Deterministic Compilation of Temporal Safety Properties in Explicit State Model Checking

Kristin Y. Rozier and Moshe Y. Vardi
Rice University

November 8, 2012

Model Checking

Model Checking:

1. Create a system model with formal semantics, M.
2. Encapsulate desired properties in a formal specification, f.
3. Check that M satisfies f.

Model checking finds disagreements between
the system model and the formal specification.

Model Checking

Model Checking:

1. Create a system model with formal semantics, M.
2. Encapsulate desired properties in a formal specification, f.
3. Check that M satisfies f.

Model checking finds disagreements between
the system model and the formal specification.

Successful industrial adoption!

Model Checking

Model Checking:

1. Create a system model with formal semantics, M.
2. Encapsulate desired properties in a formal specification, f.
3. Check that M satisfies f.

Model checking finds disagreements between
the system model and the formal specification.

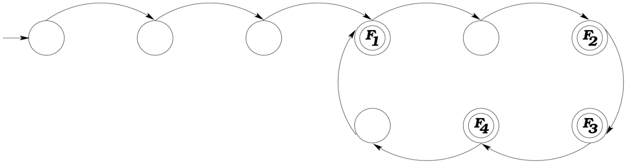
Successful industrial adoption!

NASA uses the *explicit state* **Spin Model Checker** for analysis of aerospace systems.

How Is Model Checking Implemented?

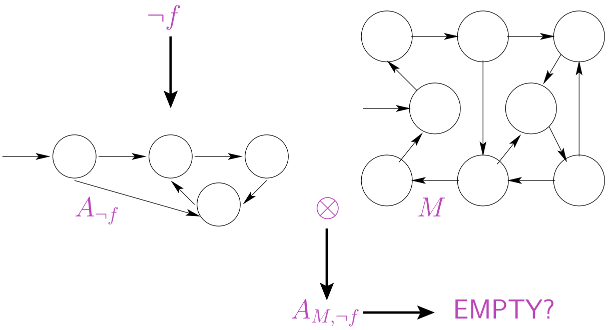
Explicit Model Checkers:

- Construct explicit automaton for specification.
- Search explicitly for a trace falsifying the specification.
  - Look for an accepting run of the property automaton.
  - Look for an accepting lasso by finding strongly connected components in the model/automaton graph.

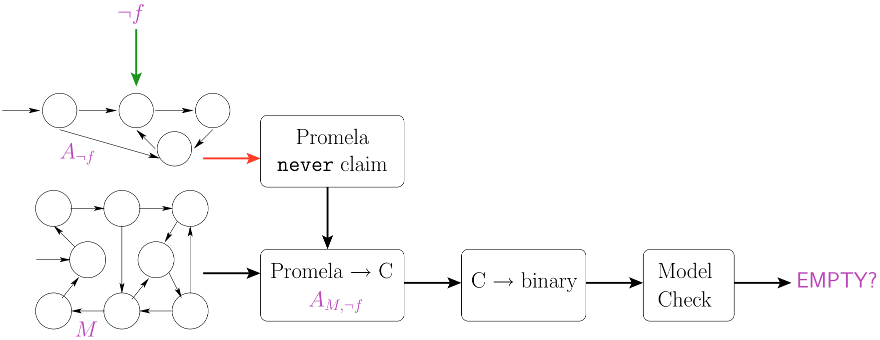


accepting lasso = counterexample trace

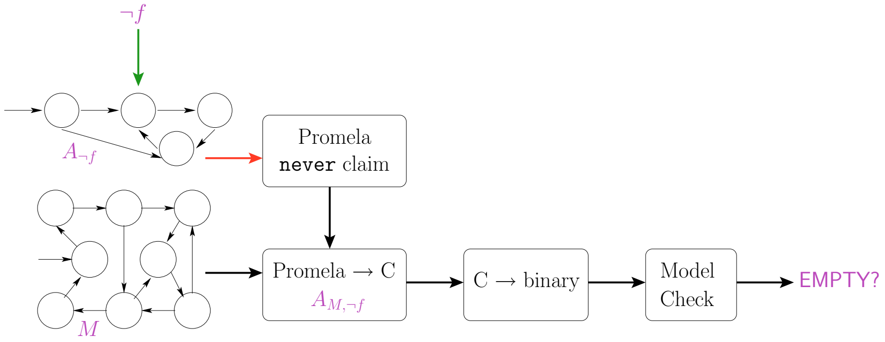
Automata-Theoretic Approach to Model Checking



Explicit Model Checking With Spin



Explicit Model Checking With Spin



We are the first to measure these compilation and model checking stages separately

LTL-to-Automaton Complexity

- LTL property of size m
- Model of size n
- LTL model checking takes time $n \cdot 2^{O(m)}$.

LTL-to-automata translation has dramatic impact on model checking.

- *heavily* studied

Promela `never` claims for Spin Model Checker:

- *hardly* studied

The encoding of $A_{\neg f}$ as a `never` claim has a major impact on complexity.

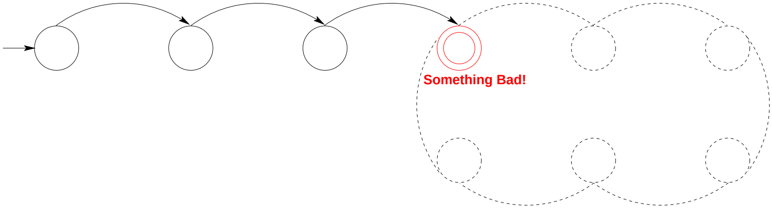
Related Work: Many Ways of Generating $A_{\neg f}$

- LTL2AUT .... (Daniele, Guinchiglia, Vardi)
  Implementations (Java, Perl) .... LTL2Buchi, Wring
- LTL2BA (C) .... (Oddoux, Gastin)
- LTL2Buchi (Java) .... (Giannakopoulou, Lerda)
- LTL $\rightarrow$ NBA (Python) .... (Fritz, Teegen)
- Modella (C) .... (Sebastiani, Tonetta)
- Spot (C) .... (Duret-Lutz, Poitrenaud, Rebiha, Baarir, Martinez)
- TMP (SML of NJ) .... (Etessami)
- Wring (Perl) .... (Somenzi, Bloem)

All of these produce nondeterministic automata for *general* LTL formulas.

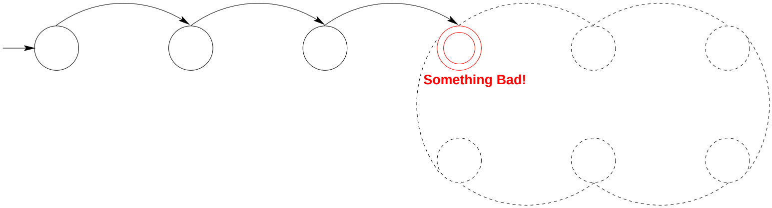
Model Checking Safety Properties

Safety: "something bad never happens"
(ALWAYS $\neg$*something_bad*)



Model Checking Safety Properties

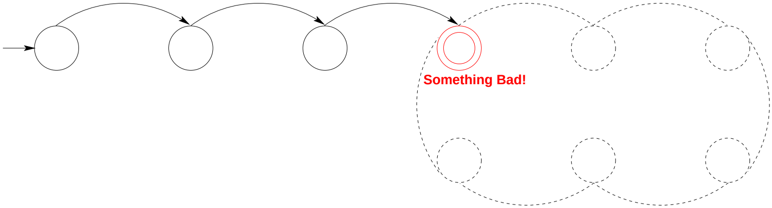
Safety: "something bad never happens"
(ALWAYS $\neg$*something_bad*)



Search for a bad prefix.

Model Checking Safety Properties

Safety: "something bad never happens"
(ALWAYS $\neg$*something_bad*)

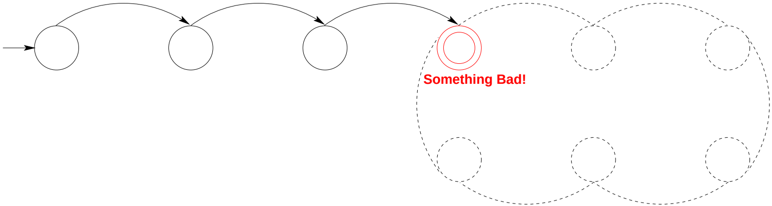


Search for a bad prefix.

We don't need the rest of the lasso!

Model Checking Safety Properties

Safety: "something bad never happens"
(ALWAYS $\neg$*something_bad*)



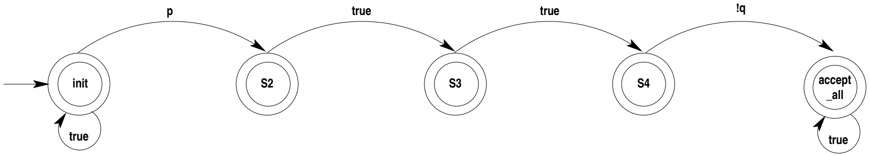
Search for a bad prefix.

We don't need the rest of the lasso!

We can form deterministic automata on finite words!

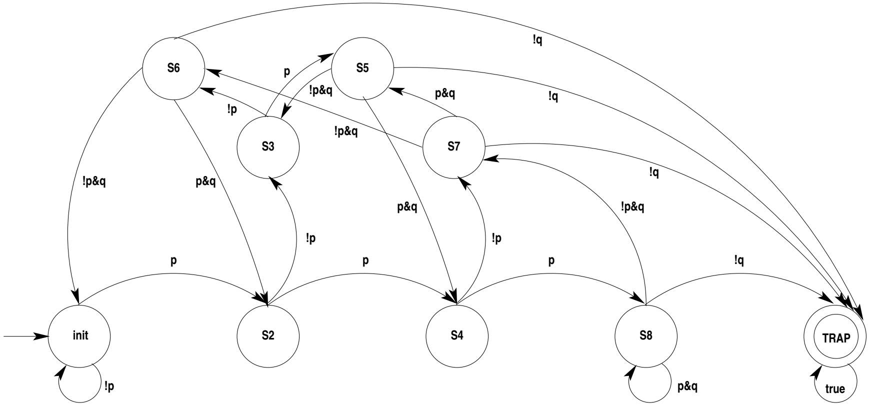
A Nondeterministic Property Automaton

$!(\text{ALWAYS}(\text{XXX } q \mid !p)) = \text{EVENTUALLY}(p \ \& \ \text{XXX } !q)$



A Deterministic Property Automaton

EVENTUALLY(p & XXX !q)



Determinism in Model Checking

- When the automaton is nondeterministic, the model checker has to find paths in both the system and the property automaton.
- When the automaton is deterministic, the model checker has to find a path only in the system.
- We do one search instead of two!
- This may increase model checking scalability!

Determinism in Model Checking

- When the automaton is nondeterministic, the model checker has to find paths in both the system and the property automaton.
- When the automaton is deterministic, the model checker has to find a path only in the system.
- We do one search instead of two!
- This may increase model checking scalability!

Safety properties are 90% of specifications!

Determinism in Model Checking

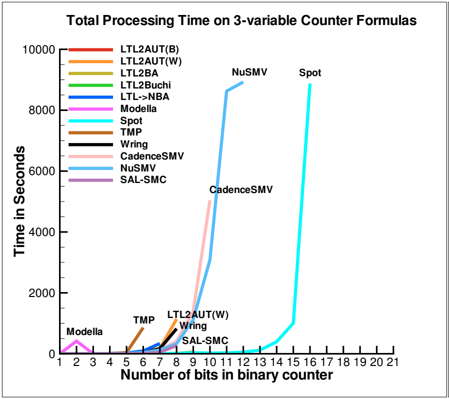
- When the automaton is nondeterministic, the model checker has to find paths in both the system and the property automaton.
- When the automaton is deterministic, the model checker has to find a path only in the system.
- We do one search instead of two!
- This may increase model checking scalability!

Safety properties are 90% of specifications!

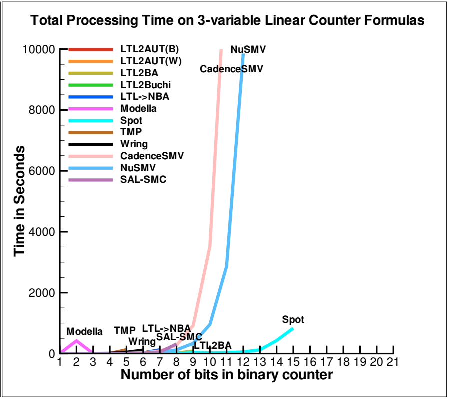
Only one tool: scheck[1] = buggy

[1] T. Latvala. Efficient model checking of safety properties. In SPIN, pages 74-88, 2003.

SPOT is the Only Industrial Quality Explicit-State Tool[2]



Conjunction of $\mathcal{X}$-subformulas.



Linearly nested $\mathcal{X}$-operators.

[2] Rozier, Kristin Y., and Vardi, Moshe Y. "LTL Satisfiability Checking." In *International Journal on Software Tools for Technology Transfer (STTT)*, Springer-Verlag, March, 2010.

Can We Now Improve Explicit Encodings?

Can we improve upon the SPOT encoding for safety formulas?

Can We Now Improve Explicit Encodings?

Can we improve upon the SPOT encoding for safety formulas?

Can new encodings for explicit automata improve model checking performance?

Can We Now Improve Explicit Encodings?

Can we improve upon the SPOT encoding for safety formulas?

Can new encodings for explicit automata improve model checking performance?

Can we exploit determinism to improve our `never` claims?

Can We Now Improve Explicit Encodings?

Can we improve upon the SPOT encoding for safety formulas?

Can new encodings for explicit automata improve model checking performance?

Can we exploit determinism to improve our `never` claims?

YES!

Encoding Safety Formulas Deterministically

We form a `never` claim for $\neg\phi$ from ϕ:

1. SPOT: $\phi \rightarrow$ Nondeterministic Büchi Automaton (NBW) A_ϕ
2. SPOT: compute $empty(A_\phi)$ & remove from A_ϕ
3. relabel remaining states *accepting* $\longrightarrow$ Nondeterministic Finite Automaton (NFW) A^f_ϕ
4. determinize with subset construction $\longrightarrow A^d_\phi$
5. complement: only the empty set of states is now accepting $\longrightarrow A^d_{\neg\phi}$
6. translate deterministic automaton into `never` claim

Many different ways to perform the last three steps . . .

A Set of 26 Promela Never Claim Encodings

Our novel encodings are combinations of seven components:

1. Determinization: beforehand[3] (det) or on-the-fly (nondet)
2. Transitions: looking forward (front) or backward (back)
    3. Encoding: front_nondet, back_nondet, back_det, front_det_switch, front_det_memory_table
4. State Minimization: min or nomin
    5. Alphabet Representation (for minimization): BDDs or assignments or assignments with edge abbreviation
6. State Representation: state numbers or state labels
7. Acceptance: finite or infinite

Winning Encoding: front_det_switch_min_abr_ea_state_fin

[3] with BRICS Automaton

Encoding Forms and Determinization

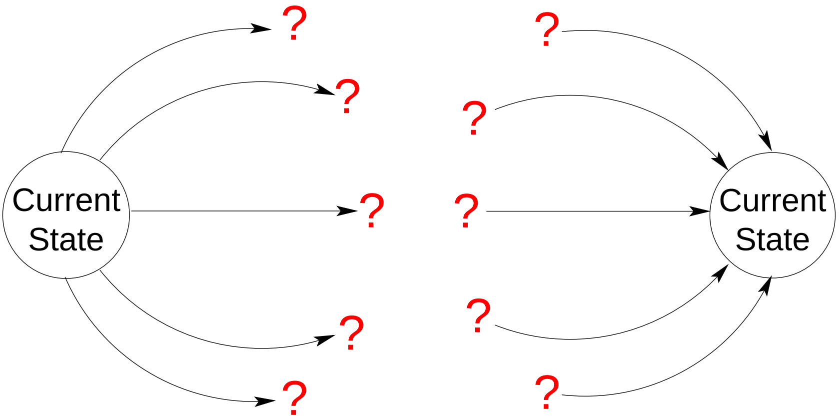
```
never {

  ...

S1:
  atomic {

    if
      :: (!p2)
        -> goto done;

      :: ((!p0 && p2)
          || (!p1 && p2))
        -> goto S1;
    fi;

  }
  ...
```

front_det_state

```
never {
  do   :: atomic{
    /*Swap current_state and next_state: */
    /* do current_state[i] = next_state[i]; i++;*/
    /*Reset next_state: next_state[i] = 0*/
    ...
    if   /*Fill in next_state array*/
    :: current_state[1] ->
         if :: (p0 && p1 && p2  )
                 -> next_state[0] = 1;
           :: else -> skip;
         fi;
         if :: ((!p0 && p2) || (!p1 && p2)  )
                 -> next_state[1] = 1;
           :: else -> skip;
         fi;
      :: else -> skip;
    fi;
  ...
```

front_nondet_number

Determinization On-the-fly: Forward vs Backward



front_nondet encoding

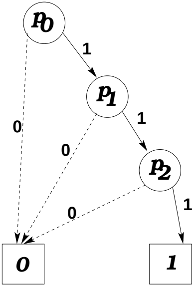
back_nondet encoding

State Minimization and Alphabet Representation

Example transition label: $(p_0 \& p_1 \& p_2)$

Integer label i: $0 \leq i < 2^n$

$I(\mathbf{p}) = p_0 2^{n-1} + p_1 2^{n-2} + \ldots + p_n 2^0$



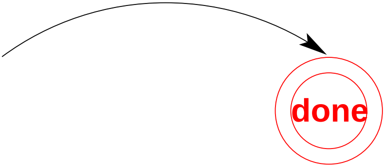
BDD-based Representation

$$\frac{1}{p_0} \quad \frac{1}{p_1} \quad \frac{1}{p_2} \quad = 7$$

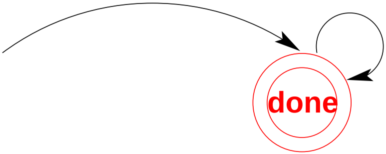
Assignment-based Representation

State Representation and Acceptance Conditions

```
never {
   ...
   if (property is violated)
      -> goto done;
   ...

done:
   skip;
}
```

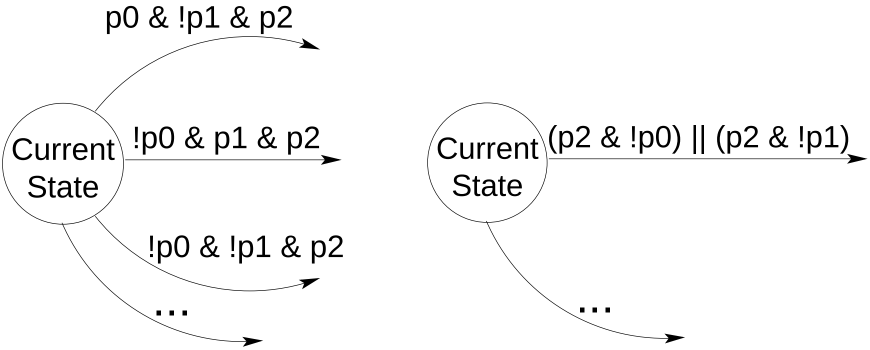


finite acceptance:
proceed to }



infinite acceptance:
loop

Edge Abbreviation for Determinized Encodings



Transitions Without Edge Abbreviation

Transition With Edge Abbreviation

26 Combinations

State Minimization	Alphabet Representation	Automaton Acceptance	Monitor Encoding	State Representation
no	BDDs	finite infinite	front_nondet back_nondet	number
yes	assignments		front_nondet back_nondet back_det front_det_memory_table	
			front_det_switch	state/number
	assignments+edge abbreviation		back_det	number

Extensive Empirical Evaluation

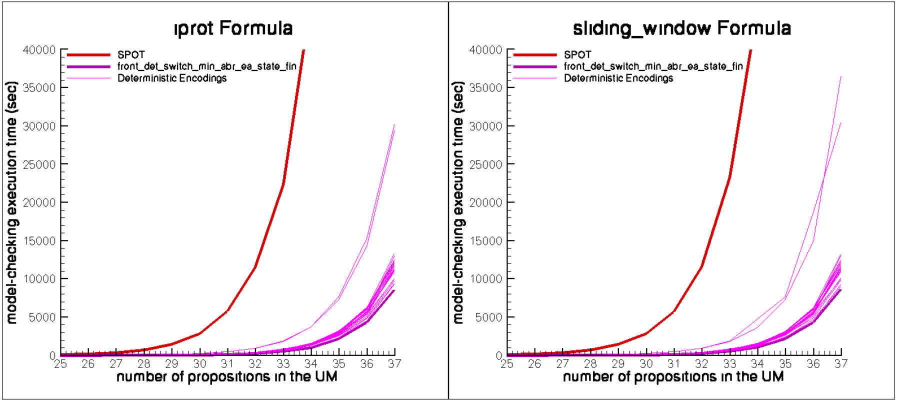
- Model-Scaling Benchmarks
  - 14 real-life safety formulas
  - Scaled universal models

- Formula-Scaling Benchmarks
  - Two classes of randomly-generated safety formulas
    - Tested for safety
    - Syntactically safe
  - One large universal model

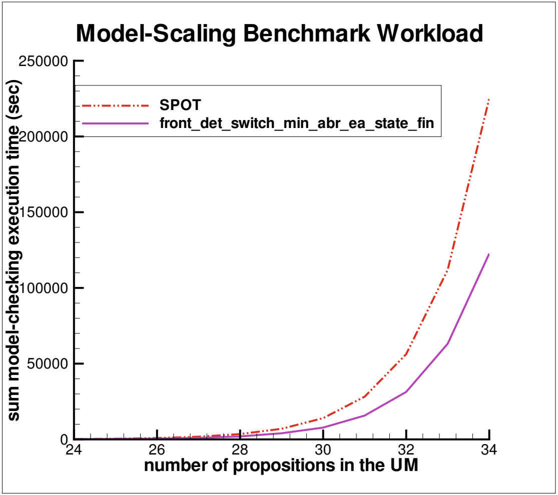
Experimental Results

1. We consistantly beat SPOT in model checking time
2. One of our encodings is always best: front_det_switch_min_abr_ea_state_fin
3. There seems to be a partial order on the performance of our encodings:
    - Deterministic automata are faster than nondeterministic
    - Determinization up front is faster than on-the-fly
    - Finite acceptance is faster than infinite acceptance
    - State labels are faster than state numbers
    - Switch-statement format is best
    - State minimization and edge abbreviation lead to better performance

Sometimes Deterministic Automata Are *Much* Better

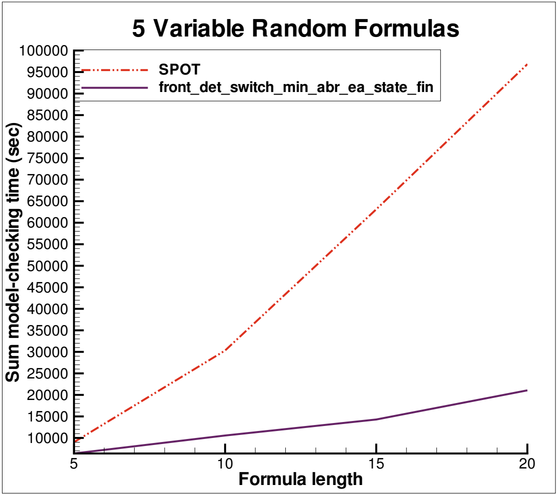


Model-Checking Performance for Industrial Specifications



- Workload: 14 industrial specifications.
- Across the whole benchmark suite, we have a factor of $\sim 2x$ performance in MC time.

Formula-Scaling Performance for Random Specifications



- ∼ 300 formulas
- factor of 5 speedup

Discussion

- Deterministic encodings are faster than nondeterministic encodings.
- One deterministic encoding is always best:
  front_det_switch_min_abr_ea_state_fin.

Winning encoding implemented in open-source **CHIMP-Spin** tool!

Recommend CHIMP-Spin for *safety* formulas; SPOT for all others.

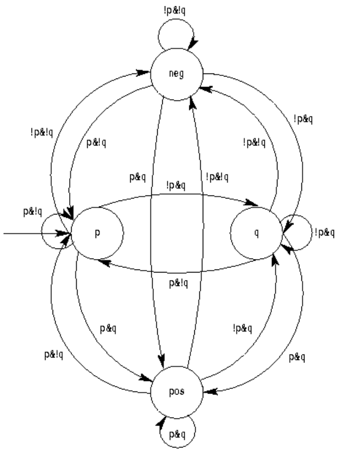
BACKUP SLIDES

Spin's Nested Depth First Search Algorithm

```
proc dfs(s)
   if error(s) then report error fi
   add {s,0} to Statespace
   add s to Stack
   for each (selected) successor t of s do
      if {t,0} not in Statespace then dfs(t) fi
   od
   if accepting(s) then ndfs(s) fi
   delete s from Stack
end
proc ndfs(s) /* the nested search */
   add {s,1} to Statespace
   for each (selected) successor t of s do
      if {t,1} not in Statespace then ndfs(t) fi
      else if t in Stack then report cycle fi
   od
end
```

Visualization of a Universal Model:

A State-Labeled Universal Model with 2 Propositions



Model-Scaling Benchmarks[4]

0	$\Box\neg bad$	"Something bad never happens."
1	$\Box(request \rightarrow \mathcal{X} grant)$	"Every request is immediately followed by a grant"
2	$\Box(\neg(p \wedge q))$	Mutual Exclusion: "p and q can never happen at the same time."
3	$\Box(p \rightarrow (\mathcal{X}\mathcal{X}\mathcal{X} q))$	"Always, p implies q will happen 3 time steps from now."
4*	$\mathcal{X}((p \wedge q)\mathcal{R} r)$	"Condition r must stay on until buttons p and q are pressed at the same time."
5*	$\mathcal{X}(\Box(p))$	slightly modified *intentionally safe* formula from KV99c
6	$\Box(q \vee \mathcal{X}\Box p) \wedge \Box(r \vee \mathcal{X}\Box\neg p)$	*accidentally safe* formula from KV99c
7*	$\mathcal{X}([\Box(q \vee \Diamond\Box p) \wedge \Box(r \vee \Diamond\Box\neg p)] \vee \Box q \vee \Box r)$	slightly modified *pathologically safe* formula from KV99c
8	$\Box(p \rightarrow (q \wedge \mathcal{X} q \wedge \mathcal{X}\mathcal{X} q))$	safety specification from TRV11
9	$(((((p0\mathcal{R}(\neg p1))\mathcal{R}(\neg p2))\mathcal{R}(\neg p3))\mathcal{R}(\neg p4))\mathcal{R}(\neg p5))$	Sieve of Erathostenes
10	$(\Box((p0 \wedge \neg p1) \rightarrow (\Box\neg p1 \vee (\neg p1 \mathcal{U}(p10 \wedge \neg p1)))))$	G.L. Peterson's algorithm for mutual exclusion algorithm
11	$(\Box(\neg p0 \rightarrow ((\neg p1 \mathcal{U} p0) \vee \Box\neg p1)))$	CORBA General Inter-Orb Protocol
12	$((\Box(p1 \rightarrow \Box(\neg p1 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box(p2 \rightarrow \Box(\neg p2 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box\neg p2 \vee (\neg p2 \mathcal{U} p1)))$	GNU i-protocol, also called iprot
13	$((\Box(p1 \rightarrow \Box(\neg p1 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box(p2 \rightarrow \Box(\neg p2 \rightarrow (\neg p0 \wedge \neg p1)))) \wedge (\Box\neg p2 \vee (\neg p2 \mathcal{U} p1)))$	Sliding Window protocol

4 Starred formulas are checked against a universal model that sets all variables to *true* first.