

*ing. We describe an algorithm that translates Finite State Machine models into AND-OR graphs. State space graphs does not suffer from exhaustive search is an NP-complete problem. We demonstrate that random search is a viable alternative to model checking for debugging and fast analysis of large models. We support our conclusions through a case study of Dekker's two process mutual exclusion algorithm and the Space Shuttle's liquid hydrogen system.*

## **1 Introduction**

Formal modelling, analysis and verification are active research areas in software engineering. There is no doubt that the application of formal methods to the software development life cycle is a major challenge. The ability (e.g., the long list of open problems) to solve these doubts exist concerning the practicality of formal methods:

- The cost of writing the models is often referred to below as the *writing cost*. The lack of mathematical expertise in writing models.

## Figure 1. The s

Figure 1 illustrates the s  
marked *saturation* represent  
model in which everything  
quickly and then a *saturation*  
saturation, a level plateau in  
search can not discover any r

Assessment methods lack  
the following property: the r  
ment, the more unique results  
*no saturation* in Figure 1). C  
methods that do exhibit a sa  
stopping rules, which can be  
mal analysis (the running co  
formal model when it is very  
uncover new results, i.e., aft  
countered.

When we use early-stopp  
*tives*—we may conclude tha  
further assessment would b  
Hence we endorse early stopp  
ods that exhibit the following

- *Adequacy*—an adequate  
fail to recognize faults i  
plored prior to early stop

*symbols* (included in  $\Sigma$ ):

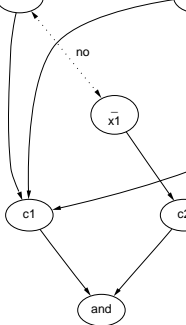
1. A transition in one machine may be triggered by the fact that another machine has reached a certain state. The effect of a transition may be to change the state of another machine.
2. A transition may be triggered by a message received from another machine, or by a message sent by the machine. The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

The key difference between these two types of transitions is their use as transition inputs. In the first case, the message *consumes* the message it is used to trigger another. But states are not consumed; they trigger; they are good for reuse in other situations.

## 2.2 NAYO Graph Transitions

Figure 3 shows an AND-OR graph representing a communicating FSM model. This type of AND-OR graph has the following features:

- A set  $N$  of undirected NAYO nodes.
- A set  $A$  of AND-nodes, each with a set of YES-edge parents.
- A set  $Y$  of directed YES-edges.



**Figure 5. NAYO graph**  
**query**  $(x_1 \vee x_2 \vee x_3) \wedge \dots$

Unfortunately, the problem of finding a particular node in the NAYO graph is not complete,<sup>1</sup> which we show here.

$3SAT \leq_P$  NAYO search (NAYO search is as hard as the 3SAT problem, which is NP-complete). For the 3SAT problem we have a formula that is the conjunction of a series of clauses, each a disjunction of 3 literals. A lit

---

<sup>1</sup> $NP$  is the class of problems for which there is a polynomial time (the time required to verify a solution of size  $n$ ) algorithm. An  $NP$ -complete problem is (1) in the class  $NP$  and is (2) itself in  $NP$ .

Figure 7 shows the random explore NAYO graphs. Each time its *wait* field is decremented and is *reached*. An OR-node's *wait* field is decremented once, because we only need to explore its children once; so OR-nodes *wait* fields reach an AND-node, we must explore its children, so its *wait* field is initialized to 2).

The central part of the search is shown in Figure 8. We begin with an input node and explore it in a particular order. The first node is explored (line 5). If it has not been disqualified, we explore some node we already believe is true (line 6). We explore its children. All children are explored (line 9). The *wait* fields are decremented (line 12), and when they reach zero, they are put into the *reached* set (line 15). This process continues until all nodes in  $Q$  are explored (line 4).

Once all nodes in the  $Q$  have been explored, we set us up for the next iteration. The nodes marked *true* at the current iteration are the nodes marked *reached* at the current iteration. The nodes marked *reached* at the current iteration are reached but disqualified, so they are not in the *true* set. The *true* set corresponds to the nodes in Figure 6, and the *reached* set corresponds to the nodes in Figure 7.

Promela has been designed as a programming language, but represents SPIN is capable of automatic generation of a Promela model from Figure 9 in this figure.

Figure 11 shows the result of a search on a NAYO graph representing Dekker's mutual exclusion solution. In order to show that our random search is not a false negative, we have added to our model a variable called *safe*, which is initialized to true. We have also added a prototype A and prototype B and a critical section of length 4 (the critical section). We have added the following transition to the model, which allows a process to enter directly into its critical section:

```
state 3 -> state 4
```

The searches shown in Figure 11 are the result of 24 experiments. In every case, we quickly find all but one of the models (23 of 24)—we never find the model with the fault, we find the model (24), including the node representing the fault. The idea of exactly how *quickly* the search finds the model, the size of the composite first found, and the model that would be searched first in this model is bounded at 2,300. In our NAYO search reached safe

Dekker model with an error at what point in a particular search was reached.

Each plot shows ten trials of search values; for each trial the search was repeated many times, each time with a different starting point, keeping track of the total OR-nodes reached and the unique OR-nodes reached.

## **Figure 11. Search Dekker's solution to the exclusion problem.**

for—a quick rise to saturation and then remains level indefinitely. We found that the number of unique OR-nodes reached was at height 52 (out of 62 total OR-nodes) and stayed there, and this happened

Why 52? Why were we not reaching 62 OR-nodes? A close look at the search graph shows three monitored variables that were changing in the environment external to the search, each taking on 2 possible values, the search was not doing well. For each of these 10 variables, we built a NAYO graph—an OR-node graph that was not reached by our search except as part of

- puter Protocols*. Prentice-Hall, 1991. Available at <http://cm.bell-labs.com/ospin/Doc/Book91.htm>
- [6] G. Holzmann. *The Model Checking Book*. Prentice-Hall, 1997.
- [7] J. Horgan and A. Mathur. The Complexity of Model Checking. In M. Lyu, editor, *The Handbook of Software Engineering*, pages 531–550. McGraw-Hill, 1996.
- [8] H. Kautz and B. Selman. The Complexity of Propositional Logic. In *Proceedings of the 13th Conference on Artificial Intelligence and Applications*, Menlo Park, Aug. 4–5, 1991. Morgan Kaufmann Press. Available at <http://www.cba.hawaii.edu/~jimmyd/summaries/ai91.html>
- [9] A. Mackworth and E. Freuder. *Propositional Network Consistency: Algorithms and Complexity*. Addison-Wesley, 1985.
- [10] T. Menzies and B. Cukic. Software Testing. *IEEE Software*, 17(5):107–112, 2000. Available at <http://www.menzies.com/pdf/0005107.pdf>
- [11] T. Menzies, B. Cukic, H. S. He, and S. Edelkamp. Indeterminate Systems. In *Proceedings of the 10th International Conference on Software Engineering*, pages 10–19, 1998. Available at <http://tim.menzies.com/pdf/980610.pdf>
- [12] T. Menzies and Y. Hu. Agents in Software Engineering. In M. Lyu, editor, *Formal Approaches to Software Engineering*, chapter 2, pages 1–14, 2002. Available at <http://www.menzies.com/pdf/01agents.pdf>