Coverage Preserving Reduction Strategies for Reachability Analysis

Gerard J. Holzmann[a], *Patrice Godefroid*[b] *and Didier Pirottin*[b]

[a] AT&T Bell Laboratories, Murray Hill, New Jersey 07974, USA

[b] Université de Liège, Institut Montefiore B28, 4000 Liège Sart-Tilman, Belgium

Abstract

We study the effect of three new reduction strategies for conventional reachability analysis, as used in automated protocol validation algorithms. The first two strategies are implementations of *partial order semantics rules* that attempt to minimize the number of execution sequences that need to be explored for a full state space exploration. The third strategy is the implementation of a *state compression* scheme that attempts to minimize the amount of memory that is used to built a state space. The three strategies are shown to have a potential for substantially improving the performance of a conventional search. The paper discusses the optimal choices for reducing either run time or memory requirements by four to six times. The strategies can readily be combined with each other and with alternative state space reduction techniques such as supertrace or state space caching methods.

Keyword Codes: D.1.3; D.2.4
Keywords: Concurrent Programming, Program Verification

1. INTRODUCTION

It is sometimes easier to think of new reduction strategies for state space exploration algorithms than it is to prove that they are useful. Reductions can be sought in the run time or in the memory requirements of large state space searches. Usually, there is a delicate trade-off to be made. A reduction of the memory requirements typically increases the run time requirements, and vice versa. In this paper it is assumed that the reader is largely familiar with the setup of a standard reachability analysis algorithm. In a nutshell, these algorithms can establish the correctness of systems of interacting concurrent processes by an exhaustive generation and inspection of all composite states that are reachable from a pre-defined initial state. If necessary, more details on a range of different reachability analysis algorithms can be found in [1].

The improvement achieved by a reduction strategy must always outweigh the overhead incurred by its implementation, where both cost and gains are expressed as run time and memory requirements on a single reference machine. We discuss three new reduction strategies that can be shown to pass this test. Section 2 gives an overview of the classic search strategy. Sections 3 and 4 discuss the foundation for a partial order semantics reduction strategy, and describe the performance of two sample implementations. Section 5 defines some further refinements of the method. In Section 6 we examine the implementation of a remarkably simple state compression technique and its potential impact. Figure 3 in that section gives an overview of the net effect of all algorithms considered, including the effect of various combinations.

Background

In a traditional state space exploration, reachable states are generated over all execution paths in a concurrent system. To generate all these execution paths, the search 'shuffles,' or interleaves, the actions of all asynchronously executing processes in every feasible way.

When the actions affect the state of shared objects, for instance by sending or receiving messages through shared queues, the result of each new interleaving can be different, and may or may not lead to errors. When two actions are independent, however, the classic shuffling can be redundant. As an extreme example, consider two non-interacting, completely independent processes. A classic reachability analysis of such a system will explore all possible interleavings of the independent actions, all leading to the same result.

This may seem like overkill, but the problems are subtle. It is fairly hard for an automated validation system to detect accurately where processes are independent and where interleavings can safely be suppressed. Early methods based on heuristics always carried the risk of incompleteness of the validation results: they could not guarantee a complete coverage of the state space.

In 1990, new interest in the study of partial order semantics was triggered by the publication of two papers in [3], reporting on (independent) work of Patrice Godefroid and Antti Valmari. The papers argued that, in certain cases, the number of interleavings that must be inspected can be reduced *provably* without loss of coverage. Extensions appeared in [5,6].

In this paper, the method described by Godefroid is generalized and integrated with an efficient automated protocol validation system called SPIN [7]. [The original version of SPIN can be obtained by anonymous ftp from research.att.com from the /netlib/spin directory.]

The original version of SPIN includes an implementation of a classic search, which we will use here. It is illustrated with Algorithm 1 in the next section.

2. CLASSIC SEARCH

The details of PROMELA, the validation language that SPIN accepts, can be found in [7]. For the following discussion a brief overview of the main features will suffice.

PROMELA defines systems of asynchronously executing concurrent processes that can interact via shared global data objects. Below, we call a definition of such a system a *model*. The correctness criteria for a model can be defined by the addition of formal assertions, special labels that signify progress –, acceptance – and termination conditions. By default, a SPIN validation checks for absence of deadlock and the observance of a range of standard completeness criteria, such as unspecified receptions, and unexecutable code segments. The validator can also detect non-progress cycles, acceptance cycles, the validity of system invariants, and the feasibility of invalid system behaviors specified as PROMELA **never** claims. The never claims have the expressive power of linear time temporal logic formulae.

Processes and message channels in PROMELA can be created dynamically, thus producing systems that can grow or shrink during execution and validation. The validator SPIN implements several different search strategies. The most important two are a standard exhaustive state space exploration and a simplified version of supertrace bit-state hashing [7,8].

There are only three distinct types of objects in PROMELA: processes, variables and message channels. Variables can be declared either global or local to a specific process. A process executes the behavior that is specified in a so-called **proctype** definition (much like a procedure definition). More than one process can execute the same **proctype** definition concurrently. Of course, the local variables of such processes are distinct.

Interaction via a message channel can be either synchronous (i.e., by rendez-vous) or asynchronous (buffered), depending on what type of channel is declared. The two types of communication can be combined.

Processes, variables, and message channels are always of a finite size in PROMELA. Also the maximum number of processes and channels that can be created is bounded to a fixed upper limit. There is a single, user defined, initial system state for each model. This means that the behavior of any system of interacting processes defined in PROMELA can be explored completely by an exhaustive enumeration of all system states that are reachable from the initial system state. All correctness properties are decidable for such a model, with standard state space exploration algorithms [1]. The classic search algorithm has P-SPACE complexity [2].

The composite behavior defined by a PROMELA model corresponds to, and could be defined as, a pure finite state machine. As noted, this global machine has a finite number of states that can be enumerated exhaustively. Every state in the global machine represents a system state of the model: a composite of all individual process states, all local and global variable values, and all message channel contents.

The behavior of the global machine can be represented by a finite graph, where every node represents a reachable system state and every transition represents the execution of a single statement in a single process. In effect, a state space exploration algorithm traverses this target global system graph in all possible ways, starting from the root (the global initial system state). In SPIN the traversal of the graph is organized as a depth-first search. Of course, the graph is not necessarily tree-shaped. Any state can be connected to any other state and thus arbitrarily many cycles can exist. Typically there exists at least one path from any state to any other state via the initial state, that is, the graph is often strongly connected.

2.1. Algorithm 1

The following simple algorithm performs a depth-first search of the nodes in such an arbitrarily connected graph that is rooted in the initial system state. For brevity, we will call the node that corresponds to the initial system state the *root* node of the graph.

ALGORITHM 1 — CLASSIC SEARCH

1. **Initialization**: Label every node in the graph with a unique ordinal number `n`, and assign it a boolean flag with initial value *true* (meaning that it remains to be visited). Define an ordered set of nodes `S`. Enter the root node of the graph into set `S`, and set its flag to *false*.
2. **Iteration**: Select the *last* element of `S`, call it `s`. From the set of immediate successors of `s` that have a flag equal to *true*, select the one with the lowest ordinal number, set its flag to *false*, and add it to `S`. If no such node exists, remove node `s` from set `S`.
3. **Termination**: Repeat step 2 until set `S` is empty.

The classic search has the following well known property for finite state systems, which is readily proven with standard graph theory.

PROPERTY — Algorithm 1 always terminates within a finite number of steps. When it terminates, every node in the graph has been visited. □

The execution time and the memory requirements of Algorithm 1 are both linear in the number of nodes N, and hence linear in the total number of reachable states of a model.

Traditionally, set `S` is called a ‘search stack,’ and the data base of all nodes in the graph is called the ‘state space.’ It is also not necessary to first construct the graph before the nodes can be enumerated, as the version of Algorithm 1 given above suggests. Enumeration, graph construction, and analysis can all happen simultaneously, on-the-fly. It is not even necessary to store the complete set of nodes, or the complete search stack to implement Algorithm 1. Optimized algorithms that avoid these problem have been used for at least a decade, and were described in detail elsewhere [e.g. 7,9]. We discuss a simple on-the-fly variant of Algorithm 1 later in this paper as Algorithm 4. For our current purposes, however, the version shown in Algorithm 1 will suffice.

The main flaw of Algorithm 1 is that it ignores where transitions are independent. It will therefore traverse more transitions than strictly necessary. In the next two sections we consider ways to avoid this problem using the implementation of partial order rules. The implementation was divided into two separate steps, leading to Algorithms 2 and 3. We study the simplest variant first.

3. REDUCED SEARCH

We can attempt to reduce the cost of the graph traversal by reducing the number of edges that is traversed in the state graph, wherever we can do so without loss of information. Consider the example of two concurrent processes executing mutually independent statements, such as assignments to local variables.

```
process X: x1; x2
process Y: y1; y2
```

The graph of the system state space that can be constructed for this fragment of the model has 9 nodes and 12 edges. In Figure 1a the nodes are numbered in the order in which they would be visited by Algorithm 1. To the left of each edge is given the order of traversal for each edge plus, in braces, the statement that corresponds to that edge.

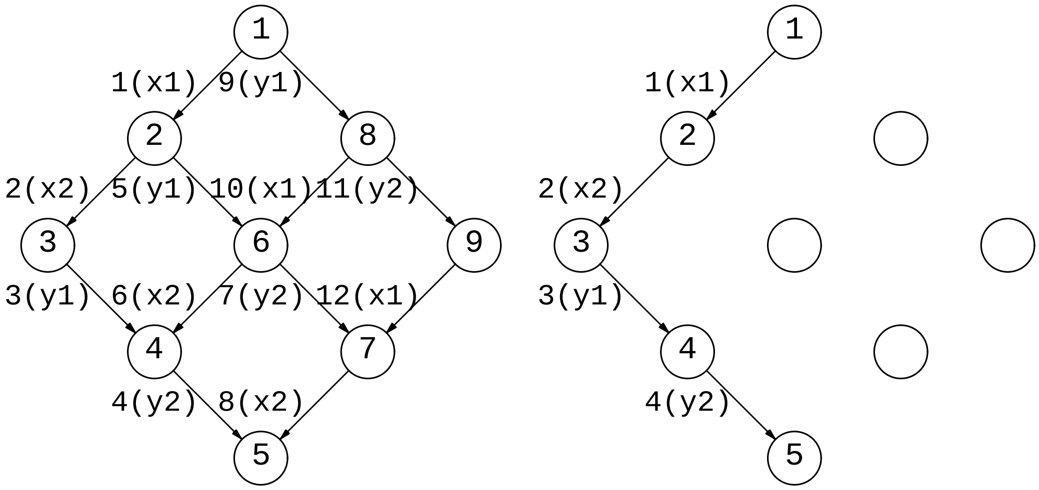

Figure 1 — Complete (a) and Reduced (b) State Graph

Since the statements in two processes are mutually independent, the outcome of all traversals of the graph is the same. They all lead from the root node 1 to the leaf node 5. The values of the local variables at leaf node 5 do not depend in any way on the path that is taken. Of course, no matter how we change Algorithm 1, we must still at least be able to establish the following two correctness properties of a model:

A. Each process performs a valid computation, without violating any local assertions.

B. All processes together proceed from a known composite initial state to a known composite final state (e.g., node 5 in Figure 1).

The method described in [5] was essentially devoted to the verification of deadlock freedom. It preserved properties of type B, but not of type A. The generalized method we describe here, however, preserves both types of properties.

We assume here that local properties of any one process can only be verified within that process and that they are invisible outside it (using standard scope rules that limit the visibility of all non-global objects). It is therefore forbidden for a process to assert anything about objects that are invisible to it, such as the value of variables that are local to another process, or the current control flow point of another process. It is also not allowed to state anything about intermediate composite process states that may or may not be reachable in the absence of any synchronization or interaction between the independent processes. Though this does restrict the correctness proving capabilities of the original SPIN, in ruling out the usage of so-called 'remote referencing of local variables,' most of these restrictions are easily circumvented in any application. Note that local variables can always be lifted to a scope where they are visible to a remote process, without changing the functionality of the model.

To establish properties A and B for the sample graph from Figure 1a, it would suffice to select any single one of the 6 edge sequences that lead from node 1 to node 5:

`1-2-3-4,  1-5-6-4,  1-5-7-8,  9-10-6-4,  9-10-7-8,  9-11-12-8`

If we select the first one of these, the graph traversal of Figure 1b results, and the cost of traversal is reduced to the inspection of only 5 instead of 9 nodes, while traversing just 4 instead of 12 edges.

The question is now, how do we modify Algorithm 1 so that we can safely decide during the normal depth first traversal sequence, that, for instance, edges 5 and 7 need not be explored? Let us first consider only an optimization for purely local transitions and defer the harder problem of determining when the access to global objects can create dependencies.

3.1. Algorithm 2

For brevity, we will call any edge in the state graph that corresponds to a transition referring only to objects local to the executing process, a 'local edge.' Any other type of edge is called a 'global edge.' We can now revise the search algorithm as follows. The new algorithm achieves its reduction by blocking selected edges in the graph with the help of boolean *eligibility* flags.

ALGORITHM 2 — REDUCED SEARCH

1. **Initialization**: as in Algorithm 1. Also assign an *eligibility* flag to each edge in the graph, with an initial value of *true* (meaning that the edge still can be traversed). Group the edges exiting from each node into separate edge sets, such that only edges that correspond to transitions from the same process in the model belong to the same edge set. Call an edge with a *true* eligibility flag, a *true* edge. Call a node with a *true* node flag, a *true* node.
2. **Iteration**: Select the *last* element of `S`, call it `s`. Find an edge set for node `s` that contains only local edges, and with at least one *true* edge leading to a successor node not contained in `S`. If there is more than one such edge, select the one that leads to the successor node with the lowest ordinal number. Mark the edges in *all other* edge sets of node `s` *false*. Mark the edge selected and its target node *false*, and add the node to set `S`. If no such edge exists, select a *true edge* that leads to a *true* successor node of `s`, taken from any edge set. If there is more than one such edge, select the one that leads to the successor node with the lowest ordinal number. Mark the edge selected and its target node *false*, and add the node to set `S`. If no such edge exists either, remove node `s` from set `S`.
3. **Termination**: as in Algorithm 1.

In Algorithm 1 we did not need the eligibility flags on edges. The eligibility flag of each edge could have been defined to be equal to the node flag of its target, since an edge only becomes ineligible in Algorithm 1 if it leads to a node that has previously been visited. The reduction in Algorithm 2 is achieved by marking edges ineligible for selection in also other cases. The following proviso, however, must be met explicitly before an edge can be marked ineligible while its target node is unvisited.

There exists a local edge exiting from the current node leading to a node outside set `S`.

The proviso says that it is not sufficient that merely a *true* local edge exists within a local edge set; it must also lead to a target state that is not in the depth-first search stack. The need for a proviso was also recognized by Valmari which contains a different solution [13].

THEOREM — The second half of the proviso is both necessary and sufficient to guarantee that properties of type A and B can be verified.

Proof of necessity — Consider a modification of the sample `process1` from above in which three statements `x1`, `x2`, and `x3` are executed infinitely often in that order, in a cycle. Let these three statements further consist of simply the assignment of some constant value to a local variable within the process. Let `process2` be defined as before. The complete state graph for this example would be as shown in Figure 1a, but with the addition of three edges: from nodes 3 to 1, from 4 to 8, and from 5 to 9, each new edge corresponding to an execution of the extra statement `x3` in `process1`.

Without the second half of the proviso, the reduced graph would be as shown in Figure 2a (left), with it the graph is as shown in Figure 2b (right). Edges in Figure 2 that do not connect to their target node represent eligible edges that lead back to previously visited nodes. In Figure 2a, the edges corresponding to the statements of `process2` are missing. This means that not all correctness properties of type A can be verified. Since there is at least one case where the omission of the second half of the proviso leads to an incorrect result, it is proven to be necessary. □

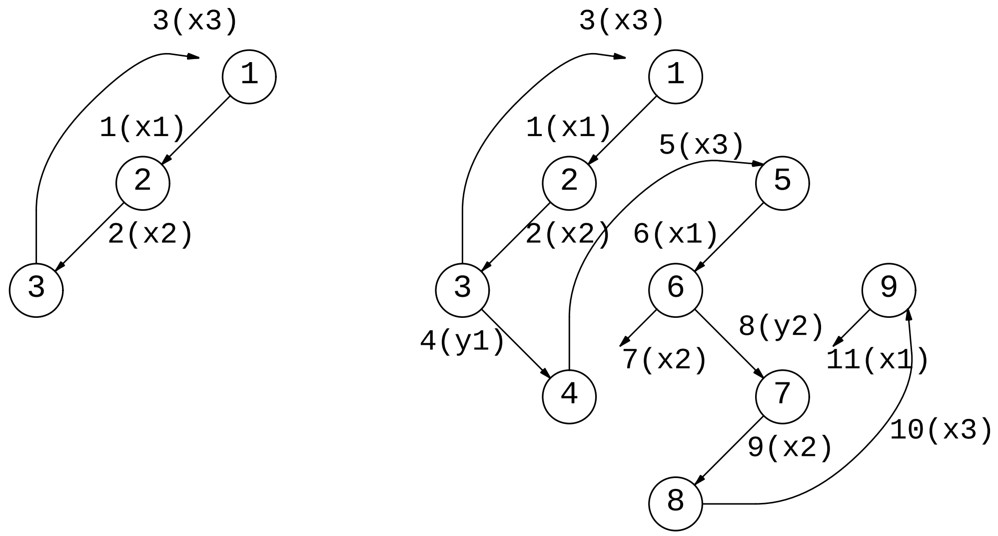

Figure 2 — Reduction with (right) and without (left) proviso

Proof of sufficiency (informal) — To prove sufficiency we show that, when both halves of the proviso are enforced, all statements of all processes that are visited in the complete graph traversal are also visited during the reduced graph traversal. We first prove the following Lemma.

Lemma — When state `s` is removed from set `S` by Algorithm 2, all transitions enabled in `s` have been executed, either at state `s` itself, or at successor states of `s`.

Proof of Lemma — The proof is by induction. Consider the first state that is backtracked during the search, call it s_0. It is the last state in the first path of the spanning tree explored by the depth-first search.

1. Either all enabled transitions are executed at s_0. Note that there may be zero such transitions.

2. Or the transitions of only one edge set are executed, and all others are labeled ineligible.

In the first case, the Lemma trivially holds for s_0. In the second case, at least one of the true edges in the edge set selected must lead to a state not currently in `S`. Such a successor state, however, would be backtracked before s_0, which contradicts the assumption that s_0 is the first such state, and proves the Lemma for s_0. Next, we prove that if the Lemma holds for the first `n` states that are backtracked, it must also hold for the `n+1`th state, s_n. There are again two cases:

1. Either all enabled transitions of s_n are executed and the Lemma is true.

2. Or all transitions of only one edge set `T` are executed.

Call `T'` the set of all other transitions enabled at state s_n. At least one true edge in `T` leads to a state `s'` not in set `S`. The set of transitions enabled in `s'` must include set `T'`. (Since all transitions in `T` are independent from the ones of `T'`, they remain enabled.)

State s_n is not backtracked until all its successors, including `s'`, have been backtracked first. From the inductive hypothesis, we know that all transitions in `T'` then must have been executed during the search below `s'`. Therefore also all enabled transitions in `s` were executed during the search below `s`, which proves the Lemma. The proof of the **Theorem** follows by the application of the Lemma to the root node of the graph and recursively to all its successors. □

The implementation of Algorithm 2 is straightforward and can be done without adding significant overhead to the search. Most of the work can be done at compile time, when a protocol specific analyzer is generated from a model specification, and need not increase the run time or the memory requirements of the search itself. The effect of the reduction achieved by Algorithm 2 can therefore be substantial.

3.2. Measurements

The results of each reduction was measured on five sample protocols. Three of these protocols are fairly standard, though artificial, tests. The two remaining tests are randomly chosen realistic protocols.

1 N independent processes, each traversing M local states, without cycles.
2 N independent processes, each cycling through M local states.
3 N completely dependent processes, each traversing M states, without cycles.
4 AT&T's Universal Receiver Protocol (URP), modeled in 419 lines of PROMELA.
5 A sample data transfer protocol (DTP), modeled in 391 lines of PROMELA.

The results of all measurements performed in this study are collected in Table II, included in Section 6. For this first experiment, only the numbers related to Algorithms 1 and 2 are relevant (the rows labeled 1 and 2). Table II lists the numbers of states visited and transitions traversed for each type of search. All measurements were run on a single 33 MHz SGI Indigo using a MIPS R3000 processor, comparing the original version of SPIN [7] in classic search mode against a version in which the classic search was modified in accordance with Algorithm 2. For each test, both versions of the search were run with the same stack limit and hash-table size. The only thing changed was the graph traversal discipline itself. Time is user time plus system time as reported by the UNIX® system time command. The traversal of edges leading back to previously visited states are included in the transitions count.

To correctly interpret these results, a few words of caution are in order. First, to keep the comparisons fair, the memory usage for each test run reported here includes the overhead of the (hashed) lookup tables that are required to build the state spaces. The table is kept at a constant size of 262,144 slots (or roughly 1 Mbyte) throughout all tests. So even when a very small number of states is stored, the overhead of the lookup table is still present.

One extra state and transition is also counted in each test for the initial system state, just before all processes are instantiated (giving, for instance, a count of $10^5 + 1$ instead of 10^5 states in the first series of tests). In the first test, the reduced algorithm successfully avoids the exploration of all redundant transitions and states, which leads to a dramatic improvement in both run time and memory usage. The count for Algorithm 2 is less than the 51 states that might be expected because the reduced algorithm can overlap the executions at the last state of each process. Included in the count of 47 are 1 initial state, plus 10 states for the first process explored, plus 9 states for each extra process, i.e., 1+M+(N-1)*(M-1) or N*(M-1) + 2 states for the run of Algorithm 2, compared to $M^N + 1$ states for the run of Algorithm 1 on the same protocol.

In the second test, because of the cycles, the reduction can only avoid the exploration of some redundant transitions, but still visits all reachable states. The improvement of the time efficiency is nevertheless quite notable. The relatively large memory requirements (compared to the first and the third test) are caused by the large stack that is required to explore all overlapping executions of N cyclic processes. In this test, the stack must be able to accommodate all reachable states.

The third test was added to measure the overhead that is incurred by the extra manipulation of edge labels in Algorithm 2. The test is constructed in such a way that no reduction can result, by making all statements refer to global variables. For Algorithm 2, the runtime overhead comes out at roughly 7%. There is no increase of the memory requirements.

The parameter settings in the test for the universal receiver protocol were chosen to produce a sample state space size that was easily manageable with both versions of the algorithm. Different state space sizes are easily created with this protocol, though each setting produces roughly the same relative differences between a run with Algorithm 1 and one with Algorithm 2. In the version tested, 3 asynchronous processes are run, communicating through a total of 6 different message channels of 5 slots each. A modest overall improvement of roughly 7% in runtime and 8% in memory requirements is obtained by Algorithm 2.

The last test is for the validation of a fairly typical modem protocol, that runs two processes, a sender and a receiver, and contains some local computations. This time a significant reduction of 78% in the run time and 62% in the memory usage is obtained by Algorithm 2.

The tests show that even with a relatively small change in the classic search algorithm, good improvements can be obtained without reducing the coverage of the search.

It is also clear that by a biased choice of examples (such as the first test protocol) an arbitrarily exaggerated impression of the improvements could be suggested. By setting N to 1000 in the first test, for instance, we can validate a state space that is equivalent to $10^{1000} + 1$ reachable states, by inspecting no more than 9002 states in mere seconds. Since no claim higher than 10^{20} states has appeared in print as yet [10], with this result we could claim to outperform any validation system in existence to date by no less than 980 orders of magnitude. Though such claims are popular, they are in essence vacuous.

4. CONFLICT SETS

To exploit a partial order search strategy fully, we must find a way to properly define the dependencies that may exist between transitions that touch also global objects. We will track these dependencies with the aid of 'conflict sets.' Conflict sets are an efficient version of 'sleep sets' [5].

In each running process, a distinct conflict set is assigned to each of the statements it can execute. If more than one process can execute the same code, the conflict sets for the statements in each process are distinct. To enforce the partial order discipline, the execution of statements must be blocked under certain conditions. If the conflict set of a statement is empty, it means that the statement is currently not blocked. If the conflict set contains entries ('conflict tags') the statement is blocked, and the entries represent the minimum conditions that must be satisfied for the statement to remain blocked.

We should be able to distinguish between a local effect of a statement execution and a global effect, that may have a side-effect on the executions in other processes. We first therefore define a special tag, called `Local`, that is assigned collectively to all *local variables* and all *symbolic constants* in the system. References to such objects can always be made without creating a dependency between process executions.

Next, each distinct *global variable* in the system is assigned a unique Read and Write tag that can be entered into the conflict set of the statement that refers to it. A simple variable name in PROMELA, the language of our target validator SPIN, always uniquely identifies a storage location for a value, and can not be used to indirectly point to other storage locations, as can be done with pointers in C for instance. For simplicity, we consider a reference to an array element to be a reference to the array as a whole (i.e., to the array name), so that it can be treated within the same framework as scalars. Usage of a statement such as

```
int a, b, c[N];
a = c[b+a+3];
```

where all variables are global, would involve the creation of the conflict tags

```
Write_a /* an assignment to global a     */
Read_c  /* a read of global array c[]    */
Read_b  /* a read of global b to index c */
Read_a  /* a read of global a to index c */
Local   /* a reference to the constant 3 */
```

where the effect of the last tag is canceled by the other four.

For message channels the situation is a little harder. A channel name really serves as a pointer to an internal buffer where values can be stored. It is valid to say in PROMELA

```
chan q = [SIZE] of { byte, int }; /* declarations   */
chan r = [SIZE] of { byte, int }; /* + initializers */

r = q;     /* set r to point to the same chan as q */
r!message; /* now has the same effect as q!message */
```

that is, to reassign a name to point to another channel. A reference to a message channel is therefore assigned two pairs of tags: one Read and one Write tag for references to the name itself, and one Send and one Receive tag for references to its current value (i.e., the actual channel location). Usage of a statement such as

```
r!message(3, a+b)
```

and assuming that `r` turns out to be the *N*th channel in the system, and that all variables are global, would involve the creation of the conflict tags

```
Read_r  /* a read of the channel name r */
Send_q_N       /* a send to channel location N */
Local   /* references to value '3' and mtype 'message' */
Read_b  /* a read of global b */
Read_a  /* a read of global a */
```

where the tag `Local` is again canceled by the other four tags.

Now let's see how the conflict tags are used to update the conflict sets that can enforce the blocking rules for statements. Each statement is assigned a unique conflict set, that is initially empty (i.e., non blocking). The tags created for a statement will ultimately be entered into its conflict set to create conditional blocks in future shufflings explored by the depth first search. The tag's name specifies the precise conditions under which that block should be lifted. If the only entry into a conflict set is the tag `Local`, the block is permanent, as in Algorithm 2. If a conflict set contains, for instance, the tag `Read_a`, the block on that statement is lifted as soon as any other statement in the system is executed that produces the tag `Write_a`, clearly, because reading and writing to the same global variable creates a dependency between two processes.

So, to enforce the partial order strategy, each statement execution should do two things:

1. Enter the appropriate tags into the conflict set of the current statement.
2. Clear the conflict sets of all other statements that contain a dependent tag.

Table I defines when two tags that point to the same global object (variable name, channel name, or channel location) are dependent.

TABLE I — DEPENDENCY RELATIONS

	Local	Read	Write	Send	Receive
Local	-	-	-	-	-
Read	-	-	+	+	+
Write	-	+	+	+	+
Send	-	+	+	+	x
Receive	-	+	+	x	+

A minus means that the two tags are always independent. Two read operations on the same object are clearly independent and can be shuffled in any order without changing the possible outcome of the read. A plus means that tags of the corresponding types are dependent when they refer to the same object. An x in the table means that the two tags may be dependent under certain extra conditions. A send and a receive operation on the same channel are independent whenever that message channel is non-full, or when a send operation on a full channel would be unexecutable anyway (the default SPIN semantics).

The only thing left to decide is now at what precise point in the depth first traversal should the conflict sets be updated? This, by no means a trivial point, is discussed next.

4.1. Algorithm 3

The conflict sets help us to determine if the execution of a statement must be repeated in the depth first search for different interleavings statement executions. This only needs to be done if another dependent statement could be shuffled ahead of the one considered. Consider a point in the depth first search where there are `N` processes and one of those, call it process `I`, has a nondeterministic choice between the execution of `M` different statements.

ALGORITHM 3 — CONFLICT SET UPDATE RULES

> **1.** Conflict tags are created upon the execution of each statement, *before* the traversal of the corresponding edge in the graph. All conflict sets are cleared that contain at least one tag that conflicts with any tag in the current set. Statements that belong to the same process are always assumed to be dependent, that is: the execution of any one of the statements in process `I` clears minimally the conflict sets of all other statements in `I`.
> **2.** The conflict tags are entered into the conflict sets of all `M` statements simultaneously. This happens *after* all states reachable from those `M` statements have been explored, and just *before* the next process is considered.
> **3.** The effect of all conflict set updates (clearings and entries) are reversed (undone) *after* all non-blocked transitions in all running processes have been explored, and just *before* the depth first search backs up to the previous state.

There are a few special cases that also have to be considered for a full implementation of these rules. The language PROMELA, for instance, allows for the definition of atomic sequences (i.e., sequence of statements that are executed in one indivisible step), rendez-vous communications etc. The discussion of the treatment of these special cases can safely be skipped here. In the measurements they did not play a role.

4.2. Measurements

Table II lists the results obtained with Algorithm 3, with and without enabling the reduction rules from Algorithm 2 (the rows labeled 2 and 2+3).

The test of protocol 3 is again intended to measure the overhead of the implementation of conflict sets alone, using an example where no optimization would result because all statements touch the same global variable. The comparison of Algorithms 1 and 3 shows that there is no significant overhead in the memory requirements, but a notable run time overhead is incurred for the manipulation of conflict sets. [Experiments were done with two independently produced implementations of the partial order rules. The figures in the table are for the most efficient implementation. That is to say, it is easy to do worse; it is probably hard to achieve a significantly better performance.]

For the second test, with *N* cycling processes, the combination of Algorithms 2 and 3 brings the number of states and transitions down to the level achieved in the first test by Algorithm 2 alone: a significant reduction. In this case, neither Algorithm 2 or 3 can achieve this effect in isolation. The differences in memory requirements are largely caused by the differences in the longest non-cyclic execution sequences encountered in each type of search (i.e., the maximum stack size).

In the last two tests the number of transitions traversed is reduced by the conflict sets to about half that explored in a classic search (or to 13% if Algorithms 2 and 3 are combined). For protocol 4, the reduction is not sufficient to make up for the additional run time overhead. For protocol 5, a small improvement does result. The combination of Algorithms 2 and 3 reduces both the number of states and the number of transitions explored and reduces the memory requirements. The run time requirements, however, are not always decreased.

> *The simplicity and effectiveness of Algorithm 2 makes it a painless first choice when partial order reduction strategies are considered. Algorithm 3 can provide an additional speedup by reducing the traversal of redundant transitions during a search.*

In Section 6 we show another application where this effect can prove to be beneficial. First, however, we discuss some further potential refinements of the method that was implemented, and a brief comparison to related work.

5. FURTHER EXTENSIONS

We have considered two sets of rules that can be used to reduce the number of states that have to be explored in reachability analyses: a preferential treatment of local transitions (Algorithm 2), and the usage of conflict set rules (Algorithm 3). In this section we consider how the treatment of non-local transitions might be optimized still further.

So far, we have not used any information about which process can access which variables. For the sake of simplicity, in what follows we will not make a distinction between read and write access. Consider, then, the following example, with three processes, with x, y, and z global variables.

```
A:  a0: access(y);  B:  b0: access(x);  C:  c0: access(x);
    a1: access(z);      b1: access(y);      c1: access(z);
        stop                goto b0             goto c0
```

Assume the system is in state <a0,b0,c0>. The transitions that are enabled in that state are access(y) of process A, access(x) of process B, and access(x) of process C. Since none of these transitions is local, Algorithm 2 will explore them all, to simulate all possible interleavings.

Starting at this state, however, it is not necessary to explore the transition access(y) in process A. Process A can never access variable x. Consequently, transitions access(x) of process B and access(x) of process C are independent from *all* transitions of process A. It is sufficient to select only the two transitions access(x) in order to explore all relevant futures of the system starting from state <a0,b0,c0>.

The reasoning we have just described can be automated by the following procedure, that is close to one that was first described by Overman as one of several alternative reduction algorithms [12]. Call the set of global variables that can be accessed by a process its *support*. For each global variable v_i, create the sets V_i and P_i and execute the following algorithm.

OVERMAN's METHOD

1. Set V_i to {v_i} and P_i to empty.
2. For each process whose support intersects V_i, add the global variables that the next transitions in this process can touch to set V_i and add the name of the process to P_i.
3. Repeat step 2 until no more variables can be added. Then, choose the set P_i that corresponds to the smallest non-zero number of enabled transitions in the processes that belong to it.

(In Overman's original version, step 3 selected the smallest set P_i with at least one non-blocked process.)

The *support* sets of A, B, and C in the example are {y,z}, {x,y} and {x,z} respectively. If we apply step 2 from Overman's method to variable y in state <a0,b0,c0>, processes A and B are added to P_y and since B is about to access x, variable x is added to V_y. The next step adds process C to set P_y and terminates the procedure. Applying step 2 to variable x immediately adds processes B and C to P_x but does not add any variables to V_x so the procedure terminates there. The set P_i that incurs the smallest set of enabled transitions is P_x and only transitions access(x) in process B and access(x) in process C then need to be explored.

Overman's method can thus also reduce the number of transitions that have to be explored. The drawback is a still further increased run time overhead. If all global variables can be accessed by all processes, the overhead is wasted and the application of this method cannot produce improvements.

The method can be refined still further by using information on the structure of the processes, i.e., by representing each process by a directed graph and by performing some analysis on these graphs. Indeed, the support set which was used in the above procedure includes variables which have been accessed by the process in the past as well as those which can be accessed by the process in the future. If some variables of the support set cannot be accessed anymore by the process from its current location, it is not necessary to take them into account in the support set.

Consider the example again and assume that the current state is `<a1,b1,c1>`. Applying step 2 of Overman's method to variable `y` adds processes `A` and `B` to set `P_y`. But, from state `a1`, process `A` cannot access variable `y` anymore. Hence it is not necessary to add `A` to `P_y`, and the procedure can stop. In this state it is sufficient to explore only transition `access(y)` of `B` since all transitions that could be performed by either `A` or `C` in the future are independent from it. Very frequently, however, the local process graphs are strongly connected, and in those cases the last optimization effort cannot help.

A related generalization of Overman's method was described and studied independently by Valmari in his *stubborn set* algorithms [6,13]. Loosely speaking, a stubborn set consists of transitions whose occurrence cannot be affected by other transitions in competing processes. Stubborn sets are comparable to Overman's set `P_i`. A stubborn set is computed for each state encountered during the reachability analysis and only enabled transitions that belong to this set are explored.

Unfortunately, the selection of the smallest set of enabled transitions at each step in these methods does not necessarily lead to the exploration of the smallest number of reachable states. A minimal selection can always be computed at an additional run time expense. Valmari reported that such a selection can be computed in quadratic time with respect to the number of transitions in the system, and when this is prohibitively expensive, "fairly good" stubborn sets can be computed in linear time with respect to the number of transitions [13]. It is an interesting topic for further study to see if these algorithms are competitive with the ones explored in this paper. At the time of writing, we have not done the experiments.

6. STATE COMPRESSION

Another worth-while attempt to reduce the memory requirements of a reachability analysis is to test the effect of various encodings of state descriptions before they are stored in a state space table. The application we have in mind is the standard reachability analysis algorithm, such as used in SPIN, with an on-the-fly generation and storage of the state space graph. Storage of states then has only one function: to prevent the repeated analysis of parts of the graph that are reachable via multiple paths.

The state compression for such an algorithm must, of course, be completely information preserving. Perhaps more relevant: it is important the state compression can be done with minimal time overhead, but there are no strict requirements on the complexity of a decompression. Comparison of states, to determine if a state has previously been visited, can be done in compressed form. Since the compressed version of a state description is naturally smaller than its uncompressed representation, the memory requirements of a reachability analysis can be reduced, and in some places the algorithm can even be sped-up, since the time requirement of a state comparison is linear in its size.

It is virtually unpredictable if the speed gain indicated above can balance out the speed loss due to the time required for state compression itself. Experiments with state compression algorithms were therefore conducted on a real-life protocol with a sufficiently large state space to make a meaningful measurement possible.

6.1. Algorithm 4

The standard algorithm for on-the-fly reachability analysis by depth first search is well known [1,9,11]. It is the counterpart of the explicit graph traversal algorithm illustrated in Algorithm 1. This time, we do not need any knowledge of the graph before the search begins. The graph is constructed, and all verification is done, on the fly. A version that uses state compression can be summarized as follows. Initially, the `Stack` contains only the initial system state.

ALGORITHM 4 — DEPTH-FIRST SEARCH WITH COMPRESSION

```
DFS()
{       while (Stack is nonempty)
        {       q  = last element of Stack
                if (q is error state)
                {       report the error
                        print the backtrace from the Stack
                } else
                {       for each successor s of q
                        {       s' = compress(s)
                                if (s' not in State_Space)
                                {       add s  to Stack
                                        add s' to State_Space
                                        DFS()    /* recursion */
                }       }       }
                delete q from Stack
}       }
```

As in the previous algorithms, no attempt is made here to describe also the detection of non-progress cycles or acceptance cycles, but all these are fairly straightforward extensions of the above basic algorithm. Note in particular that it is not necessary to construct strongly connected components in a graph to detect non-progress cycles or acceptance cycles (as is often suggested). The first description of a simple algorithm that performs this type of verification is due to Holzmann [7, pg. 235-238].

A requirement on the compression mechanism is that it has to be statically defined for the duration of a search. The reason is trivially that, to make state comparison possible, a given state must always be compressed in the same way, whether it is generated at the beginning or at the end of a search. Dynamic Huffman coding, or Lempel-Zvi & Welch compression are therefore not directly usable.

6.2. Measurements

Two types of state compression have the required property and were tried as part of Algorithm 4: *static* Huffman encoding (Algorithm 4 in Table II), and run length encoding (Algorithm 4’ in Table II). For the Huffman encoding [14], the relative frequency of byte values in state descriptions was measured over a range of protocols, and then hard-coded. Values close to zero can be expected to occur more frequently than larger values, and the results confirm that. The two compression schemes were applied to a single test protocol with a sufficiently large number of states (DTP). No other aspect of a protocol, other than the number of reachable states, is relevant in such an experiment, so a single test-case sufficed.

The results shown in Table II (rows 1+4 and 1+4’) are easy to interpret.

> *Run length encoding adds a large run time overhead (413%) in return for a modest (18%) reduction of the memory requirements. Huffman encoding adds a smaller run time overhead (302%) in return for a more substantial reduction of the memory requirements (63%).*

Combination with Partial Orders

The state compression technique can easily be combined with partial order reduction techniques. The results are included in Table II, which is presented below.

TABLE II — RELATIVE PERFORMANCE OF ALGORITHMS

Protocol	Algorithm	States	Transitions	Time(sec.)	Memory (Mb)
1 (N=5,M=10)	1	100,001	450,002	16.8	5.1
	2	47	47	(<0.01)	1.0
	3	100,001	100,001	9.2	5.1
	2+3	47	47	(<0.01)	1.3
2 (N=5,M=10)	1	100,001	500,002	15.6	11.1
	2	100,001	111,112	4.0	11.1
	3	100,001	111,112	8.3	9.1
	2+3	47	52	(<0.01)	1.1
3 (N=5,M=10)	1	100,001	450,002	18.7	5.46
	2	100,001	450,002	20.2	5.46
	3	100,001	450,002	63.4	5.47
	2+3	100,001	450,002	66.4	5.47
4 (URP)	1	19,515	47,836	2.9	3.5
	2	17,163	36,239	2.7	3.2
	3	19,515	21,236	11.7	3.5
	2+3	15,628	17,056	9.0	3.1
	2+4	17,163	36,239	5.5	2.0
	2+3+4	15,628	17,056	10.2	1.9
5 (DTP)	1	251,409	648,467	40.8	36.6
	2	91,343	117,396	8.7	13.9
	3	251,409	262,561	59.9	36.6
	1+4’	251,409	648,467	114.4	33.5
	1+4	251,409	648,467	83.5	15.1
	2+3	65,774	69,611	13.8	10.4
	2+4	91,343	117,396	16.3	6.2
	2+3+4	65,774	69,611	18.2	4.8

For easier comparison, the run time and memory usage data for the most relevant validations of the URP and DTP protocols are shown graphically, in ‘lolliplot’ format in Figure 3.

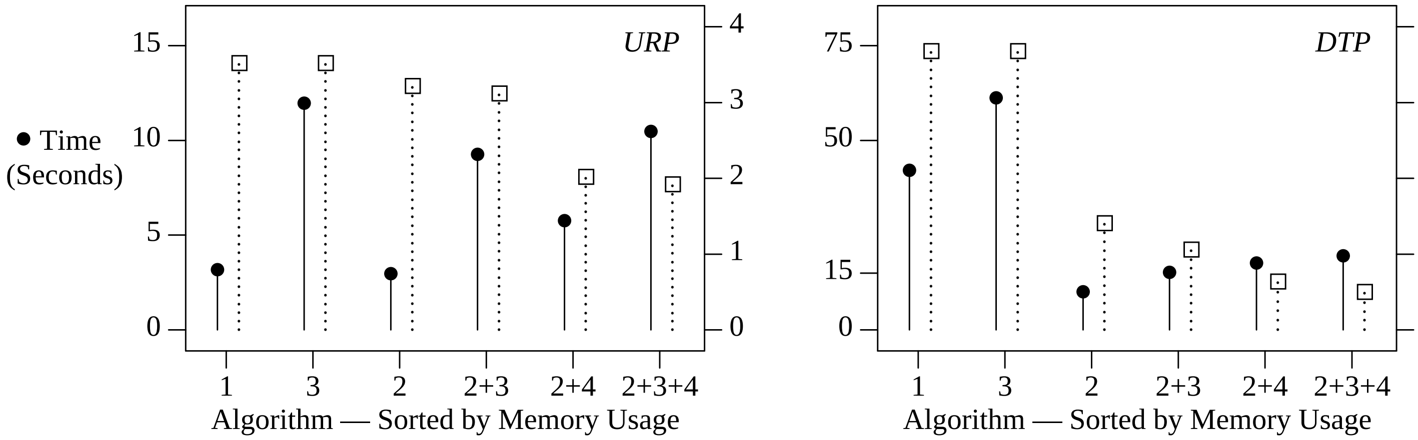

Figure 3 — Runtime and Memory Usage for URP and DTP Validations

The main conclusions can be summarized as follows.

The fastest run is obtained with Algorithm 2 used separately. The speed up is 6.8% for the first protocol, and 78% for the second. The lowest memory requirements result from a combination of Algorithms 2, 3 and 4. The reduction is 46% for the first protocol, and 87% for the second.

The state compression method is sensitive to the number of transitions that is traversed during a search. Note that each extra transition requires an extra state compression and a comparison to the state table to verify that the newly generated state was previously unvisited. Since the effect of Algorithm 3 is precisely to reduce the number of transitions it can produce a more significant reduction of the run time requirements with state compression enabled.

7. SUMMARY

As a starting point for this work we took the familiar implementations of a depth first graph traversal, illustrated by Algorithm 1, and its counterpart for on-the-fly verification, illustrated in Algorithm 4. Both algorithms have appeared in print before [1,7]. We have studied potential improvements of these classic algorithms with two simple generalizations of recently developed partial order semantics rules [5] and of a simple state compression scheme, using SPIN as a testbed for comparisons.

The effort required to upgrade the classic search into the reduced search from Algorithm 2 can be measured in hours, and the addition of no more than twenty lines of code to the source of SPIN. The implementation of the more sophisticated reduction strategy from Algorithm 3 was much more time consuming. The implementation of the state compression scheme from Algorithm 4 was, much like Algorithm 2, a matter of hours, and gave good reductions of state space complexity, though only in return for a non-negligible run time overhead. In combination with the more complete implementation of partial order reduction rules, however, the run time increase of the state compression scheme was reduced. The combination creates a competitive algorithm that can substantially reduce the memory requirements of a search without too seriously affecting its run time requirements.

Acknowledgements

The work of the last two authors was partially supported by the European Community ESPRIT BRA project SPEC (3096) and by the Incentive Program ‘Information Technology,’ Computer Science of the Future, initiated by Belgian State, Prime Minister’s Service, Science Policy Office. The scientific responsibility is assumed by the authors.

8. REFERENCES

1. Holzmann, G.J., ‘‘Algorithms for automated protocol validation,’’ *AT&T Techn. Journal*, Special issue on Protocol Testing and Verification. 1990, Vol 69, No 1, pp. 32-44.

2. Cunha, P.R.F., and Maibaum, T.S.E. ‘‘A synchronization calculus for message oriented programming,’’ *Proc. Int. Conf. on Distributed Systems*, 1981, IEEE, pp. 433-445.

3. *Proc. 2nd Workshop on Computer Aided Verification*, LNCS 531, Eds. R. Kurshan and E. Clarke, New Brunswick, New Jersey, June 18-21, 1990.

4. *Proc. 3rd Workshop on Computer Aided Verification*, Eds. K. Larsen and Arne Skou, Aalborg, Denmark, July 1-4, 1991.

5. Godefroid, P., and Wolper P., ‘‘Using partial orders for the efficient verification of deadlock freedom and safety properties,’’ in [3].

6. Valmari, A., and Tienari, M. ‘‘Improved failure equivalence for finite state systems with a reduction algorithm,’’ *Proc. 11-th Symp on Protocol Spec. Testing, and Verif.*, Sweden, 1991.

7. Holzmann, G.J., *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

8. Holzmann, G.J. ‘‘An improved protocol reachability analysis technique,’’ *Software, Practice and Experience*, Vol 18, No. 2, Feb. 1988, pp. 137-161.

9. Holzmann, G.J., ‘‘Tracing protocols,’’ *AT&T Techn. J.*, Vol 64, No. 12, pp. 2413-2434.

10. Burch, J., et al., ‘‘Symbolic model checking, 10^{20} states and beyond,’’ *Proc. 5th Symp. on Logic in Computer Science*, Philadelphia, June 1990.

11. Holzmann, G.J. ‘‘Automated protocol validation in Argos — assertion proving and scatter searching,’’ IEEE Trans. on Software Engineering, SE-13, No. 6, June 1987, 683-696.

12. Overman, W.T., ‘‘Verification of concurrent systems: functions and timing,’’ PhD Thesis, University of California, Los Angeles 1981, 174 pgs.

13. Valmari, A. ‘‘Stubborn sets for reduced state space generation,’’ *Proc. 10th*

International Conference on Application and Theory of Petri Nets - Vol 2, Bonn, 1989, pp. 1-22, also in "Advances in Petri Nets 90", LNCS 483.
14. Knuth, D.E., *The Art of Computer Programming*, Vol 1, Addison-Wesley, 1973.