Random Search of AND-OR Graphs Representing Finite-State Models

Tim Menzies
SE, ECE Depts.,
Univ. of British Columbia
Vancouver, BC, Canada
tim@menzies.com

David Owen
CSEE Dept.,
West Virginia Univ.
Morgantown, WV, USA
dowen@csee.wvu.edu

1 Introduction

One of the benefits of model-based reasoning is that automatic methods can assess that model. Such assessment is particularly important early in the software lifecycle when decisions are being made that will greatly affect the rest of the project.

Automatic formal methods can be impractically expensive. Formal tools that explore the entire state space of a program can require exponential time and memory. Despite years of research, this *state space explosion* problem remains. Model checkers that exploit the simplicity of the JAVA virtual machine [3] only work on implemented systems—not until long after the requirements phase has ended. Certain optimizations require expensive pre-processing, such as [9]. Also, these methods may rely on certain topological features of the system. For example, exploiting symmetry to avoid repeating prior searches (e.g. as done in [3]) is only useful if the system under study is highly symmetric. Also, clustering larger problems into smaller, simpler problems [2] fails for tightly connected models.

This paper proposes a method of *stochastic search* for counter-examples to temporal properties. The disadvantage of stochastic search is that randomly probing a theory can miss important features of that theory. This incompleteness of stochastic search makes it unpopular amongst software engineers; for example, "nondeterminacy is the enemy of reliability" [11].

If the incompleteness problem can be avoided, then the benefits of stochastic search are considerable. Stochastic search has proved effective for (e.g.) scheduling problems and has solved hard planning problems many times faster than traditional methods such as a systematic Davis-Putnam procedure [10]. One explanation for this surprising phenomenon is the *funnel theory* of Menzies, Easterbrook, et.al. [14]. The funnel theory is a claim that search spaces within programs contain *funnels*; i.e. a small number of critical variables that set all other variables (the metaphor here is that all arguments run down the funnel). The concept of such critical variables has been reported in many domains. These have been called a variety of names such as *master-variables* in scheduling [4]; *prime-implicants* in model-based diagnosis [16] or machine learning [17]; *backbones* in satisfiability [15]; and the *minimal environments* in the ATMS [5]. All of these terms share the same core intuition: what happens in the total space of a program is controlled by a small critical region within the program.

Theoretically, for systems with narrow funnels, the incompleteness of stochastic search may be minimal. *Any* stochastically selected pathway to goals must pass through the funnel (by definition). That is, repeated application of some fast random search technique will stumble across the funnel variables; i.e. stochastic search will be adequate for searching for counter examples to temporal properties.

Menzies et.al. show how a simple finite-state model written for the model checker SPIN [8] (see Figure 1) might be translated into a type of AND-OR graph called a NAYO [12]. The NAYO is small, but slow to search exhaustively—in some sense the opposite of finite-state models, which may be very large but faster to search exhaustively. It has been shown that a random (not exhaustive) search of a NAYO graph can quickly tell us a lot [12, 13], but this conclusion is based on randomly generated NAYO's. Would we learn as much from the random search of a NAYO translated from a finite-state model representing a real program?

In this paper we present an automatic translation from a simple finite-state model form to NAYO graphs. As an example, a program model written for the model checker SPIN is written in in this form and then automatically translated to a NAYO graph. A larger NAYO graph translated from a different finite-

state model, based on an SCR ("Software Cost Reduction") requirements specification, is subjected to a random search, with encouraging results.

2 Translating to NAYO

2.1 Finite State Model

The first step toward automatic translation is to formally define a simplified version of the finite-state model used by Promela, which is the input language used by the model checker SPIN. Consider a set S of finite-state machines:

1. Each finite-state machine $F \in S$ is a 3-tuple (Q, Σ, δ);
2. Q is a finite set of states;
3. Σ is a finite set of input/output symbols;
4. $\delta : Q \times B \longrightarrow Q \times B$, where B is a set of zero or more symbols from Σ, is the transition function.

```
byte a=1, b=1; bit f=1;
active proctype A() {
     do :: f==1 -> if :: a==1 -> a=2;
                      :: a==2 -> a=3;
                      :: a==3 -> f=0; a=1;
                   fi od}
active proctype B() {
     do :: f==0 -> if :: b==1 -> b=2;
                      :: b==2 -> b=3;
                      :: b==3 -> f=1; b=1;
                   fi od}
```

Figure 1: Promela input example (from [13]).

```
proctype A
     state 11 ...-> state  9 ...line 3 => ((f==1))
     state  9 ...-> state  3 ...line 3 => ((a==1))
     state  9 ...-> state  5 ...line 3 => ((a==2))
     state  9 ...-> state  7 ...line 3 => ((a==3))
     state  3 ...-> state 11 ...line 3 => a = 2
     state  5 ...-> state 11 ...line 4 => a = 3
     state  7 ...-> state  8 ...line 5 => f = 0
     state  8 ...-> state 11 ...line 5 => a = 1
proctype B
     state 11 ...-> state  9 ...line 8 => ((f==0))
     state  9 ...-> state  3 ...line 8 => ((b==1))
     state  9 ...-> state  5 ...line 8 => ((b==2))
     state  9 ...-> state  7 ...line 8 => ((b==3))
     state  3 ...-> state 11 ...line 8 => b = 2
     state  5 ...-> state 11 ...line 9 => b = 3
     state  7 ...-> state  8 ...line 10 => f = 1
     state  8 ...-> state 11 ...line 10 => b = 1
```

Figure 2: finite-state machines for Figure 1 as output by SPIN.

	From State	Input	Output	To State
	11	f==1		9
δ_A	9	a==1		3
	9	a==2		5
	9	a==3		7
	3		a=2	11
	5		a=3	11
	7		f=0	8
	8		a=1	11
	11	f==0		9
δ_B	9	b==1		3
	9	b==2		5
	9	b==3		7
	3		b=2	11
	5		b=3	11
	7		f=1	8
	8		b=1	11

Figure 3: transition function chart for Figure 2 SPIN output.

Figure 1 shows a simple Promela model of two concurrent processes with access to three global variables. Promela has been designed to look like a high-level computer language, but is actually used to write finite-state models. Figure 2 shows SPIN's finite-state model interpretation of the code in 1, and Figure 3 shows a chart with the transition functions for the finite-state machines output by SPIN. In terms of the formal description above:

$S = \{F_A, F_B\}$;

$F_A = (Q_A, \Sigma_A, \delta_A)$;

$Q_A = \{$`11,9,3,5,7,8`$\}$;

$\Sigma_A = \{$ `f==1,a==1,...a=2,a=3,...` $\}$, etc.

2.2 NAYO

A NAYO graph $G = (N, A, Y, O)$ [12] consists of:

1. A set Y of directed YES-edges;
2. A set O of OR-nodes—an OR-node is TRUE if any of its YES-edge parents are TRUE;
3. A set A of AND-nodes—an AND-node is TRUE if all of its YES-edge parents are TRUE;
4. A set N of undirected NO-edges connecting incompatible nodes.

In the NAYO graph representing a finite-state model we do distinguish between the equality test ("==") and assignment ("=") operators. For each value taken by a particular variable there is one node assigned by YES-edge parents and tested by YES-edge children; it is connected by NO-edges to all nodes representing other values for that variable. Because

of this, the finite-state model for `proctype A` (from Figure 3) can be simplified to Figure 4, which shows `proctype A` as input to an automatic NAYO translator. Here each finite-state machine description begins with a list of its states; then transitions are listed (current state, input, output, next state). Global variables `a` and `f` are represented by machines without any transitions, but with states accessible to the other machine.

```
begin A                 begin a
11;                     "a=1";
9;                      "a=2";
3;                      "a=3";
5;                      end a
7;
8;                      begin f
11; "f=1"; -; 9;        "f=0;"
9; "a=1"; -; 3;         "f=1;"
9; "a=2"; -; 5;         end f
9; "a=3"; -; 7;
3; -; "a=2"; 11;
5; -; "a=3"; 11;
7; -; "f=0"; 8;
8; -; "a=1"; 11;
end A
```

Figure 4: `proctype A` and variables `a`, `f` from Figure 2 as input for NAYO translator.

2.3 Translation Algorithm

The following algorithm translates from the finite-state machine form shown in Figure 4 to a NAYO graph:

For each finite-state machine:

1. Make an OR-node for each state, and connect all of this machine's OR-nodes with NO-edges;
2. For each transition:
   i. If input is listed make an AND-node, make the current state and the input(s) YES-edge parents of the AND-node, and make the next state and any output(s) YES-edge children of the AND-node;
   ii. If no input listed make the next state and any output(s) YES-edge children of the current state.

Figure 5 shows the NAYO graph produced by this algorithm from the input shown in Figure 4. For a system of k finite-state machines with n states and m single-input, single-output transitions per machine, the resulting NAYO has:

$m \cdot k$ AND-nodes + $n \cdot k$ OR-nodes = $O(m+n)k$ nodes;

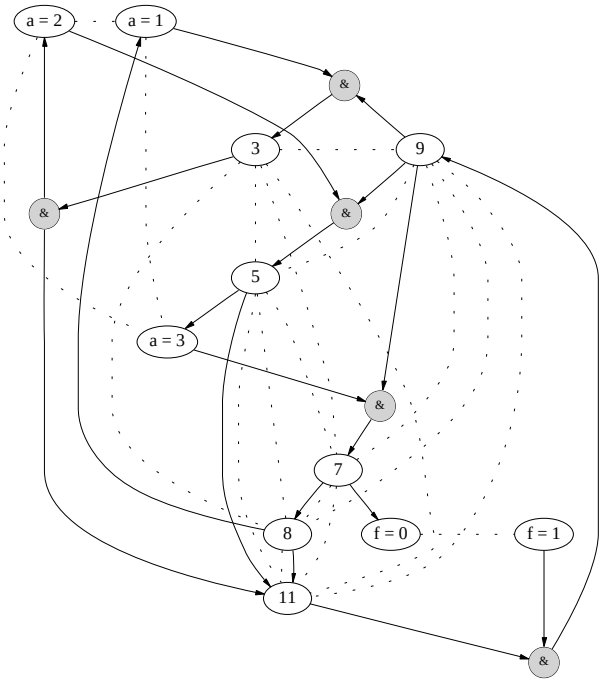

Figure 5: finite-state model from Figure 4 as NAYO (NO-edges are dotted; AND-nodes are shaded).

$4m \cdot k$ YES-edges + $\frac{1}{2}n(n-1) \cdot k$ NO-edges = $O(m + n^2)k$ edges.

A finite-state machine composite for the same system will in the worst case require $O(n^k)$ states and $O(mk^2)$ transitions [7].

3 Random NAYO Search

The NAYO graph search begins with a random input set of OR-nodes, consistent with each other (no NO-edges between any nodes in this set) and TRUE at time = 0. The search expands this set to include all consequences of the input, including a *frontier* of nodes inconsistent with the input. The frontier set is the input for the next time value. The search progresses through the graph, finding a set of consistent nodes TRUE at each time value, until a time is reached at which no new nodes are found.

Figure 6 shows the result of repeated random searches of a 73-node, 139-edge NAYO graph, translated from the SCR requirements specification in [1]. Like Promela, the SCR notation (see [6]) is based on finite-state machines, and can easily be written in the form of Figure 4. The plot in Figure 6 rises early to a plateau of reached nodes, suggesting that this is a sys-

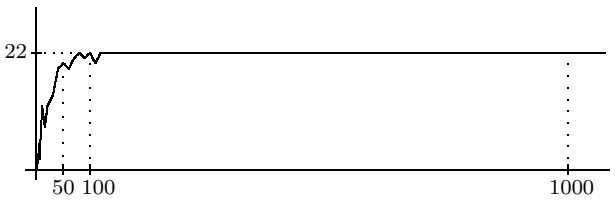

Figure 6: results for search of SCR specification example; horizontal axis: number of random searches, vertical axis: number of goals reached.

tem with a *funnel* of key variables, as described in the introduction. A small number of of random searches (in this case approximately equal to the number of nodes in the graph) finds the funnel, and therefore finds everything that will be found by a much larger number of searches.

4 Conclusion

The finite-state model form used in this paper (Figure 4) for automatic translation to NAYO graphs is too simple to represent many systems. It cannot handle communication protocols, which require a scheme by which messages may be sent and received and should be consumed by the receiver, so that they are no longer available to be received by another. Also, variables require an entry for every possible value, so floating point or even integer variables are out of the question. But for all its shortcomings this simple finite-state model can represent many real systems, and our preliminary results with one of these produced just the *funnel* behavior expected. Based on these encouraging results we suggest more thorough testing of NAYO random search, working from an extension of our finite-state model, which would include a message-passing capability and would be able to handle variables more efficiently.

References

[1] B. Atanacio. *Modeling the Space Shuttle Liquid Hydrogen Subsystem*, available from`http://www.sei.cum.edu/publications/documents/00.reports/00tn002.html`. SEI, Carnegie Mellon University, 2001.

[2] E.M. Clark and D. E. Long. Compositional Model Checking. *Fourth Annual Symposium on Logic in Computer Science*, 1989.

[3] J. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasarenu, Robby and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. *ICSE2000, Limerick, Ireland*, 2000, pp. 439-448.

[4] J. Crawford and A. Baker. Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. *AAAI '94*, 1994.

[5] J. DeKleer. An Assumption-Based TMS. *Artificial Intelligence*, Vol. 28, 1986, pp. 163-196.

[6] C. Heitmeyer, B. Labaw and D. Kiskis. Consistency Checking of SCR-Style Requirements Specifications. *International Symposium on Requirements Engineering, York, England*, March 26-27, 1995.

[7] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[8] G. Holzmann. The Model Checker SPIN. *IEEE Transactions of Software Engineering*, 23(5), May 1997.

[9] Y. Ishida. Using Global Properties for Qualitative Reasoning: A Qualitative System Theory. *Proceedings of IJCAI '89*, 1989, pp. 1174-1179.

[10] H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic and Stochastic Search. *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, available from `http://www.cc.gatech.edu/~jimmyd/summaries/kautz1996.ps`, AAAI PRess / MIT Press, Menlo Park, 1996, pp. 1194-1201.

[11] N. Leveson. *Safeware System Safety and Computers*. Addison-Wesley, 1995.

[12] T. Menzies, B. Cukic, H. Singh and J. Powell. Testing Nondeterminate Systems. *ISSRE 2000, San Jose, California*, October 8-11, 2000.

[13] T. Menzies and B. Cukic. Maintaining Maintainability = Recognizing Reachability. *WESS 2000, San Jose, California*, October 14, 2000.

[14] T. Menzies, S. Easterbrook, Bashar Nuseibeh and Sam Waugh. An Empirical Investigation of Multiple Viewpoint Reasoning in Requirements Engineering. *RE '99*, available from `http://tim.menzies.com/pdf/99re.pdf`, 1999.

[15] A. Parkes. *Lifted Search Engines for Satisfiability*. PhD Thesis, University of Oregon, available from `http://citeseer.nj.nec.com/parkes99lifted.html`, 1999.

[16] R. Rymon. An SE-tree-based Prime Implicant Generation Algorithm. *Annals of Math. and A.T., Special Issue on Model-Based Diagnosis*, available from `http://citeseer.nj.nec.com/193704.html`, Vol. 11, Console & Friedrich, Ed., 1994.

[17] R. Rymon. An SE-tree based Characterization of the Induction Problem. *International Conference on Machine Learning*, 1993, pp. 268-275.