An Alternative to Model Checking: Verification by Random Search of AND-OR Graphs Representing Finite-State Models

David Owen, Bojan Cukic, Tim Menzies
Lane Department of Computer Science and Electrical Engineering
West Virginia University
PO Box 6109 Morgantown, WV 26506-6109, USA
{dowen|cukic}@csee.wvu.edu, tim@menzies.com

Abstract

ABSTRACT
In the development of high-assurance systems, formal modeling, analysis and verification techniques are playing an increasingly important role. In spite of significant advances, formal modeling and verification still suffers from limited applicability due to exponential runtime space growth exhibited by model checkers.

In this paper, we describe an alternative to model checking. We describe an algorithm that automatically translates Finite State Machine models used by model checkers into AND-OR graphs. State space verification of AND-OR graphs does not suffer from state space explosion, but its exhaustive search is an NP complete problem. Hence, we demonstrate that random searches of AND-OR graphs are a viable alternative to model checking, suitable for system debugging and fast analysis during system development. We support our conclusions through the studies of two models, Dekker’s two process mutual exclusion algorithm and the Space Shuttle’s liquid hydrogen subsystem.

1 Introduction

Formal modelling, analysis and verification are very active research areas in software assurance. There is little doubt that the application of formal methods throughout the software development life cycle improves software reliability (e.g., the long list of applications in [6]). However, doubts exist concerning the practicality and cost of formal methods:

- The cost of writing the formal model or queries, referred to below as the writing cost—often, Ph.D.-level mathematical expertise is required for writing such models.
- The cost of executing queries over the formal model, the running cost—a formal query may be prohibitively slow or require too much memory to execute over the exponentially large formal model representing all possible interactions of components in the original system.
- The cost of modifying the formal model, the rewriting cost—analysts often rewrite formal models using various strategies to minimize the running cost.

Many researchers have worked to reduce these costs using a variety of methods including the design of special restricted modelling languages, automatic generation of models from code, and optimizations that exploit models’ symmetry. Much progress has been made to reduce the writing cost, but the general problem of high running cost persists despite decades of effort.

In an attempt to minimize the running cost, and therefore eliminate the rewriting cost, which is generally due to unacceptably long verification run times, we compile formal models into compact NAYO (a type of AND-OR) graphs [10–12]. With a NAYO graph it is possible to represent all possible interactions of the individual components of a formal model in a much smaller space than that required by a finite-state machine representing the same information. Unfortunately, complete search of NAYO graphs is in general intractable (it is NP-complete, as we will show later), so the running cost is still a problem.

The alternative to complete search is incomplete search. A recent but repeated result in the artificial intelligence literature is that incomplete search may be surprisingly effective. The logical form of complete NAYO search is similar to the satisfiability (SAT) problem [9]. Partial random search is an incomplete strategy often used to solve SAT:

- When competing constraints block progress, a single constraint, selected at random, is favored.
- Future conflicts are also, in effect, resolved randomly, i.e., the search finds a randomly selected subset of the formal model; hence it is incomplete.
• Random search is run, reset, and repeated $n$ times. The best solution seen in any run is returned.

Random search is capable of finding optimal or nearly optimal results for large satisfiability problems [7, 8, 15]. In addition, random search finds a result even when exhaustive search is not feasible. The success of partial random search in artificial intelligence motivates our use of the same type of search in software engineering. Our research has brought us to the following conclusion:

Random search over formal models exhibits a saturation effect.

Figure 1. The saturation effect.

Figure 1 illustrates the statement above. The curve marked saturation represents random search results for a model in which everything possible to find was found quickly and then a saturation point was reached. After saturation, a level plateau indicates that extra effort by the search can not discover any new unique results.

Assessment methods lacking the saturation effect have the following property: the more time spent on the assessment, the more unique results found (e.g., the curve marked no saturation in Figure 1). On the other hand, assessment methods that do exhibit a saturation effect support early-stopping rules, which can be used to reduce the cost of formal analysis (the running cost). We can stop searching a formal model when it is very unlikely continued search will uncover new results, i.e., after the saturation plateau is encountered.

When we use early-stopping rules, we risk false positives—we may conclude that no faults are present, when further assessment would have eventually found them. Hence we endorse early stopping only for assessment methods that exhibit the following properties:

• Adequacy—an adequate assessment method does not fail to recognize faults in the portion of the model explored prior to early stopping.

• Flat Plateaus—if the plotted search result is a flat plateau (e.g., the curve marked saturation in Figure 1) then additional errors, if present in the unexplored portion of the model, are not detected; however, if the model has been written correctly, any errors in this portion of the model are as likely to remain unnoticed upon system deployment as they are by our search.

We claim that random search of NAYO graphs generated from formal models written as finite-state machines is adequate and demonstrates flat plateaus. This justifies the use of early stopping rules, which make random search an effective strategy for decreasing the running and rewrite cost of formal verification and an exciting alternative to other techniques including model checking.

The rest of the paper is organized as follows. Section 2 introduces traditional formal modeling techniques and describes the algorithms for automated model translation into NAYO graphs. The same section provides the proof that the exhaustive search of NAYO graphs is an NP complete problem, necessitating the use of random search techniques. Section 3 demonstrates the application of NAYO graph modeling and analysis to two problems taken from the literature and evaluates their success. Section 4 concludes the paper and outlines the directions for the further development.

2 Formal Models Written as Finite-State Machines

This section begins with our formal definition of communicating finite-state machines (2.1), or FSMs. This material is condensed from [13]. Model checking tools use the same form, with some minor variations, to represent programs with concurrent processes [5]. To verify that a model matches a property specification, a model checker must build an exponentially large composite finite-state machine representing all possible interleavings of the individual FSMs in the original model. We show how a NAYO graph may be used to represent the same information, with size just polynomial in the size of the input; then we show that exhaustive search of a NAYO graph is an NP-complete problem (2.2). We then describe the partial random search procedure used to search our AND-OR graphs (2.3).

2.1 Communicating FSMs

We define a system $S$ of communicating FSMs in the following way:

• Each FSM $M \in S$ is a 3-tuple $(Q, \Sigma, \delta)$.
• $Q$ is a finite set of states.
• $\Sigma$ is a finite set of input/output symbols.
Figure 2. A system of communicating FSMs (\(m\) is a message passed between the machines).

Figure 2 shows a very simple communicating FSM model. States are indicated by labelled ovals, and edges represent transitions that are triggered by input and that result in output. Edges are labelled: input / output. We have observed that in the variety of existing FSM schemes there are two different ways individual machines communicate. Because of this, we define two different kinds input/output symbols (included in \(\Sigma\)):

1. A transition in one machine may be triggered by the fact that another machine is in a particular state, or the effect of a transition may be to change the state of another machine.
2. A transition may be triggered by a message received from another machine, or the effect of a transition may be to send a message.

The key difference between states and messages is in their use as transition inputs. A transition triggered by a message consumes the message, so that it is no longer able to trigger another. But states are unaffected by transitions they trigger; they are good for an arbitrary number of transitions.

2.2 NAYO Graph Translation

Figure 3 shows an AND-OR graph equivalent to the communicating FSM model shown in Figure 2. We call this type of AND-OR graph a NAYO since it contains the following features:

- A set \(N\) of undirected NO-edges connecting incompatible nodes.
- A set \(A\) of AND-nodes—an AND-node is TRUE if all of its YES-edge parents are TRUE.
- A set \(Y\) of directed YES-edges.

A close look at Figure 3 reveals something strange: there is an edge from node \(A1\) to the upper right AND-node, and another edge going from the AND-node back to node \(A1\) (the same thing occurs with node \(B2\) and the lower right AND-node). To understand this, we reiterate the point made above, that there are two different ways FSMs communicate: states and messages. And the key difference between them is that messages are consumed. We use a simple trick to represent this in a NAYO graph. First, we define the search so that any time an AND-node is reached its parents are consumed—they are no longer available to be used again. Then, for nodes representing the state-fron another machine type of input that should not be consumed (e.g., \(A1\)), we add an extra edge from the AND-node back to the state node—we consume but then immediately regenerate the state node, so that it is available to be used again.

```
1: for (each finite-state machine) do
2:   for (each state) do
3:     Make an OR-node; connect it with a NO-edge to each OR-node representing another of this machine’s states.
4:   end for
5: for (each transition in this finite-state machine) do
6:   Make an AND-node;
7:   Make current state a YES-edge parent of the AND-node;
8:   Make input(s) (a) YES-edge parent(s) of the AND-node;
9:   Make next state a YES-edge child of the AND-node;
10:  Make output(s) (a) YES-edge child(ren) of the AND-node.
11:  if (input is a state from another machine) then
12:    Make input a YES-edge child of the AND-node;
13:  end if
14: end for
15: end for
```

Figure 4. Automatic translation procedure from FSMs to NAYOs.

Figure 4 shows the procedure used to automatically translate from a communicating FSM model, e.g., Figure 2, to a NAYO graph, e.g., Figure 3. In general, for a system of \(k\) FSMs with \(n\) states and \(m\) single-input, single-output transitions per machine, the resulting NAYO has:

- \(mk\) AND-nodes + \(nk\) OR-nodes = \(O((m + n)k)\) nodes.
Table 1. Size comparison of NAYO and FSM composite for three formal models.

<table>
<thead>
<tr>
<th>Model</th>
<th>States in FSM Composite (upper bound)</th>
<th>Nodes in NAYO Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dekker 2-Process Mutual Exclusion Model from [4]</td>
<td>2,304</td>
<td>40</td>
</tr>
<tr>
<td>TCP Protocol Model from [14]</td>
<td>2,467</td>
<td>84</td>
</tr>
<tr>
<td>SCR Specification Model from [1]</td>
<td>1.68x10^8</td>
<td>126</td>
</tr>
<tr>
<td>Large Randomly Generated Model</td>
<td>2.65x10^178</td>
<td>4,007</td>
</tr>
</tbody>
</table>

- $4nk$ YES-edges + $(n/2)(n - 1)k$ NO-edges = $O((m + n^2)k)$ edges.

An FSM composite (which would be exhaustively searched by a model checker) for the same system will in the worst case require $O(n^6)$ states and $O(n^6)$ transitions [5]. Table 1 compares the size of FSM composite representations and NAYO graphs for several formal models, including the two we use later for random search examples.

Figure 5. NAYO graph representing the 3SAT query $(x_1 \lor x_2 \lor x_3) \land (\overline{x_1} \lor \overline{x_2} \lor \overline{x_3})$.

Unfortunately, the problem of determining whether a particular node in the NAYO graph can be reached is NP-complete,¹ which we show here in two steps.

3SAT $\leq_P$ NAYO search (NAYO search is at least as hard as the 3SAT problem, which is known to be NP-complete): for the 3SAT problem we have a Boolean expression that is the conjunction of a series of clauses, each of which is the disjunction of 3 literals. A literal is either a variable ($x_i$ for example) or its negation ($\overline{x_i}$). The problem is to determine whether the expression is satisfiable; that is, does there exist an assignment of values to the variables that satisfies the total conjunction? Figure 5 shows a NAYO graph representing a very simple 3SAT query. A NAYO graph for a 3SAT query will have a single AND-node; if this AND-node can be reached then the original 3SAT query is satisfiable.

NAYO Search $\in$ NP: clearly we can verify a NAYO search solution in polynomial time; we would (1) verify that the solution is a valid path of YES-edges, which requires $O((n-1)$ time (where $n$ is the number of nodes in the NAYO graph); (2) verify that no two nodes in the solution path are connected by a NO-edge, which requires $O(n(n-1))$ time.

2.3 NAYO Random Search

Our NAYO random search is designed to solve the following problem: given some (not necessarily consistent) input set of OR-nodes, find an output set consistent with at least part of the input, and make that output set as large as possible. Ideally the output set contains an OR-node for a state in each of the finite-state machines from the original model—if so, the output is equivalent to one of the states in the composite finite-state machine that would be searched by a model checker (and we have found it without explicitly constructing the composite). But in general, because the random search is not exhaustive, it may not tell us quite as much as a more time- (and space-) consuming technique; that is, the output set will constitute a partial description of a state in the composite.

Once we have the first output set, we use it as the input set for time $t+1$. Figure 6 shows the sets of nodes involved in successive iterations of the random search procedure shown in Figure 7.

Figure 6. Sets involved in successive iterations of the random search procedure shown in Figure 7.

Once we have the first output set, we use it as the input set for time $t+1$. Figure 6 shows the sets of nodes involved in successive iterations of the random search procedure. At time $t = i$, the search builds an output set of consistent nodes. These are marked true at time $t = i$ in Figure 6.

The set marked frontier is the set of nodes that will serve as input for the next iteration. The frontier includes (1) all nodes true now and (2) nodes that are almost true, i.e., nodes

---

¹NP is the class of problems for which a solution can be verified in polynomial time (the time required is a polynomial function of the input size); an NP-complete problem is (1) at least as hard as all problems in the class NP and is (2) itself in NP.
that are implied by but contradict nodes true now. In the next iteration \((\text{time} = i + 1)\) we start with the frontier (which will include contradictions) as the input set and use it to build a consistent output set of nodes true at \(\text{time} = i + 1\).

1: OR-nodes' wait field ← 1.
2: AND-nodes' wait field ← |parents|.
3: while \((\text{time} \leq \text{MAX})\) do
4: while \((Q \neq \emptyset)\) do
5: \(n \leftarrow \text{pop}(Q)\).
6: if \((n \text{ not disqualified})\) then
7: Mark \(n\) true at current time.
8: for \((\forall n' \text{ linked to } n \text{ by a NO-edge})\) do
9: Mark \(n'\) disqualified at current time.
10: end for
11: for \((\forall \text{YES-edge children } n' \text{ of } n)\) do
12: Decrement \(n'\) wait field.
13: if \((n' \text{ wait } = 0)\) then
14: Mark \(n'\) reached at current time.
15: \(Q \leftarrow n'\) at random index.
16: end if
17: end for
18: end if
19: end while
20: \(Q \leftarrow \text{all nodes reached at current time at random index (including nodes disqualified at current time)}\).
21: Reset all other nodes' wait fields (as in lines 1-2).
22: Increment time.
23: end while

**Figure 7. Random search procedure for NAYO graphs.**

Figure 7 shows the random search procedure used to explore NAYO graphs. Each time the search comes to a node its wait field is decremented. When wait = 0, the node is reached. An OR-node's wait need only be decremented once, because we only need to reach it via one of its parents; so OR-nodes wait fields are initialized to 1 (line 1). To reach an AND-node, we must first reach all of its parents, so its wait field is initialized to its number of parents (line 2).

The central part of the search procedure occurs in lines 4-19. We begin with an input set of nodes in the \(Q\), in no particular order. The first node is removed from the \(Q\) (line 5). If it has not been disqualified, i.e., it does not contradict some node we already believe true at the current time, we explore its children. All children via NO-edges are disqualified (line 9). The wait fields of all children via YES-edges are decremented (line 12), and if any are decremented all the way to zero, they are put into the \(Q\) at some random index (line 15). This process continues until the \(Q\) is empty (line 4).

Once all nodes in the \(Q\) have been processed, lines 20-22 set us up for the next iteration. At this point there is a set of nodes marked true at the current time, which is a subset of the nodes marked reached at the current time (some nodes are reached but disqualified, so they are never marked true). The true set corresponds to the set true at time \(= i\) in Figure 6, and the reached set corresponds to the frontier set in Figure 6. The reached set is put back into the \(Q\) (line 20) to serve as input for the next iteration. All other nodes’ wait fields are reset, and the time is incremented (lines 21-22).

**Figure 8. Comparison of output type for exhaustive FSM search and our partial random NAYO graph search.**

Figure 8 shows a comparison of search results for (1) exhaustive search of a composite finite-state machine, which generates an execution path of fully defined global states, and (2) our NAYO search, which generates an execution path of partially defined global states (without explicitly constructing the exponentially large composite FSM representing the global system). The search continues for a specified number of iterations or until it hits a dead end. The search is random in that, when there are two or more contradictory nodes that might be added to the output set, the choice of which node to add is random.

### 3 Random Search of Example Models

In (3.1) and (3.2) following we use our NAYO random search to verify properties of formal models written as FSMs. In the first case, we show how a particular node representing a violation of a simple safety property is found by the search in a formal model to which an error has been added. In the second example, we use the search to show that every state except those representing external information is reachable during execution of the formal model.

In these examples, we use in each search iteration, as inputs for the random search, a random set of consistent nodes (a partial description of a global state—see Figure 8). Our output is a set of nodes from all iterations reachable from but not including inputs.
3.1 Promela Formal Model: Dekker’s Solution to the 2-Process Mutual Exclusion Problem

Figure 9 shows Dekker’s solution to the two-process mutual exclusion problem written in Promela (from [4]), which is the input language used with the model checker SPIN [6]. Promela has been designed to look like a high-level programming language, but represents communicating FSMs. SPIN is capable of automatically generating the finite-state machine version of a Promela model; Figure 10 shows the model from Figure 9 in this form.

Figure 11 shows the result of a series of random searches on a NAYO graph representing the finite-state model of Dekker’s mutual exclusion solution from Figure 10. In order to show that our random search is capable of finding a fault, we have added to our NAYO graph a Boolean variable called safe, which is initially true and becomes false if proctype A and proctype B are ever simultaneously in state 4 (the critical section). We have also added the equivalent of the following transition to proctype A(), allowing it to go directly into its critical section without checking variables y and t:

\[
\text{state 3} \rightarrow \text{state 4} \Rightarrow (\text{true})
\]

The searches shown in Figure 11 are typical of hundreds of experiments. In every fault-free Dekker model search, we quickly find all but one of the OR-nodes in the NAYO graph (23 of 24)—we never find the node safe = false. In the model with the fault, we quickly find every node (all 24), including the node representing the fault. To give an idea of exactly how quickly the random search explored the model, the size composite finite-state machine (the form of the model that would be searched by a model checker) for this model is bounded at 2,304 states. Figure 11 shows that our NAYO search reached saturation after processing about 200 OR-nodes.

3.2 SCR Specification Example: the Space Shuttle Liquid Hydrogen Subsystem

The Software Cost Reduction (SCR) language, used for writing software requirements specifications, is formally based on finite-state machines [3]. It is relatively easy to rewrite SCR specifications as FSMs, as long as the specification does not use variables with a large number of possible values. In practice, this means making some assumptions about key values and creating a new variable that can take on only those key values (the abstraction strategy used often in creating formal models, e.g., Clarke et.al. [2]).

In a technical report comparing SCR-based model checking tools (incorporating SPIN) to the SMV model checker, Atanacio includes an SCR specification for the Space Shuttle Liquid Hydrogen Subsystem [1]. A set of FSMs representing the specification was automatically translated to a NAYO graph using the procedure outlined in Figure 4. In creating FSMs from the specification it was not necessary to reduce the number of values taken by any of the variables; this had already been done when the original SCR specification was written, in order to decrease the state space (and therefore the amount of memory and time) required by the model checker SPIN, which was used in conjunction with SCR tools to verify the model.

Figure 12 shows results for a series of random searches on the NAYO graph version of the Space Shuttle Liquid Hydrogen Subsystem. There are 10 searches plotted in Figure 12. The results show the basic shape we are looking for.
unique OR-nodes reached (x-axis) during that trial.

**Figure 11. Search results for model of Dekker's solution to the two-process mutual exclusion problem.**

Dekker model with an error added; the dots indicate at what point in a particular trial the error node was reached.

Each plot shows ten trials covering a range of MAX time values; for each trial the search in Figure 7 was repeated many times, each time with a random set of inputs, keeping track of the total OR-nodes processed (y-axis) and the unique OR-nodes reached (x-axis) during that trial.

**Figure 12. Result of random searches on NAYO graph version of SCR specification for the Space Shuttle Liquid Hydrogen Subsystem [1].**

For—a quick rise to saturation and then a plateau that then remains level indefinitely. We observe in Figure 12 that the number of unique OR-nodes reached rose to a plateau of height 52 (out of 62 total OR-nodes in the NAYO graph) and stayed there, and this happened in all of trials.

Why 52? Why were we unable to reach the other 10 OR-nodes? A close look at the SCR specification from [1] shows three monitored variables (representing information in the environment external to the system), the first taking on 2 possible values, the second 4, and the third 4 as well. For each of these 10 values there is an OR-node in the NAYO graph—an OR-node we would not expect to reach by our search except as part of the input, since it represents activity outside the system. So for this model, which is small enough to read and understand directly, the random search finds everything we expect to be able to find.

Figure 12 shows that our random search reached saturation after about 5,000 OR-nodes were processed, which may seem like a lot compared to the previous example. But the upper bound on the composite FSM (that would be searched by a model checker) in this case is about $1.68 \times 10^9$, which is about $34,000 \times 5000$.

### 4 Summary and Further Work

In this paper we presented a practical alternative to model-checking: the random search of AND-OR graphs, which are automatically generated from finite state machine representations. Unlike model checking, random search of AND-OR graph is a very quick analysis method which does not suffer from state space explosion. Random searches are utilised instead of exhaustive graph traversals because the algorithms achieving the later are NP-complete, as proven in Section 2. The application of random search obviously limits the rigour of verification, because faults can hide in portions of the state space not explored in random trials. This weakness is only partially mitigated by the fact that random searches of NAYO graphs exhibit saturation, i.e., prolonged searches do not include states unvisited in the (system specific) small number of random trials. In case of high assurance systems, rather than recommending our methodology for (sub)system verification, we suggest its usage during system development and debugging, when the number of logical faults is higher than in the later stages of the development lifecycle, and when their removal is the most cost effective.
The case studies reported in this paper provide very encouraging results. The translation of Finite State Machines to NAYO graphs is fully automated and supported by the tool. We intend to perform additional case studies and, hopefully, reconfirm the saturation effects observed in all the examples so far. We also plan to analyze the correlation between the search saturation and the design structures that lead to it. This will lead us towards providing the set of design recommendations, with the goal of making the models and ensuing designs more testable and/or easier to verify.

References


