These predicates cannot be found automatically by existing techniques and are tedious for the user to provide. In this work we demonstrate a method of discovering most of these predicates automatically by analyzing spurious abstract counter-example traces. Since predicate discovery for unbounded state systems is an undecidable problem, it can fail on some problems. The method has been applied to a simplified version of the Ad hoc On-Demand Distance Vector Routing protocol where it successfully discovers all required predicates.

1 Introduction

Unbounded state systems have to be reasoned about to prove the correctness of a variety of real life systems including microprocessors, network protocols, software device drivers and security protocols. *Predicate Abstraction* is an efficient way of reducing these infinite state systems into more tractable finite state systems. A finite set of *abstraction predicates* defined on the concrete system is used to define the finite-state model of the system. The states of the abstract system consist of truth assignments to the set of abstraction predicates, that is each predicate is assigned a value of *true* or *false*. The abstraction is conservative, meaning that for any property proved on the abstract system, a concrete counterpart holds on the actual system.

There are many hard problems that need to be solved to make predicate abstraction useful. The first is that the problem of proving arbitrary safety properties of a transition system is (obviously) undecidable.

Given a pre-selected set of predicates and certain other assumptions, it is possible to prove in some cases that the system satisfies a safety property, but a failed proof may indicate that the property is violated, or simply that the abstraction is not sufficiently precise to complete the proof. Automating such

^{*} This work was supported by National Science Foundation under grant number 0121403 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.

proofs is quite difficult in practice, since it involves automatically solving logic problems that have high complexity and searching potentially large state spaces. In spite of the difficulty of this problem, there has been substantial progress towards solving it in the last few years.

Another problem is how to discover the appropriate set of predicates. In much of the work on predicate abstraction, the predicates were assumed to be given by the user, or they were extracted syntactically from the system description (for example, predicates that appear in conditionals are often useful). It is obviously difficult for the user to find the right set of predicates (indeed, it is a trial-and-error process involving inspecting failed proofs), and the predicates appearing in the system description are rarely sufficient. There has been less work, and less progress, on solving the problem of finding the right set of predicates. In addition to the challenge of finding a sufficient set of predicates, there is the challenge of avoiding irrelevant predicates, since the cost of checking the abstract system usually increases exponentially with the number of predicates.

In our system quantified predicates are used to deal with parameterized systems. In a parameterized system, it is often interesting (and necessary) to find properties that hold for all values of the parameter. For instance if a message queue is modeled as an array and rules parameterized by the array index are used to deliver messages then the absence of certain kinds of messages is expressed by a universally quantified formula. So predicates with quantifiers in them are used.

This paper describes new ways of automatically discovering useful predicates by diagnosing failed proofs. The method is designed to find hard predicates that do not appear syntactically in the system description, including quantified predicates, which are necessary for proving most interesting properties. As importantly, it tries to avoid discovering useless predicates that do not help to avoid a known erroneous result. Furthermore, the diagnosis process can tell when a proof fails because of a genuine violation of the property by the actual system.

Implementation

The system was implemented using Binary Decision Diagrams (BDD) to represent the abstract system. A decision procedure for quantifier-free first-order logic, CVC [1] was used to do the satisfiability checks. The system is built around the predicate abstraction tool described in Das and Dill [9].

The state variable declarations describe the state of the concrete system. The transition relation is described using a list of parameterized guarded commands. Each guarded command consists of a guard and an action. The guard is a logic formula over the state variables and possibly the parameters that evaluates to either *true* or *false*. Each of the actions is a procedure that modifies the current concrete state into a new value. At each point the action corresponding to one of the enabled rules (rules whose guards evaluate to *true*) is non-deterministically executed and the concrete state changes.

The prototype is implemented as shown in Figure 1. The upper block is the tool described in our previous work [9]. Given a set of abstraction predicates,

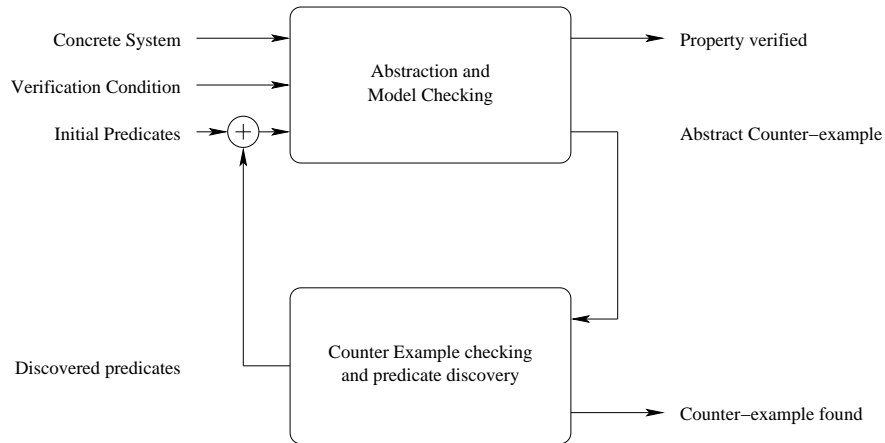


Fig. 1. Predicate Abstraction Algorithm

a verification condition and the concrete system description it first computes an approximate abstract model. This abstract model is model checked and the abstract system refined appropriately if it was too inexact. Notice that this refinement does not change the set of abstraction predicates and concentrates on using the existing predicates more efficiently. Finally this process terminates with either the verification condition verified (in which case nothing else needs to be done) or with an abstract counter-example trace.

The current work, represented by the lower block in the diagram, checks whether a concrete counter-example trace corresponding to the abstract trace exists. If so the verification condition is violated and an error is reported, otherwise new predicates are discovered which avoids this counter-example.

The new predicates are added to the already present abstraction predicates and the process starts anew. Since all the old predicates are reused a lot of the work from previous iterations are reused.

Related Work

Recently a lot of work has been done on predicate abstraction. The use of automatic predicate abstraction for model checking infinite-state systems was first presented by Graf and Saïdi in 1997 [11]. The method used *monomials* (monomials are conjunctions of abstract state variables or their negations) to represent abstract states. Parameterized systems are handled by using a counting abstraction [13]. Similar work has also been proposed in [17] and [14]. In 1998 [8], Colón and Uribe describes a method of constructing a finite abstract system and then model checking it. The abstraction produced in both methods are coarse and could fail to prove the verification condition even if all necessary predicates were present. By constructing the abstraction in a demand driven fashion, the method of Das and Dill [9] is able to compute abstractions efficiently that are as precise

as possible given a fixed finite set of predicates. This ensures that if the desired properties can be proved with the abstraction predicates then the method will be able to do so. The predicate abstraction methods described so far have relied on user provided predicates to produce the abstract system.

Counter-example guided refinement is a generally useful technique. It has been used in by Kurshan *et al.* [2] for checking timed automata, Balarin *et al.* [3] for language containment and Clarke *et al* [7] 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.* [12]. Invariant generation techniques have also used similar ideas [19, 5]. Invariant generation techniques generally produce too many invariants many of which are not relevant to the property being proved. This can cause problems with large systems. The counter-example guided refinement techniques do not produce the quantified predicates that our method needs.

Predicate abstraction is also being used for software verification. Device drivers are being verified by the SLAM project [4]. The SLAM project has used concrete simulation of the abstract counter-example trace to generate new predicates. The BLAST project [18] also uses spurious counter-examples to generate new predicates. Predicate abstraction has also been used in software verification as a way of finding loop invariants [10]. These systems do not deal with parameterized systems, hence they do not need quantified predicates.

2 Abstraction Basics

As in previous work [9], sets of abstract and concrete states will be represented by logical formulas. For instance the concrete predicate, X represents the set of concrete states which satisfy, X . The main idea of predicate abstraction is to construct a *conservative* abstraction of the concrete system. This ensures that if some property is proved for the abstract system, then the corresponding property also holds for the concrete system.

Formally the concrete transition system is described by a set of *initial states* represented by the predicate I_C and a *transition relation* represented by the predicate R_C . $I_C(x)$ is *true* iff x is an initial state. Similarly, $R_C(x, y)$ is *true* iff y is a successor of x . The *safety property*, P is the verification condition that needs to be proved in the concrete system. An *execution* of the concrete system is defined to be a sequence of states, x_0, x_1, \dots, x_M such that $I_C(x_0)$ holds and for every $i \in [0, M)$, $R_C(x_i, x_{i+1})$ holds. A *partial trace* is an execution that does not necessarily start from an initial state. A *counter-example trace* is defined to be an execution, x_0, x_1, \dots, x_M such that $\neg P(x_M)$ holds (i.e., the counter-example trace ends in a state which violates P).

The abstraction is determined by a set of N predicates, $\phi_1, \phi_2, \dots, \phi_N$. The abstract state space is just the set of all bit-vectors of length N . An *abstraction function*, α maps sets of concrete states to sets of abstract states while the *concretization function*, γ does the reverse. In the following definitions the predicates Q_C and Q_A represent sets of concrete states and abstract states re-

spectively. Then $\alpha(Q_C)$ is a predicate over abstract states such that $\alpha(Q_C)(s)$ holds exactly when s is an abstraction of some concrete state x in Q_C . Similarly $\gamma(Q_A)(x)$ holds exactly when there exists an abstract state, s in Q_A and s is the abstraction of x .

Definition 1 *Given predicates, Q_C and Q_A over concrete and abstract states respectively, the abstraction and concretization functions are defined as:*

$$\alpha(Q_C)(s) = \exists x. Q_C(x) \wedge \bigwedge_{i \in [1, N]} \phi_i(x) \equiv s(i)$$

$$\gamma(Q_A)(x) = \exists s. Q_A(s) \wedge \bigwedge_{i \in [1, N]} \phi_i(x) \equiv s(i)$$

Using the above definitions, the abstract system is defined by the set of *abstract initial states*, $I_A = \alpha(I_C)$ and the *abstract transition relation*, $R_A(s, t) = \exists x, y. \gamma(s)(x) \wedge \gamma(t)(y) \wedge R_C(x, y)$. An *abstract execution* is a sequence of abstract states, s_0, s_1, \dots, s_M such that $I_A(s_0)$ holds and for each $i \in [0, M)$, $R_A(s_i, s_{i+1})$ holds. An *abstract counter-example trace* is an abstract execution, s_0, s_1, \dots, s_M for which $\alpha(\neg P)(s_M)$ holds.

The atomic predicates in the verification condition, P , are used as the initial set of predicates. The abstract system is constructed and the abstract property, $\neg\alpha(\neg P)$ checked for all reachable states. If this is successful then the verification condition holds. Otherwise the generated abstract counter-example is analyzed to see if a concrete execution corresponding to the abstract trace exists. In that case, a concrete counter-example has been constructed. Otherwise the abstract counter-example is used to discover new predicates. Then the process is repeated with the discovered predicates being added to the already present predicates.

An abstract trace is called a *real trace* if there exists a concrete trace corresponding to it. Conversely if there are no concrete traces corresponding to an abstract trace then it is called a *spurious trace*.

3 Predicate Discovery

As described in the previous section, the system generates a counter-example trace to the verification condition that was to be proved. Now the system must analyze the abstract counter-example trace to either confirm that the trace is real, that is a concrete trace corresponding to it exists, or come up with additional predicates which would eliminate the spurious counter-example.

First the trace is minimized to get a *minimal spurious trace*. A minimal spurious trace is defined to be an abstract trace which is

1. spurious (no corresponding concrete trace exists.)
2. minimal (removing even a single state from either the beginning or end of the trace makes the remainder real.)

Checking the Abstract Counter-Example Trace

There is a concrete counter-example trace x_1, x_2, \dots, x_L corresponding to the abstract counter-example trace, s_1, s_2, \dots, s_L if these conditions are satisfied:

1. For each $i \in [1, L]$, $\gamma(s_i)(x_i)$ holds. This means that each concrete state x_i corresponds to the abstract state s_i in the trace.
2. $I_C(x_1) \wedge \neg P(x_L)$ holds. The concrete counter-example trace starts from a initial state and ends in a state which violates P .
3. For each $i \in [1, L)$, $R_C(x_i, x_{i+1})$. For every i , x_{i+1} is the successor of x_i .

The conditions (1) and (3) determine that a concrete trace corresponding to the abstract trace exists and condition (2) determines that the trace starts from the set of concrete initial states and ends in a state that violates the verification condition.

To write the logic concisely the logic for the initial state has been disregarded. In the implementation, an initial totally unconstrained state is added to the trace and it is assumed that the initial rule produces the initial state of the system.

Since all the atomic predicates of P are present among the abstraction predicates the condition $\neg P(x_L)$ is implied by $\gamma(s_L)(x_L)$. Hence, if the formula

$$\bigwedge_{i=1}^L \gamma(s_i)(x_i) \quad \wedge \quad \bigwedge_{i=0}^{L-1} R_C(x_i, x_{i+1})$$

is satisfiable then the abstract counter-example trace is real. Otherwise there is no satisfying assignment and the abstract counter-example trace is spurious. To simplify the presentation it shall be assumed that the same transition relation, R_C can be used for each of the concrete steps including the first where R_I is actually used. In our implementation the first step is handled specially and R_I is used instead of R_C .

The test for spuriousness is completely a property of the transition relation and the trace itself and does not depend either on the initial states or the verification condition. So we will generalize the definition of spuriousness to partial traces. A partial trace is spurious if the above formula is unsatisfiable.

Predicate Discovery

To understand predicate discovery we must first understand when predicate abstraction produces a spurious counter-example. Assume that in Figure 2 the whole abstract trace s_1, s_2, \dots, s_L is spurious but the partial trace s_2, s_3, \dots, s_L is real. So there are two kinds of concrete states in $\gamma(s_2)$:

1. Successor states of states in $\gamma(s_1)$.
2. States (like x_2) that are part of some concrete trace corresponding to s_2, \dots, s_L .

It must be the case that the above two types of states are disjoint. Otherwise it would be possible to find a concrete trace corresponding to the whole trace

thereby making it real. If predicates to distinguish the two kinds of states were added then the spurious counter-example would be avoided. In the method described here, the discovered predicates will be able to characterize states of the second type above.

Once it has been determined that the abstract counter-example is spurious, states are removed from the beginning of the trace while still keeping the remainder spurious. When states can no longer be removed from the beginning, the same process is carried out by removing states from the end of the trace. This will eventually produce a minimal spurious trace.

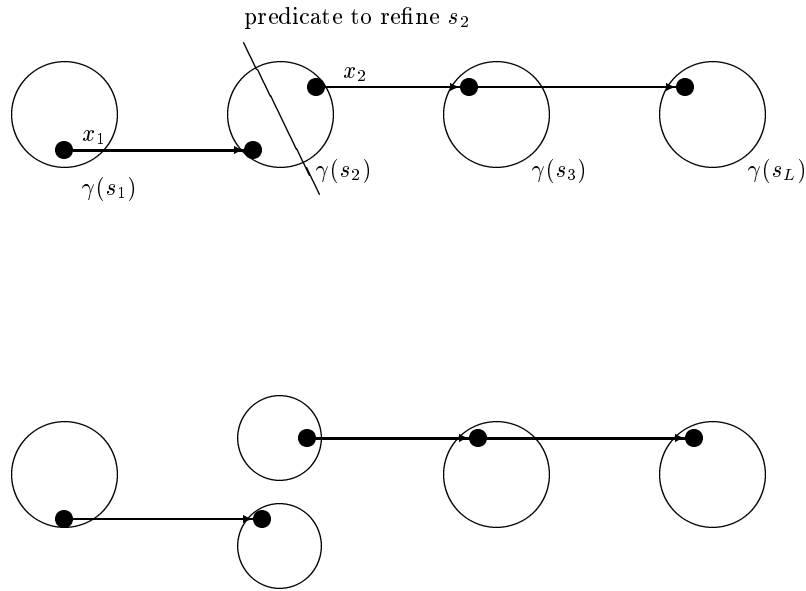


Fig. 2. Abstraction Refinement

Now consider the minimal spurious trace, $s_1, s_2, s_3, \dots, s_L$ shown in Figure 2. Here the circles representing $\gamma(s_1), \gamma(s_2)$ etc. are sets of concrete states while the black dots inside the sets represent individual concrete states. Since the trace $s_2, s_3 \dots s_L$ is real,

$$Q_0 = \bigwedge_{i=2}^L \gamma(s_i)(x_i) \wedge \bigwedge_{i=2}^{L-1} R_C(x_i, x_{i+1})$$

is satisfiable for some concrete states, x_2, x_3, \dots, x_L .

Now CVC is queried about the satisfiability of Q_0 . This returns a finite conjunction of formulas, $\psi_1(x_2) \wedge \psi_2(x_2) \wedge \dots \wedge \psi_K(x_2) \wedge \theta(x_3, \dots, x_L)$ which implies Q_0 . So the ψ_i 's are conditions that any x_2 must satisfy for it to be the first state of the concrete trace corresponding to s_2, s_3, \dots, s_L . Now it must be the case

that,

$$\gamma(s_1)(x_1) \wedge R_C(x_1, x_2) \wedge \bigwedge_{i=1}^K \psi_i(x_2) \wedge \theta(x_3, \dots, x_L)$$

is unsatisfiable. Otherwise it would be possible to find a concrete trace corresponding to s_1, s_2, \dots, s_L ! More specifically, if the predicates $\psi_1, \psi_2, \dots, \psi_K$ are added to the set of abstraction predicates, and the verifier rerun, this particular spurious abstract counter-example will not be generated. So, we have an automatic way of discovering new abstract predicates.

However it is possible to reduce the number of additional abstraction predicates. In fact it is quite likely that all of the predicates ψ_1, \dots, ψ_K are not needed to avoid the spurious counter-example. The satisfiability of the above formula is checked after leaving out the $\psi_1(x_2)$ expression. If the formula is still unsatisfiable then it is dropped altogether. The same procedure is repeated with the other ψ_i 's till an essential set of predicates remain (dropping any one of them makes the formula satisfiable). Notice that there may be multiple essential sets of predicates that make the above formula unsatisfiable. This method finds one such set.

Now consider the effect that the abstraction refinement has on the abstract system. The original abstract state, s_2 will be split into two – in one part all the added predicates hold while in the other part at least one of the assertions does not hold. Also, in the abstract transition relation, the transition from the state s_1 to the first partition of s_2 is removed. It is still possible that there is a path from s_1 to s_3 through the other partition of s_2 . However the refined abstraction will never generate a spurious counter-example in which a concrete state corresponding to s_1 has a successor which satisfies all the assertions $\psi_1, \psi_2, \dots, \psi_K$.

Parameterized rules and quantified predicates

When proving properties of parameterized systems, quantified predicates are needed. These quantified predicates cannot be found either from the system description or by existing predicate discovery techniques. Invariant generation methods do find quantified invariants which may be useful in some cases. But the problem there is that a lot of invariants are generated and there is no good way of deciding which ones are useful.

In the presence of parameterized rules, the predicate discovery works exactly as described above. But the parameters (which are explicitly not part of the concrete state) in the rules may appear in the predicates finally generated. Recall that the predicates discovered characterize the set of states like x_2 (in Figure 2) that are part of a real abstract trace. Appearance of a rule parameter in these expressions implies that the parameter must satisfy some conditions in the concrete counterpart of the abstract trace. Any other value of the parameter which satisfies the same conditions could produce another concrete trace. Naturally, an existential quantifier wrapped around these expressions would find a predicate that is consistent with all possible behaviors of the (possibly unbounded) parameter.

```

state
  N: positive_integer
  status : array [N] of enum {GOOD, BAD}
  error : boolean

initialize
  status := All values are initialized to GOOD
  error := false /* No error initially */

rule(p : subrange [1..N])
  (status[p] = BAD) ⇒ error := true

property
  ¬error

```

Fig. 3. Quantified predicate example

Quantifier scope minimization is carried out so that smaller predicates may be found. In some cases the existential quantifiers can be eliminated all together. Often predicates of the form, $\exists x. Q(x) \wedge (x = a)$ where a is independent of x , are discovered. Heuristics were added so that this predicate would be simplified to $Q(a)$.

To illustrate the way quantified predicates are discovered automatically, a really trivial example is presented in Figure 3. In the example system we want to prove that *error* is always false. So the initial abstraction predicate chosen will be just the atomic formulas of the verification condition, in this case the predicate: $B_1 \equiv error$. With this abstraction the property can not be proved and an abstract counter-example trace, $\neg B_1, B_1$ is returned. Since the initialization rule is handled like any other rule (only with implicit guard *true*) the abstract counter-example that shall be analyzed is, *true*, $\neg B_1, B_1$. Using the test for spuriousness described earlier, the counter-example is shown to be a minimal spurious trace. Also the partial trace, $\neg B_1, B_1$ is real (that is a concrete counterpart exists) when $status[p_0] = BAD$ holds (p_0 is the specific value of the parameter chosen). However the initialization rule specifically sets all the elements of the *status* array to *GOOD*. Hence the predicate discovered will be, $status[p_0] = BAD$. But notice that the parameter appears in the predicate. Hence the new predicate will be, $B_2 \equiv \exists q. status[q] = BAD$.

Now the abstraction will be refined with the extra predicate. The additional bit will be initialized to *false*. Also the transition rule will now be enabled only when the new bit is *true*. Since that never happens the rule is never enabled and the desired property holds.

4 Application to AODV

As an application of this method we shall consider a simplified version of the Ad Hoc On-demand Distance Vector (AODV) routing protocol [15,16]. The simplification was to remove timeouts from the protocol since we could not find a way of reasoning about them in our system. The protocol is used for routing in a dynamic environment where networked nodes are entering and leaving the system. The main correctness condition of the protocol is to avoid the formation of routing loops. This is hard to accomplish and bugs have been found [6]. Finite instances of the protocol has been analyzed with model checkers and a version of the protocol has been proved correct using manual theorem proving techniques.

Briefly the protocol works as follows. When a node needs to find a route to another, it broadcasts a route request (RREQ) message to its neighbors. If any of them has a route to the destination it replies with a route reply (RREP) message. Otherwise it sends out a RREQ to its neighbors. This continues till the destination node is reached or some node has a route to the final destination. Then the RREP message is propagated back to the node requesting the route. When a node receives a RREQ message it adds a route to the original sender of the message, so that it can propagate the RREP back. Also nodes will replace longer paths by shorter ones to optimize communication.

The routing tables are modeled by the three two-dimensional arrays *route_p*, *route* and *hops*. Given nodes *i* and *j*, *route_p*[*i*][*j*] is *true* iff *i* has a route to *j*, *route*[*i*][*j*] is the node to which *i* forwards packets whose final destination is *j* and *hops*[*i*][*j*] is the number of hops that *i* believes are needed for a packet to reach *j*.

The message queue is modeled as an unbounded array of records. Each record has *type*, *src*, *dst*, *from*, *to* and *hops* fields. The *src* and *dst* fields are the original source and final destination of the current request (or reply). The *from* and *to* fields are the message source and destination of the current hop. The field *hops* is an integer which keeps track of the number of hops the message has traversed.

As explained before, for every route that a node has, it keeps track of the number of hops necessary to get to the destination. Consider three arbitrary but distinct nodes: *a*, *b* and *c*. The node *a* has a route to *c* and its next hop is *b*. In this situation the protocol maintains the invariant that *b* has a route to *c* and *a*'s hop count to *c* is strictly greater than *b*'s hop count to *c*. This makes sure that along a route to the destination the hop count always decreases. Thus there can not be a cycle in the routing table. This is the property that was verified automatically.

In the actual protocol, where links between nodes can go down, the age of the routes is tracked with a *sequence number* field. The ordering relation is more complex in that case. To simplify the system for the sake of discussion here the sequence numbers have been dropped. The simplified version is described in Figure 4 and 5.

The atomic predicates in the the verification condition are used as the initial set of predicates. The initial predicates are, $B_1 \equiv route_p[a][c]$, $B_2 \equiv route[a][c] = b$, $B_3 \equiv route_p[b][c]$ and $B_4 \equiv hops[a][c] > hops[b][c]$. The abstract

```

type
  cell_index_type : subrange(1..N)
  msg_index_type : subrange(1..infinity)
  msg_sort : enum of [INVALID, RREQ, RREP]
  msg_type : record of [type : msg_type;
                        from,to,src,dst : cell_index_type;
                        hops : integer];

state
  route_p : array [N][N] of boolean
  route : array [N][N] of cell
  queue : array [infinity] of msg_type
  a, b, c : msg_index_type

initialize
  msg_queue := all messages have type INVALID
  route_p := all array elements are false

/* Generate RREQ */
rule (msg : msg_index_type; src,dst : cell_index_type;)
  queue[msg].type = INVALID  $\wedge$   $\neg$  route_p[src][dst]  $\Rightarrow$ 
    queue[msg] := [# type = RREQ; src = src; dst = dst; from = src; hops = 0 #]

/* Receive RREP */
rule (in, out: msg_index_type;)
  queue[in].type = RREP  $\wedge$  queue[out].type = INVALID  $\Rightarrow$ 
    /* Add route to immediate neighbor */
    route_p[queue[in].to][queue[in].from] := true
    route[queue[in].to][queue[in].from] := queue[in].from
    hops[queue[in].to][queue[in].from] := 1

    /* Add route to RREP source if this is a better route */
    if hops[queue[in].to][queue[in].src] > queue[in].hops
       $\vee$   $\neg$  route_p[queue[in].to][queue[in].src] then
        route_p[queue[in].to][queue[in].src] := true
        route[queue[in].to][queue[in].src] := queue[in].from
        hops[queue[in].to][queue[in].src] := queue[in].hops + 1
      end

    /* Forward RREP */
    if queue[in].to  $\neq$  queue[in].dst  $\wedge$  route_p[queue[in].to][queue[in].dst] then
      queue[out] := [# type=RREP; src=queue[in].src; dst=queue[in].dst;
                    from=queue[in].to; to=route[queue[in].to][queue[in].dst]
                    hops=hops[queue[in].to][queue[in].src] #]
    end

```

Fig. 4. AODV protocol

```

/* Receive RREQ */
rule (in, out: msg_index_type;)
  queue[in].type = RREQ  $\wedge$  queue[out].type = INVALID  $\Rightarrow$ 
    /* Add route to immediate neighbor */
    route_p[queue[in].to][queue[in].from] := true
    route[queue[in].to][queue[in].from] := queue[in].from
    hops[queue[in].to][queue[in].from] := 1

    /* Add route to RREQ source if this is a better route */
    if hops[queue[in].to][queue[in].src] > queue[in].hops
       $\vee \neg$  route_p[queue[in].to][queue[in].src] then
        route_p[queue[in].to][queue[in].src] := true
        route[queue[in].to][queue[in].src] := queue[in].from
        hops[queue[in].to][queue[in].src] := queue[in].hops + 1
      end

    /* RREQ has reached final destination */
    if queue[in].dst = queue[in].to then
      queue[out] := [# type=RREP; src=queue[in].dst; dst=queue[in].src;
                    from=queue[in].to; to=queue[in].from; hops=0 #]
    /* The RREQ receiver has a route to final destination */
    elsif route_p[queue[in].to][queue[in].dst] then
      queue[out] := [# type=RREP; src=queue[in].dst; dst=queue[in].src
                    from=queue[in].to; to=queue[in].from;
                    hops=hops[queue[in].to][queue[in].dst] #]
    /* Forward RREQ */
    else
      queue[out] := [# type=RREQ; src=queue[in].src; dst=queue[in].dst
                    from=queue[in].from; hops=queue[in].hops+1 #]
    end

end

property
  (route_p[a][c]  $\wedge$  route[a][c] = b)  $\rightarrow$  (route_p[b][c]  $\wedge$  hops[a][c] > hops[b][c])

```

Fig. 5. AODV protocol (contd.)

system generates a counter-example of length one where a receives a $RREQ$ and adds a route to c through b while b does not have a route to c . The predicate discovery algorithm deduces that this cannot happen since in the initial state there are no $RREQ$ s present. So the predicate,

$$\exists x. \text{queue}[x].\text{type} = RREQ$$

is added and the new abstraction is model checked again. Now a two step counter-example is generated. In the first step an arbitrary cell generates an $RREQ$. In the next step a receives an $RREQ$ from b originally requested by c and sets its routing table entry for node c to b . Since b does not have a routing table entry to c this violates the desired invariant. Again the predicate discovery algorithm deduces that such a message cannot exist. So the predicate

$$\begin{aligned} \exists x. (& \text{queue}[x].\text{type} = RREQ \\ & \wedge \text{queue}[x].\text{from} = b \wedge \text{queue}[x].\text{src} = c \wedge \text{queue}[x].\text{to} = a) \end{aligned}$$

is discovered. Continuing in this manner in the next iteration the predicate,

$$\begin{aligned} \exists x. (& \text{queue}[x].\text{type} = RREQ \\ & \wedge \text{queue}[x].\text{from} = b \wedge \text{queue}[x].\text{src} = c \wedge \text{queue}[x].\text{to} = a \\ & \wedge \text{hops}[b][c] > \text{queue}[x].\text{hops}) \end{aligned}$$

is discovered. This is exactly the predicate that is required to prove the desired invariant. While verifying the actual protocol, similar predicates are discovered for the $RREP$ branch of the protocol as well. The predicates needed to prove the actual protocol are different from the predicates listed here but are of the same flavor. The program requires thirteen predicate discovery cycles to find all the necessary predicates.

References

1. David L. Dill Aaron Stump, Clark W. Barrett. CVC: a cooperating validity checker. In *Conference on Computer Aided Verification*, Lecture notes in Computer Science. Springer-Verlag, 2002.
2. R. Alur, A. Itai, R.P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation* 118(1), pages 142–157, 1995.
3. F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *5th International Conference on Computer-Aided Verification*, pages 29–40. Springer-Verlag, 1993.
4. Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3. ACM Press, 2002.
5. Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In *10th International Conference on Computer-Aided Verification*, pages 505–510. Springer-Verlag, 1998.

6. Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols, August 1999. Presented in the Recent Research Session at Sigcomm 1999.
7. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169. Springer-Verlag, 2000.
8. Michael A. Colón and Tomás 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.
9. Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 51–60. IEEE Computer Society, 2001. June 2001, Boston, USA.
10. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2002.
11. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna 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.
12. Yassine Lakhnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, pages 98–112, Genova, Italy, 2001. Springer-Verlag.
13. D. Lessens and Hassen Saïdi. Automatic verification of parameterized networks of processes by abstraction. *Electronic Notes of Theoretical Computer Science (ENTCS)*, 1997.
14. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.
15. Charles E. Perkins and Elizabeth M. Royer. Ad Hoc On-Demand Distance Vector (AODV) Routing. In *Workshop on Mobile Computing Systems and Applications*, pages 90–100. ACM Press, February 1999.
16. Charles E. Perkins, Elizabeth M. Royer, and Samir Das. Ad Hoc On-Demand Distance Vector (AODV) Routing. Available at <http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-05.txt>, 2000.
17. A. P. Sistla and S. M. German. Reasoning with many processes. In *Symp. on Logic in Computer Science, Ithaca*, pages 138–152. IEEE Computer Society, June 1987.
18. Rupak Majumdar Thomas A Henzinger, Ranjit Jhala and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*. ACM Press, 2002.
19. A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In Tiziana Margaria and Wang Yi, editors, *TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127, Genova, Italy, apr 2001. Springer-Verlag.