Explicit-Enumeration based Verification made Memory-efficient

Ratan Nalumasu
Department of Computer Science
University of Utah
Salt Lake City, UT 84112
e-mail: ratan@cs.utah.edu

Ganesh Gopalakrishnan*
Department of Computer Science
University of Utah
Salt Lake City, UT 84112
e-mail: ganesh@cs.utah.edu

Abstract—We investigate new techniques for reducing the memory requirements of an on-the-fly model checking tool that employs explicit enumeration. Two techniques are studied in depth: exploiting symmetries in the model, and exploiting sequential regions in the model. These techniques can result in a significant reduction in memory requirements, and often find progress violations at much lower stack depths. Both techniques have been implemented as part of the SPIN verifier, a widely used on-the-fly model-checking tool.

I. INTRODUCTION

With the growing complexity of hardware and software systems, there is growing awareness that they must be formally verified. Model-checking [3] is a popular approach for verification, especially for reactive systems. Model-checking can be done in many ways, two popular methods being implicit enumeration and explicit enumeration. Explicit enumeration proceeds by building the state graph of the model and checking the desired properties on the state graph.

Explicit enumeration forms the basis for a number of tools that have been used with great success in validating several realistic protocols [9, 7, 8]. Despite this, the amount of computer memory available often proves to be a bottleneck. In this paper, we address the problem of combating state explosion in on-the-fly explicit enumeration methods. We develop two solutions, namely state normalization and sequentiality exploitation, both of which have been incorporated into a version of the SPIN [8] verifier. Although our solutions may increase the run-time for verification, they are still quite practical, considering the fact that, in practice, space often proves to be more of a bottleneck than time.

*Supported by ARPA Order # B990, SPAWAR Contract # N00039-95-C-0018, and NSF award MIP-9321836

On Exploiting Symmetries

Many verification models contain multiple identical components. Considerable state-savings can be effected by capitalizing on such symmetries, and thereby avoiding the enumeration of identical states. There are many techniques available for exploiting symmetries. Three prominent categories of methods are (1) scalar sets [10], (2) homomorphic reductions [11], and (3) network invariants [12].

In the scalar-set method, a non-traditional data type (actually a non-traditional family of data types) called the scalar set is employed. A scalar set is a set with a finite and fixed number of elements. The elements of a scalar set, essentially, support only four operations: (1) equality testing, (2) inequality testing, (3) for-all, and (4) there-exists. As an example of usage of a scalar-set, consider an array A whose elements are treated identically by the protocol being verified. One would then index A using an index variable of type scalar-set. One would then only be able to test whether two index variables are the same or not, and either step through all the array locations using for-all or choose an arbitrary array index using there-exists. This information can be used by the verification tool to cut down the state space explored.

Another method for exploiting symmetries employed in tools such as COSPAN [7] is that of homomorphic reductions. In one instance of this approach, the system being verified is simplified by examining its sub-component(s), identifying those that are subject to state explosion, and replacing them by simplified sub-components that are equivalent with respect to the properties being verified.

The network invariant method is a family of methods concerned with proving properties about arbitrarily sized networks. In one approach of this type [12], a network P1 || P2 || ... of processes is represented by a more general description of the form P || Q where Q represents a network of an arbitrary number of P's. If a process such as Q (called the “network invariant”) can be found, the task of verification is greatly simplified. In another approach, a network of the form P1 || P2 || ... is replaced by an equivalent network of the form P^N.
existence of a network invariant has been widely studied [1, 12]. In yet another approach [2], given a finite-state model, a quotient model that takes the symmetries in the problem into account is found, and used as the basis for model checking.

A drawback of scalar-sets that we have identified is that there are some situations which call for more than the four operations supported by scalar-set data objects. In the second example used in this paper, that of the Rollback Chip, the so called written bits (WB) array is indexed by two counters (called CMF and OMF) that are incremented in a modulo fashion. It is the joint behavior of WB, CMF, and OMF that reveals symmetries. As an example, the state (CMF=0,OMF=0, WB="ones-only-at-position-0") happens to be the same as the state (CMF=1,OMF=1, WB="ones-only-at-position-1") because both these situations are observationally equivalent as far as the RBC operations are concerned. If CMF and OMF were implemented as variables of type scalar-set, they cannot be used to index WB and at the same time be subject to modulo-increment operations which are carried out on them in the RBC design. Such symmetries are closer in spirit to the notion of representation invariants [6]. In this paper, we present our technique called state normalization for exploiting symmetries at this level.

The homomorphic reduction approach is more general than the method we propose, but not as direct and simple to apply. Although network invariants methods are elegant and some of the results in this area are quite powerful [5], these methods have, hitherto, been demonstrated only for simple classes of behaviors. For systems of the size and complexity we are interested in tackling, it is not clear how difficult it will be to find suitable network invariants or quotient models.

State Normalization

In this paper, we propose a simple method for combating state explosion in the context of SPIN, called state normalization. In order to use state normalization, the designer must first identify the symmetries in the system (currently done manually). He/she then writes a normalizer function that maps each system state s to a corresponding normalized state s'. Whenever state s is generated during the execution of the SPIN verifier, it is immediately mapped into s' via the normalizer. From then on, the execution of the SPIN verifier continues with respect to s; s is not explored. Our results show that the method introduces only a low overall time overhead and effects a dramatic reduction in the number of states generated.

II. AN OVERVIEW OF SPIN

The operation of the verifier SPIN can be understood in terms of the algorithm known as supertrace, illustrated in Figure 1. This algorithm works as follows. First, supertrace generates an automaton representing the joint execution of all the components of the concurrent system being verified, using an asynchronous product operator. An example of such a product automaton is given in Figure 1. Next, supertrace elaborates the graph of this automaton depth-first. During the depth-first traversal, each new state generated is hashed into the index-space of a one dimensional bit-array called the “bit-bucket” hashtable, H. Suppose the current state is S, and has successors S1, . . . Supertrace computes the index k at which Sj falls, and if H[k] is already set, it is assumed that Sj has already been visited; the search then continues with Sj+1. On the other hand, if H[k] is not set, the depth-first elaboration is continued at Sj. If there are no hash collisions, supertrace will generate all reachable states. On the other hand, in the presence of hash collisions, supertrace prunes down the size of the reachable state-space. This feature of “randomized” pruning of the state-space is quite handy in handling very large verification problems, albeit with a loss of coverage. For small problem sizes, a regular hash table with linked-list buckets can be employed, which will then give full state coverage. SPIN supports this option also. One could view full-search as an extreme case of supertrace (amount of pruning equals zero).

SPIN supports four basic kinds of checks: local (state) assertions, deadlocks, progress loops, and accept cycles. State assertions establish safety properties. Any number of assertions can be placed in the user’s Promela code, and these will be checked when control reaches that
III. ILLUSTRATION OF STATE NORMALIZATION

As explained earlier, in our approach state normalization is carried out on states generated during the execution of SPIN. Whenever an un-normalized state is generated, it is necessary to normalize it and pursue it, rather than discard it rightaway, hoping that the normalized form of the same state will be eventually generated. This is explained with aid of an example in Figure 2 where \( I \) is the initial state, state \( N1 \) is the normalized form of the state \( U1 \), \( N2 \) is the normalized form of state \( U2 \), and \( N3 \) is the normalized form of the state \( U3 \). If the the un-normalized states are just discarded, then \( U1 \) will be discarded, and hence \( N2 \) will never be generated. \( N1 \), an equivalent state of \( U1 \), will be generated, and explored. However \( U2 \), the successor of \( N1 \), is also discarded because \( U2 \) is not a normal form. Thus the search never visits \( N3 \) or its equivalent form \( U3 \).

It is standard practice to require the designer to identify the symmetries in a system \([9, 7, 2]\). The global state of a symmetric system is, for example, a tuple of the states of the individual processing nodes plus the state of the medium (or “bus”). Any specific state that arises can always be normalized by taking processor 1 (for example) as the reference point. For example, in a truly symmetric system, the situation of processor 2 having sent a request to processor 3 and expecting a response from it can be rewritten into an equivalent situation with processor 1 playing the role of processor 2, and processor 2 playing the role of processor 3. We now proceed to present the details of the locking protocol and the state normalization function used.

A. Details of the Locking Protocol

Consider a system of \( N \) processors communicating by sending message through a medium. The processors coordinate among themselves to gain access to a shared resource protected by means of a lock. Every processor maintains “probable owner”; a variable pointing to the processor (possibly itself) which in its view is owning the lock (variable po of Figure 3). The lock is said to be owned by a processor if and only if the probable owner is itself. The lock itself can be in one of the two possible

![Fig. 2, A Caveat During Normalization](image)

![Fig. 3, State Machine Describing the Locking Protocol](image)
states at the owner: available or held. In Figure 3, states labeled C are part of the critical section implemented by the locking protocol while those labeled NC and W are outside this critical section. When processor p wants access to this critical section, it will execute the Acquire process which first checks whether the lock is currently owned by p (the check “po == me”). If it is, then the lock is set to “held” (locked ← 1), and the critical section is accessed. If p is not the current owner, a request for the lock is sent to the probable owner, and p then waits in state W for a granted message. When a request message is received by processor q, its Handle process is executed. This process checks to see if processor q is the owner of the lock. If it is not, then the message is forwarded to whom processor q thinks is the probable owner. Otherwise, if the lock is currently in the held state (locked == 1), then the request is enqueued into the queue maintained by processor q. On the other hand, if the lock is in the available state (locked == 0), a granted message is sent to the requester (p) along with the current queue which is empty (Q); it is an error to find locked == 0 and the current queue non-empty).

When the lock is released by the Acquire process, the queue is inspected to see if there are any enqueued requests for the lock. If there are none, the lock is set to the available state. If there are pending requests, a granted message is sent to the processor whose identity is at the head of the queue (h). This message also carries the rest of the queue (t). The probable owner is set to h and the locked status is cleared.

Each process in the state machine of Figure 3 is coded as one proc type in Promela. The communication medium is modeled as a collection of ports, one port per process. The ports are order-preserving, and their sizes are picked so as to make all send operations non-blocking. The queues (called “queue” in Figure 3) are modeled as chan data type. Since a transition of an acquire process can’t be taken simultaneously with a transition of a handle process on the same node, the two processes co-ordinate by using a semaphore called ‘mutex’.

B. An Assessment of State Normalization

Properties checked (with and without state normalization) include: (1) At most one process is in the critical section implemented by the protocol (i.e., in state C) at any given time; (2) The protocol is deadlock-free. We also tried to establish global progress, defined as the ability of at least one of the processors to be eventually in state C starting from any point in time. SPIN reported that global progress was not being met, and gave the error trace shown in Figure 4.

The following table summarizes the performance of state normalization (N) relative to un-normalized executions (U), for checking the above mentioned safety (S) and progress (P) properties. (Note: Recall that the safety property was met while the progress property was found to be not met). “Depth reached” was the stack depth bound set for each SPIN run. “nStates” was the total number of states visited, and “Time” was the elapsed time reported by the Unix command time, in seconds.

For Progress properties, both methods visited the same number of states before finding the error. The un-normalized execution time is always lesser than the normalized execution time which is to be expected in any method that tries to trade-off time for space. It was found that the process of normalizing a state (details given in Section V) itself consumed about 89% of the total execution time. It is clear that techniques for reducing the runtime of the normalizer can have a significant payoff. Also, normalization can help detect progress violations at much lower search depths, because with normalization, the depth-first search procedure used by SPIN does not stack equivalent states.

Soundness: Assuming that symmetries in the system have been correctly identified, establishing safety properties with respect to normalized states implies that the same properties also hold with respect to un-normalized states. Furthermore, finding a safety violation with respect to normalized states also implies that a similar vio-

<table>
<thead>
<tr>
<th>Prop.</th>
<th>Depth</th>
<th>nStates</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>U/N</td>
<td>S</td>
<td>100/100</td>
<td>37.46/3</td>
</tr>
<tr>
<td>U/N</td>
<td>S</td>
<td>300/300</td>
<td>29356/178</td>
</tr>
<tr>
<td>U/N</td>
<td>P</td>
<td>100/100</td>
<td>246/246</td>
</tr>
<tr>
<td>U/N</td>
<td>P</td>
<td>300/300</td>
<td>246/246</td>
</tr>
</tbody>
</table>

Fig. 4. Trace of an Error in the Locking Protocol
lation exists with respect to un-normalized states. These results directly follow from the fact that safety properties are dependent only on states, and not on the execution path(s) followed. Also, progress and accept violations are flagged correctly, assuming that fairness is not imposed.

We do not advance any claims as far as checking for progress or accept loops with fairness (either weak or strong) imposed. In our current work, we are working on modifying a version of the fairness algorithm described in [4].

IV. State Saving by Exploiting Sequentiality

The depth-first traversal method of SPIN is currently implemented in such a way that it saves newly generated states on the stack. However, in purely sequential regions of execution (in which only one thread is enabled), such state-savings are un-necessary.

To assess the benefits due to this optimization, we ran experiments on the RBC system which is highly sequential in nature. The sequentiality exploitation technique resulted in a sixfold reduction in memory requirements, when checking for safety properties. More specifically, the unoptimized version needed a depth of 350 just to visit every single statement of the protocol while the optimized version could achieve the same effect with a depth of only 55. A total of 18946 states were stored in the optimized version, while a total of 97947 states were stored in case of the unoptimized version. With a hash table of size $2^{18}$, the former produced only 641 collisions, while the later produced 60,973 collisions.

As described in Section I, the state-space of the RBC system can be reduced via state normalization also. Thus, a combination of these techniques can effect significant reductions in the state space of models that possess symmetries as well as contain a large number of sequential regions of execution.

V. Implementation of the Normalizer

We now describe details of the normalizer with respect to the locking protocol, which is symmetric with respect to the processor IDs. However, because of interdependencies between the processors through their “probable owner” variables, the normalizer is somewhat involved. These dependencies also extend through the message queues and other data structures.

A simplified version of the normalizer is shown in Figure 5. In this figure, LESS-TIMAN corresponds to a partial order on the processor indices chosen by the user with respect to which the normalizations are performed. Function normalize “sorts” the positions of the processors in the state vector to correspond to the partial order LESS-TIMAN. It achieves this as follows. It first exchanges the local variables of processors i and j. Next, it goes through every processor index k, and sees if processor[k]'s

```c
-- 'state' contains processor, queue, and medium state
function normalize (state)
    { for i := 1 to number_of_processes do
      for j := i+1 to number_of_processes do
        if LESS-TIMAN(i,j) then
          -- Exchange value 'i' with 'j'
          -- First exchange the local variables.
          local_variables(processor[i]) := local_variables(processor[j]);
          local_variables(processor[j]) := local_variables(processor[i]);
          -- Now adjust any dependencies through probable_owner
          -- or the queue.
          for k := 1 to number_of_processes do
            if (processor[k].probable_owner==i) then
              processor[k].probable_owner := j;
            elseif (processor[k].probable_owner==j) then
              processor[k].probable_owner := i;
            end if;
            foreach element (e in processor[k].queue) do
              if (e==i) then
                replace e with j
              elseif (e==j) then
                replace e with i
              endif;
            end for
          end for k;
          -- Now check the medium state.
          foreach message (m in medium) do
            if (DestinationOf(m)==i) then
              DestinationOf(m) := j;
            elseif (DestinationOf(m) == j) then
              DestinationOf(m) := i;
            endif;
          end foreach;
          foreach message (m in medium) do
            if (SourceOf(m)==i) then
              SourceOf(m) := j;
            elseif (SourceOf(m) == j) then
              SourceOf(m) := i;
            endif;
          end foreach;
        end if; -- if LESS-TIMAN
      end for j;
    end for i;
}
```

Fig. 5. The Normalizer
Fig. 6. High-Level Description of Symmetries

probable-owner field value i (j); if so, it changes it to j (i). Also, if i (j) is enqueued in processor[k]'s queue (queue of waiters for the lock), it is replaced with j (i). Finally, for each message m in the message medium, if the destination of the message is i (j), it sets the new destination value to j (i). A similar change is also effected on the message source field values.

In our current SPIN prototype implementing normalizations, the code in Figure 5 was manually written. As this process is error-prone, we are in the process of developing a compiler that can take a high level description of the symmetries and automatically generate the normalizer code. In this compiler, symmetries in the locking protocol would be described similar to what is shown in Figure 6.

VI. CONCLUSIONS AND FUTURE WORK

In conclusion, we present a symmetry exploitation procedure that helps combat the state-explosion problem in verifying concurrent systems through explicit enumeration based model-checking. Our technique is more general than scalar sets [10] or network invariants [12]. While not as general as homomorphic reductions [11], it is simpler, and straightforward to apply. We also present a technique to exploit sequential regions of a protocol to effect further space savings. A version of the verifier SPIN incorporating these techniques has been developed.

REFERENCES


