Explicit-enumeration based Verification made Memory-efficient

Ratan Nalumasu,
Ganesh Gopalakrishnan *,
Department of Computer Science,
University of Utah, Salt Lake City, UT 84112
{ratan,ganesh}@cs.utah.edu

February 24, 1995

Abstract

We investigate techniques for reducing the memory requirements of a model checking tool employing explicit enumeration. Two techniques are studied in depth: (1) exploiting symmetries in the model, and (2) exploiting sequential regions in the model. The first technique resulted in a significant reduction in memory requirements at the expense of an increase in run time. It is capable of finding progress violations at much lower stack depths. In addition, it is more general than two previously published methods to exploit symmetries, namely scalar sets and network invariants. The second technique comes with no time overheads and can effect significant memory usage reductions directly related to the amount of sequentiality in the model. Both techniques have been implemented as part of the SPIN verifier.

Keywords: Formal Methods, Verification, Model Checking

1 Introduction

With the growing complexity of hardware and software, the need to formally verify them is being increasingly felt. Among the options available today, two of the prominent ones are based on *deduction* and *model-checking* [8]. Although both methods have their proponents, model-checking [3] is preferred when a relatively high degree of automation is desired, and when one-of-a-kind reactive behaviors are involved. Model-checking can be carried out either via *implicit* enumeration where the state graph is implicitly traversed using (for example) BDD-methods or via *explicit* enumeration where the state graph is explicitly traversed and processed using graph algorithms. Both these approaches have their own strengths. Also, both methods suffer from *state explosion* [16], combating which forms a central research problem. This paper is about combating state explosion in explicit-enumeration-based verification.

Space/Time Tradeoffs During Explicit Enumeration

Explicit enumeration forms the basis for a number of tools that have been used with great success in validating several real-life protocols [12, 10, 11]. One problem with explicit enumeration is that the available amount of memory often decides the size of the problem that can be handled; most explicit-enumeration-based tools give 100% "coverage" till this limit is reached, and give 0% coverage once this limit is exceeded. Designers combat this abrupt loss of coverage in several ways; almost always, they use techniques such as throwing away irrelevant states, reducing the dimensions of the arrays involved, etc. [12]. Although this is essential in any verification

*Supported in part by NSF award MIP 9215878 and APRA Contract N-0003995-C0018

approach, there are cases where even after problem-size reductions the number of states exceed the available amount of memory.

Most tools in this area "prefer time over space"—*i.e.*, given a choice between running out of memory and giving slower responses, they prefer the latter. A justification for this attitude is that verification "jobs" can (and are) often run as background jobs, and designers are often willing to patiently wait for these jobs to come back with their answers, provided (according to their experience), their patience will (almost always) be rewarded. These ideas are key to our approach.

Promela/SPIN, Supertrace, and Two State-space Search

A simple and yet powerful method for effecting this "space/time tradeoff" is used in an explicit-enumeration-based tool called SPIN [11]. Strictly speaking, SPIN employs *two* techniques for effecting the space/time tradeoff: (1) *supertrace*, in which a "randomized" pruning of the state graph is effected; (2) *two state-space method*, in which only the amount of stack growth generated during a normal recursive depth-first search needs to be saved[1]. The experience of the SPIN user-community (including us) tells that these methods work well in practice, and can scale up to large problem sizes. In addition, descriptions of the system to be verified using SPIN can be provided in a high-level programming language called Promela. Promela is based on an *asynchronous* computational model which embodies powerful sequential and concurrent programming constructs that help the designer translate his/her thoughts about the protocols being verified into Promela with minimal semantic distance. For these reasons, the work reported in this paper is about enhancing the performance of the Promela/SPIN system.

Even with the use of supertrace and the two state-space method, SPIN suffers from state explosion, mainly due to the sheer complexity that real-life systems have. There are two main reasons for state explosion: interleaved concurrent execution, and the size of the data-state space. In this paper, we are mainly concerned with the latter. Our research in this direction was motivated by the fact that we are currently faced with the problem of verifying a large distributed memory multiprocessor in which multiple identical components exist at all levels. It is essential that we capitalize on the existence of these *symmetries* and avoid enumerating identical states repetitiously. The importance of exploiting symmetries is a widely studied problem, and is described in the next section.

Comparing Methods for Exploiting Symmetries

There are many techniques available for exploiting symmetries. Three prominent categories of methods are (1) scalar sets [13], (2) homomorphic reductions [15], and (3) network invariants [16].

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. In [13], it has been shown that scalar-sets are very useful in practice.

Another method for exploiting symmetries employed in tools such as COSPAN [10] 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.

[1] Plus a small overhead, actually

The network invariant method is a family of methods concerned with proving properties about arbitrarily sized networks. In one approach of this type [16], a network `P||...` 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 `P||...` is replaced by an equivalent network of the form $\mathtt{P}^N$. The existence of a network invariant has been widely studied [1, 16]. 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 described in Section 4, 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* captured in works such as [9]. 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 [4], 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.

Our Contribution: State Normalization

In this paper, our contribution is a simple method called *state normalization*. In this method, the designer identifies the symmetries in the system manually, and expresses them as *rewrite rules* on system states. Then, when the SPIN verifier runs, these rewrite rules are repeatedly invoked on each new state generated until a *normal form* system-state is obtained[2] The search continues with respect to normal form states, and all un-normalized states are discarded. Our results show that the method introduces only a low overall time overhead and effects a dramatic reduction in the number of states generated.

We have implemented state normalization as an extension to the SPIN verifier. We report experimental results obtained in the context of two non-trivial examples. The first example is concerned with *distributed locking* and was manually derived from an actual C/C++ implementation being developed by the systems group in our Department. Section 3 introduces state normalization with the help of this example.

The second example is concerned with verifying the *Rollback Chip* which was developed by our group several years ago [7] and is an IFIP WG 10.5 benchmark contributed by the second author. A functional/equational manual proof of correctness has been completed for the RBC [7]. Our present exercise of re-describing RBC in a reactive system description language is consistent with the manner in which system design refinement happens in formal design approaches: an initial functional description is gradually transformed into a more reactive version that embodies scheduling- and resource-related details. (See [6] for a case study of functional derivation followed by reactive process derivation.) The RBC example is detailed in Section 4.

[2]This rewriting process always terminates; depending on efficiency considerations, the designer may not always want to attain unique normal forms.

Because of the emphasis on implementation efficiency, the SPIN verifier compiles each Promela specification annotated with verification assertions into C-code and runs this C-code, rather than interpret Promela directly. In our current prototype, the state normalization procedures are coded in C and included with the above C-code. Writing normalization procedures in the C language is an error-prone activity; hence, we have come up with a scheme to automate this process, which is described in Section 5.

The rest of the paper is organized as follows. An overview of SPIN, the two state-space method, and supertrace is provided in Section 2. This is followed by a detailed look at state normalization via two examples (Sections 3 and 4). The normalizer is detailed in Section 5. Concluding remarks are provided in Section 6.

2 An Overview of SPIN

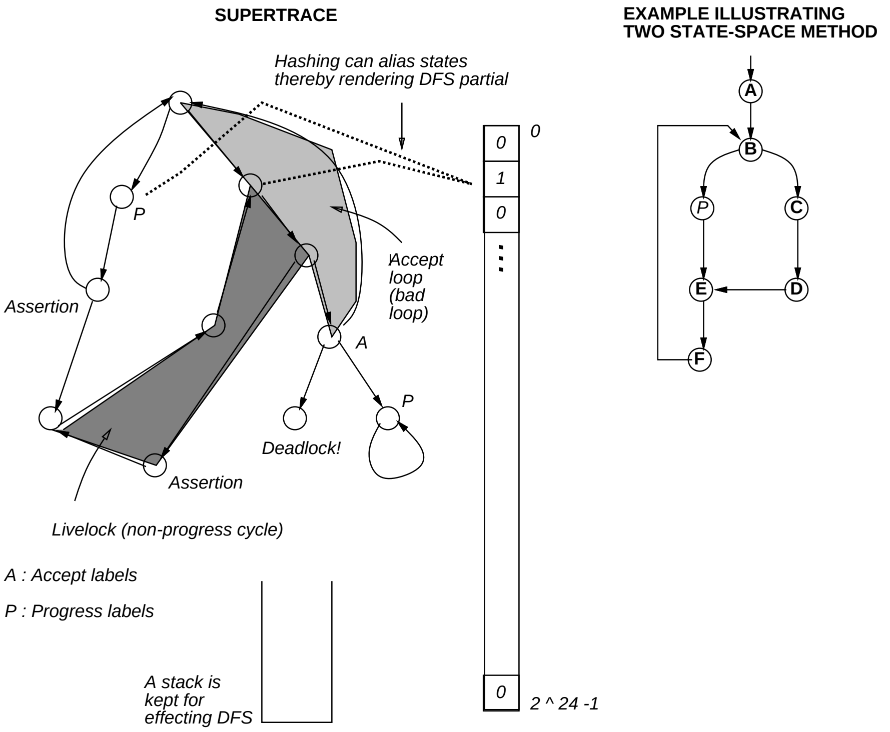

Figure 1: Overview of SPIN and Supertrace

Figure 1 provides an overview of supertrace. An automaton representing the joint execution of all the components of the concurrent system being verified is generated through the asynchronous product operator. An example of a product automaton is given in Figure 1. The graph of this automaton is elaborated depth-first. The size of the automaton graph is cut down via pruning which is achieved as follows. Each new state generated is hashed into the index-space of a one dimensional bit-array called the "bit-bucket" hash-table, H. Suppose the current state is S, and has successors $S_i \ldots$. Supertrace computes the index k at which S_i falls, and if $H[k]$ is already set, it is assumed that S_i has already been visited; the search then continues with S_{i+1}. On the other hand, if $H[k]$ is not set, the depth-first elaboration is continued at S_i. A "randomized" pruning of the state-space naturally occurs through hash collisions. For small problem sizes, a regular hash table with linked-list buckets can be employed, which will then give full 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 state. Deadlocks are automatically detected and reported by SPIN when a state without a successor is generated. Progress loops are loops in the state graph indicated by labels that begin with keyword `progress` (shown as "P" in the figure). For a system to be free from livelocks, its execution must be confined to one of its progress loops. Accept loops are opposite in sense: executions traversing an accept loop are considered "bad". They correspond to the acceptance condition of a *Büchi* automaton that captures an undesirable infinitary execution (for instance, *unfair selection*). Stack-growth during supertrace depends on the length of an execution path before a state is deemed to have been revisited by supertrace.

Checking for progress loops and accept cycles by building the entire state-graph is highly memory intensive. SPIN avoids this complexity by using the two state-space method, an example illustrating which appears to the right of Figure 1. Suppose we would like to detect and report the non-progress loop F, B, C, D, E, F (state "P" indicates the progress loop). A naive algorithm to detect non-progress loops during the depth-first search phase proves to be very inadequate [11]. SPIN uses modified depth-first search which works as follows on our example: when state B is revisited, it builds the subgraph rooted at the state immediately prior to the revisited state (state F in our example) in its entirety, in "the heap". In our example, let us say that depth-first search generated A, B, C, D, E, F, and B. At this point, the first path to be built in the heap is F, B, P. This path is abandoned because it includes the progress label "P". The next path built in the heap is F, B, C, D, E, F, and this path is reported as being a non-progress loop.

Thus, instead of building the entire state-graph, the two state-space method needs to, at a time, build only the amount of state contained in the depth-first stack plus a piece of the state graph rooted at the revisit-point. State normalization is aimed at reducing the size of the depth-first search stack and the number of entries (including collisions) made into the hash table.

3 State Normalization Illustrated on a Locking Protocol

The basic idea behind state normalization is extremely simple: (1) manually identify states that are equivalent, (2) select one of the states as the *normal form*, and (3) whenever a state is generated during depth-first search, normalize it if it is not already so. Note that if an

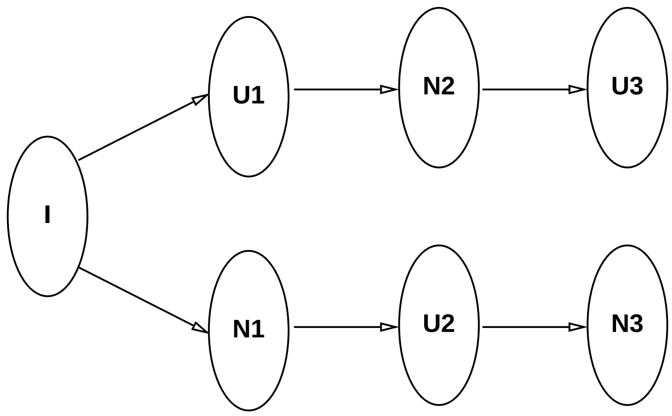

Figure 2: A Caveat During Normalization

unnormalized state is generated as part of the regular depth first search, it is not acceptable to just ignore that state and continue the search, 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. Hence, whenever an un-normalized state is generated, it is necessary to normalize it and pursue it, rather than discard it rightaway.

Equivalences among states are induced by the symmetries in the system being verified. It is standard practice to require the designer to identify the symmetries in a system [12, 10, 2]. Most of the symmetries in concurrent systems are self-evident (*e.g.*, Figure 3). This processor topology

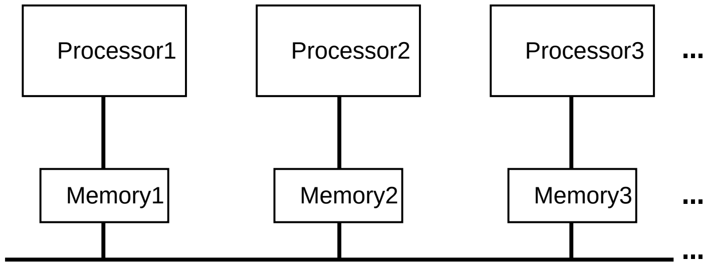

Figure 3: The Multiprocessor Supporting the Locking Protocol

is typical of many concurrent protocols. The global state of such a system is 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.

3.1 Details of the Locking Protocol

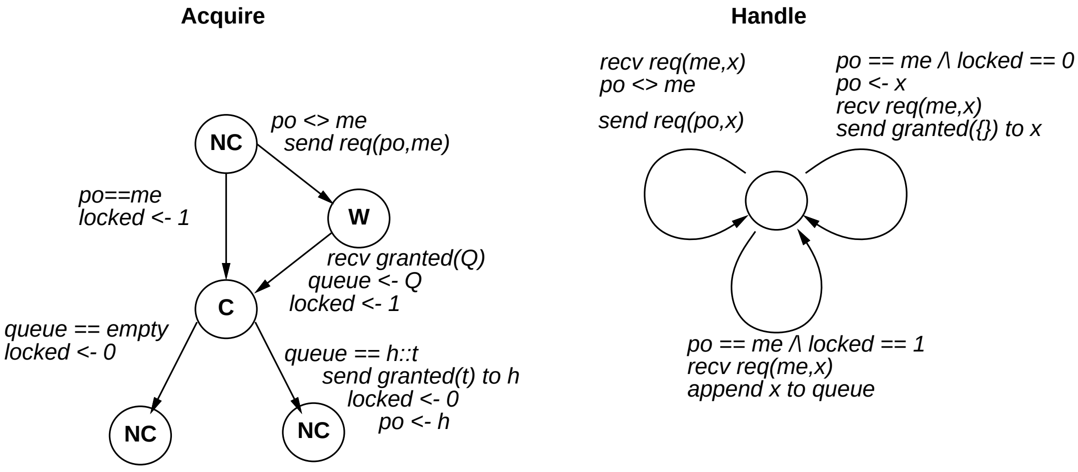

Figure 4: State Machine Describing the Locking Protocol

A system of N processors communicate 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 4). 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 states at the owner: *available* or *held*. In Figure 4, 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 $\leftarrow$ 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 to be 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 (`{}`; 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 4 is coded as one `proctype` 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 4) 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'. (Semaphore 'mutex' is different from the variable 'locked', since mutex is a regular semaphore on a *uniprocessor*, while variable 'locked' is distributed on multiple nodes. Also, to achieve the atomicity needed to implement the test-and-set of 'mutex', the `atomic` construct of Promela is used.)

3.2 Properties Checked and Results

The following properties were established:

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 5.

The following table summarizes the performance of state normalization (N) relative to unnormalized executions (U). "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.

	Properties	Depth reached	nStates	Time
U/N	Safety	100/100	37704/19240	1.5/4.0
U/N	Safety	300/300	293560/178546	11.7/35.7
U/N	Progress	100/100	246/246	0.06/0.09
U/N	Progress	300/300	246/246	0.07/0.12

Processor $P0$ is the current owner of the lock, and Processor $P1$ and Processor $P2$ point to $P0$ through their probable-owner variable. At this time, the medium, and all the queues are empty.

1. $P1$ sends a request to $P0$ for the lock.
2. Upon receipt of this request, $P0$ sends a granted message to $P1$. In addition, it sets its probable-owner variable to $P1$. (*At this point, there is a temporary cycle in the probable-owner chain between* $P0$ *and* $P1$*. Though this cycle is meant to vanish, it may not always, as we will see shortly.*)
3. Concurrently $P2$ also sends a request to $P0$.
4. $P0$ receives this request from $P2$ and forwards it to $P1$, given that $P0$'s probable-owner variable is set to $P1$.
5. $P1$'s Handle process acts on the request forwarded by $P0$ before $P1$'s Acquire process can act on the granted message sent in step 2. This causes the Acquire process to block.
6. The handle process of $P1$ continues to run, and forwards the request message from $P2$ to $P0$, since $P1$'s probable owner is still pointing to $P0$.
7. $P0$ forwards the message again to $P1$, which again interferes with the reception of granted message at $P1$ (just as it did in step 5). This process repeats. (*Had the granted message been serviced, the probable-owner cycle would have vanished.*)

Figure 5: Error Trace

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. Also, upon deeper examination, it was found that the process of normalizing a state (details given in Section 5) itself consumed about 89% of the total execution time. Techniques to reduce the time taken to normalize states need to be investigated.

It is worth noting that normalization can help detect progress violations at much lower search depths. This is due to the fact that with normalization, the depth-first search procedure used by SPIN does not stack equivalent states.

4 State Saving by Exploiting Sequentiality

We now illustrate our second state-saving technique on a different category which applies to systems that are fairly deterministic in nature, and are typically derived from a procedural/functional description. Examples of this category are data-intensive modules such as memory management units, various tabular data structures, and the like. In particular, we pick an example called the Rollback Chip for which we have, in our prior work, come up with a functional/equational specification and verified correctness using verification conditions generated from a computational induction scheme [7]. Our asynchronous synthesis group is currently actively engaged in trying to reimplement the RBC by detailing its operations to include more scheduling and resource sharing information. In a formal sense, this is a process of conducting design refinements [14] in a functional framework, and leading through a process/reactive framework. (An example of our past work in this area in deriving a pipelined multiplier is reported in [6].)

4.1 Overview of the RBC Specification

RBC [5] is a simple memory management unit designed to speed up the process of state saving and rollback in distributed discrete event based simulation using Time Warp. For the purpose of this paper, its functionality can be understood as follows. The RBC behaves like an abstract data type object with interface operations *reset* to initialize the RBC, *mark* and *rollback* to change the address mapping function, and *read* to map a given logical address to a physical address. All these operations have a purely functional description given in [7], where a proof of correctness (using equational reasoning) of the refinement of the RBC architecture has been reported. The Promela version of the RBC system was arrived at by modeling each RBC operation through a `proctype`. Invoking an operation is achieved by a message to the process associated with the operation, and waiting for a reply from the process.

Symmetries in this example cannot be exploited using scalarsets for reasons explained on Page 3. However, our normalization technique does work, as it is based on explicitly normalizing states. We do not elaborate upon state normalization in the context of the RBC example, as it has already been illustrated on the locking protocol. Instead, in this section, we look at another method to cut down memory requirements which is based on exploiting purely sequential regions of the RBC operations.

4.2 Results

Despite scaling the problem size down, the RBC model couldn't be completely verified due to the high number of reachable states. One problem identified was that the SPIN run time system was saving state after executing each statement of a process. However, this state saving is necessary only if there are multiple enabled threads in an execution. In case of the RBC, however, only one thread is enabled at any given time. (This was because the Promela version was a direct translation of the functional description given in [7]. Successive refinements of this Promela version will have much more concurrency; however, in these versions also there would be occasional sequential regions.) This fact can be exploited by not saving states in-between the individual steps contained in a sequential region. This resulted in a sixfold reduction in memory requirements. 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 18646 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.

5 Implementation of the Normalizer

We now describe details of the normalizer with respect to the locking protocol. This protocol 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 6. In this figure, LESS-THAN corresponds to an arbitrary partial order chosen by the user with respect to which the normalizations are performed. Function normalize "sorts" the positions of the processors in the state vector according to the partial order LESS-THAN. Whenever the partial order is violated, the normalizer exchanges the processors involved. It first exchanges the local variables, and then proceeds to examine the dependencies introduced by the probable owner variable and adjusts them accordingly. Then dependencies through the messages in the medium are traced, and normalized suitably.

In our current SPIN prototype implementing normalizations, the code in Figure 6 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. For instance, the locking protocol would be described as shown in Figure 7.

In Figure 7, queues are represented as a list of processor-ids and the medium by "from:`<ID>`, to:`<ID>`, message`<BODY>`". Notation [P/{e1 `->` e1', e2 `->`e2', ...}] means every occurrence of e1 is replaced by e1' and every occurrence of e2 is replaced by e2' in P (all replacements are done concurrently). If e1 = e2, then e1' and e2' must end up being equal.

Given such a rewrite rule, the system creates a partial order function compare(i,j) which returns '`<`', '=', or '`>`'. If multiple rewrite rules are present, one compare function is generated

```
function normalize (state)
{ for i := 1 to number_of_processes do
  for j := i+1 to number_of_processes do
    if LESS-THAN(processor[i], processor[j]) then
      -- Exchange value 'i' with 'j'
      -- First exchange the local variables.
      temp := local_variables(processor[i]);
      local_variables(processor[i]) := local_variables(processor[j]);
      local_variables(processor[j]) := temp;

      -- Now adjust any dependencies throuh probable_owner
      -- or the queue.
      for k := 1 to number_of_processors do
        if (processor[k].probable_owner==i) then
            processor[k].probable_owner := j;
        elsif (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
            elsif (e = j) then
                replace e with i
            endif;
        end do e;
      end for k;

      -- Now check the medium state.
      foreach message (m in medium) do
        if (DesinationOf(m)== i) then
            DesinationOf(m) := j;
        elsif (DesinationOf(m) == j) then
            DesinationOf(m) := i;
        endif;
      end do m;
      foreach message (m in medium) do
        if (SourceOf(m)== i) then
            SourceOf(m) := j;
        elsif (SourceOf(m) == j) then
            SourceOf(m) := i;
        endif;
      end do m;
    end if; -- if LESS-THAN
 end for j;
 end for i;
}
```

Figure 6: The Normalizer

```
{ (forall X
    (Processor[X].po = P), (processor[X].Queue = Q))
  (medium = M)
}
EQUALS  -- Exchange processor i with processor j
        -- for some arbitrary i, and j.
{
  (Processor[X].po = [P/{i->j, j->i}]),
  (Processor[X].Queue=[Q/{i->j,j->i}]),
  (medium = M/{ from:i->from:j, from:j->from:i,
                to:i->  to:j,   to:j->  to:i})
}
```

Figure 7: High-Level Description of Symmetries

per rule. Compare(i,j) applies the rewrite rule, and then checks to see if the new vector is less than, equal, or greater than the original vector. If Compare(i,j) returns '>' then the original vector is considered to be in normal form. If Compare(i,j) returns '=', the rewrite rule has no effect on the current state. If Compare(i,j) returns '<' then the rewrite rule is applied, and the new vector is considered to be in normal form.

This normalization is not confluent when more than one rewrite rule is present, in that two state vectors which are equivalent under the rewrite rules might not be reduced to the same normal form. However, when only one rewrite rule is present, the process is confluent. In general, finding the normal form for a state is known to be an NP-complete problem [2].

6 Conclusions and Future Work

The results of Section 3 demonstrate that exploiting the symmetries can result in a significant improvement in the usage of available memory. It is therefore very important to exploit such symmetries to be able to verify large concurrent systems. This technique is more general than scalar sets [13] or network invariants [16]. While not as general as homomorphic reductions[15], it is simpler, and straightforward to apply. Also, Section 4 demonstrates that identifying the sequential regions of a protocol can result in significant savings in memory.

In Section 5 we presented a technique to translate high-level rewrite rules into a low-level normalization routine. This technique needs to be further investigated and implemented. It would also be useful to provide an automatic procedure to (1) identify the symmetries in a system and (2) check that the rewrite rules are consistent.

References

[1] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. On the automatic computation of network invariants. In *Computer-Aided Verification*, pages 234–246, Stanford, CA, June 1994.

[2] E. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Computer Aided Verifcation*, pages 450–163, Elounda, Greece, June 1993.

[3] Edmund Clarke, Allen Emerson, and Arvind Sistla. Automatic verification of finite-state concurent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[4] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *Proc. of the 21st Annual Symposium on the Principles of Prog. Langs.* ACM, 1994.

[5] Richard M. Fujimoto, J. -J. Tsai, and Ganesh Gopalakrishnan. Design and evaluation of the rollback chip: Special purpose hardware for time warp. *IEEE Transactions on Computers*, 41(1):68–82, January 1992.

[6] Ganesh Gopalakrishnan and Venkatesh Akella. A transormational approach to asynchronous high-level synthesis. In *VLSI'93*, number 2, September 1993. Grenoble, France.

[7] Ganesh C. Gopalakrishnan and Richard Fujimoto. Design and verification of the rollback chip using HOP: A case study of formal methods applied to hardware design. *ACM Transaction on Computer Systems*, 11(2):109–145, May 1993.

[8] Aarti Gupta. Formal methods: A survey. *Formal Methods*, 1994.

[9] John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, December 1978.

[10] Z. Har'El and R.P. Kurshan. Software for analysis of coordination. In *Proc. Int'l Conference on System Science*, 1988.

[11] Gerard Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[12] Alan Hu, David Dill, Andreas Drexler, and Han Yang. Higher-level specification and verification with BDDs. In *Computer Aided Verification*, pages 82–96, Montreal, Canada, June 1992.

[13] C. Norris Ip and David L. Dill. Better verification through symmetry. In *Int'l Conference on Computer Hardware Description Language*, 1993.

[14] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. The MIT Press, 1984. An ACM Distinguished Dissertation-1983.

[15] Robert P. Kurshan. Formal verification of coordinating processes. Mathematics Research Center, AT&T Bell Labs Murray Hill, NJ, 1994.

[16] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.