ACM Transactions on Software Engineering and Methodology, 1998.

Model Checking Large Software Specification

William Chan,
Richard J. Anderson,
Paul Beame,
Steve Burns,
Francesmary Modugno,
David Notkin,
Jon D. Reese

JUNBEOM YOO

Dependable Software Laboratory
KONKUK University

http://dslab.konkuk.ac.kr

2010.11.09

Abstract

In this paper, we present our experiences in using symbolic model checking to analyze a specification of a software system for aircraft collision avoidance. Symbolic model checking has been highly successful when applied to hardware systems. We are interested in whether model checking can be effectively applied to large software specifications. To investigate this, we translated a portion of the state-based system requirements specification of Traffic Alert and Collision Avoidance System II (TCAS II) into input to a symbolic model checker (SMV). We successfully used the symbolic model checker to analyze a number of properties of the system. We report on our experiences, describing our approach to translating the specification to the SMV language, explaining our methods for achieving acceptable performance, and giving a summary of the properties analyzed. Based on our experiences, we discuss the possibility of using model checking to aid specification development by iteratively applying the technique early in the development cycle. We consider the paper to be a data point for optimism about the potential for more widespread application of model checking to software systems.

Contents

1. Introduction
2. Model Checking
3. Translation Basics
4. Translation Rules
5. Obstacles
6. Results of Analysis
7. Related Work
8. Discussion
9. Conclusion

1. Introduction

- How can we increase our confidence in the specifications, particularly those of safety-critical systems?

- Formal methods offer opportunities for mechanical verification, but most existing techniques either do not scale to large systems, require extensive human guidance, or are limited to verifying simple (though important) properties like deadlock freedom, consistency, and completeness.

- Symbolic model checking [15] based on binary decision diagrams (BDDs) [10] is an efficient automatic verification technique that is simultaneously capable of scaling and of verifying a wide range of properties.
  - It has been applied successfully to many industry-scale hardware circuits, but not aggressively to the analysis of software specifications.

1. Introduction

- In this paper, we describe an experience in analyzing a large system requirements specification using symbolic model checking.
- In our experiment, we translated a significant portion of a preliminary version of the Traffic Alert and Collision Avoidance System II (TCAS II) System Requirements Specification from the Requirements State Machine Language (RSML) into input to the Symbolic Model Verifier (SMV).
- We were able to control the size of the BDDs representing the RSML specification so that we could analyze a number of properties.
  - Robustness properties
  - Safety-critical properties specific to the domain

1. Introduction

- Our objective was to test the effectiveness of model checking on software systems with the hope that most or all of these techniques are applicable to other situations.
- We stress two approaches that we found crucial in overcoming the complexity and size of the specification,
  - the use of nondeterministic modeling primarily to abstract nonlinear arithmetic and to allow checking part of the specification
  - the use of an iterative process to analyze the specification
- We also point out some limitations of the current model checking techniques and tools, and suggest some future research directions.

2. Model Checking

- Model checking is a formal verification technique based on state exploration
  - Given a state transition system and a property, model checking algorithms exhaustively explore the state space to determine whether the system satisfies the property.

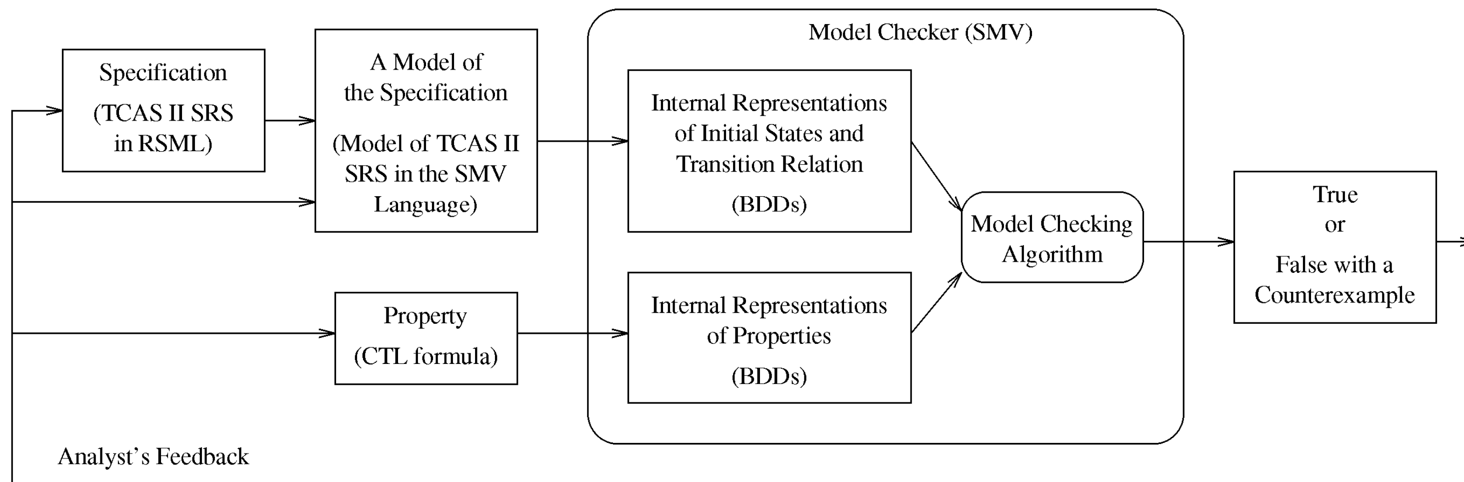

Fig. 1. Model-checking a specification.

2.1 The CTL Model Checking Problem

- In temporal-logic model checking, we are given a state transition system, which models a software or hardware system, and a property specified as a formula in a certain temporal logic, and determine whether the system satisfies the formula.

- A common logic for model checking is the branching-time Computation Tree Logic (CTL).
  - **AG *safe*** : All reachable states are safe.
  - **AG AF *stable*** : The system is stable infinitely often.
  - **AG (*request ® AF response)*** : A request is always followed by a *response* sometime in the future.
  - **AG EF *restart*** : It is possible to restart the system in any reachable state.

2.1 The CTL Model Checking Problem

- Formally, a state transition system $< Q, R, I >$ consists of a set of states Q, a state transition relation $R \subseteq Q \times Q$, and a set of initial states $I \subseteq Q$.
- The set of states Q is often encoded by a set of state variables, such that each state corresponds to some valuation for the variables and no distinct states correspond to the same valuation.
- The system satisfies a formula if the formula holds at all initial states. If not, a model checker typically attempts to find a counterexample.

2.2 Symbolic Model Checking and BDDs

- In explicit model-checking techniques, the truth value of a CTL formula is determined in a graph-theoretic manner by traversing the state diagram, with time complexity linear in the size of the state space and in the length of the formula.

  → State explosion problem

- Symbolic techniques: Instead of visiting individual states as in conventional state space search, symbolic model checkers visit a set of states at a time.

2.2 Symbolic Model Checking and BDDs

- When the state space is finite, we can assume without loss of generality that the state variables are boolean and there are only finitely many of them.
- A predicate on these variables is simply a boolean function, which can be represented by reduced ordered binary decision diagrams(BDDs).

- A number of BDD-based symbolic model checkers have een built, mainly for hardware circuit verification.

2.3 SMV

- SMV is a CTL symbolic model checker using BDDs to represent state sets and transition relations.
  - An SMV program consists of the description of a finite state transition system and a list of CTL formulas.

```
VAR
  b: boolean;
  x: 0..7;
  s: {on, off};
```

```
ASSIGN
  init(b) := 0;
```

```
ASSIGN
  next(b) := !b;
```

```
ASSIGN
  next(x) := case
               x < 7: x + 1;
               1: 0;
             esac;
```

→ **modulo-8 counter**

```
DEFINE
  d := x = 7 & b = 0;
ASSIGN
  next(s) := case
               d: on;
               1: off;
             esac;
```

→ **macro**

```
TRANS
  (d & next(s)= on) | (!d & next(s) = off)
```

2.3 SMV

- Two sources of nondeterminism in SMV are relevant to us.

1. An expression can be a set, and it nondeterministically valuates to a value from that set.

```
ASSIGN
  init(x) := {0, 1};
```

2. when the initial or the next-state value of a variable is not specified, it nondeterministically evaluates to a value of its type.

3. Translation Basics

- Section 3.1 gives an informal overview of RSML,
- Section 3.2 provides intuition of the translation from RSML to SMV by showing an example
- Section 4 describes general translation rules.

3.1 RSML Overview

- RSML is a state-machine language based on statecharts, extending conventional state diagrams with state hierarchies and broadcast communications.

3.1.1 State Hierarchy

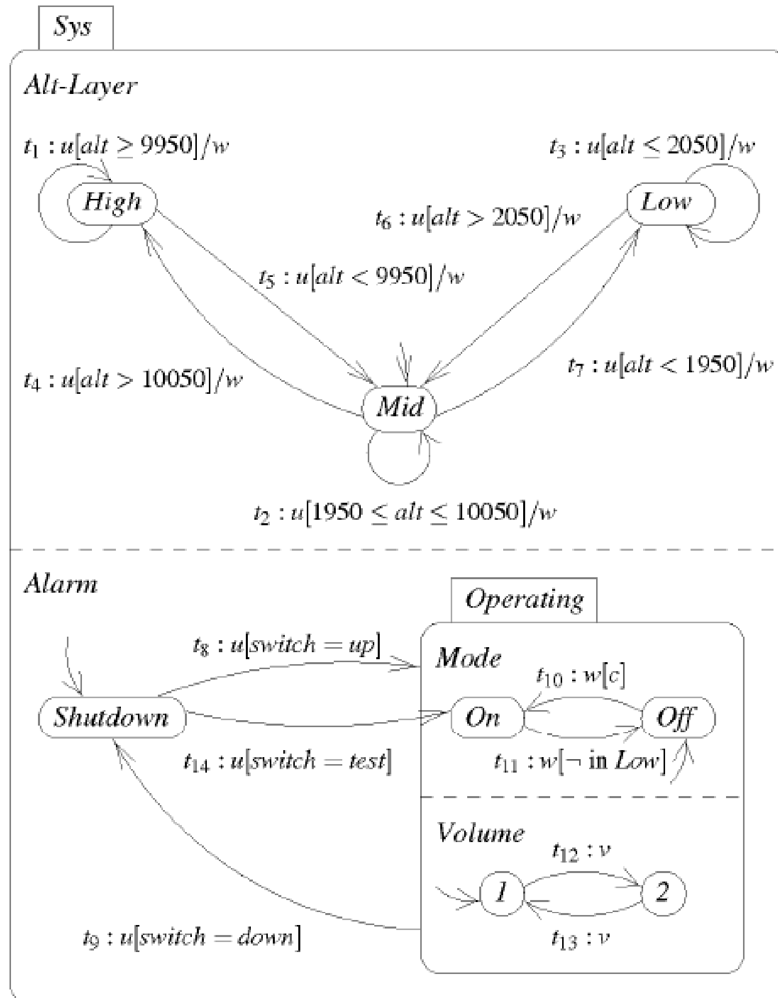

Fig. 2. An example of an RSML machine.

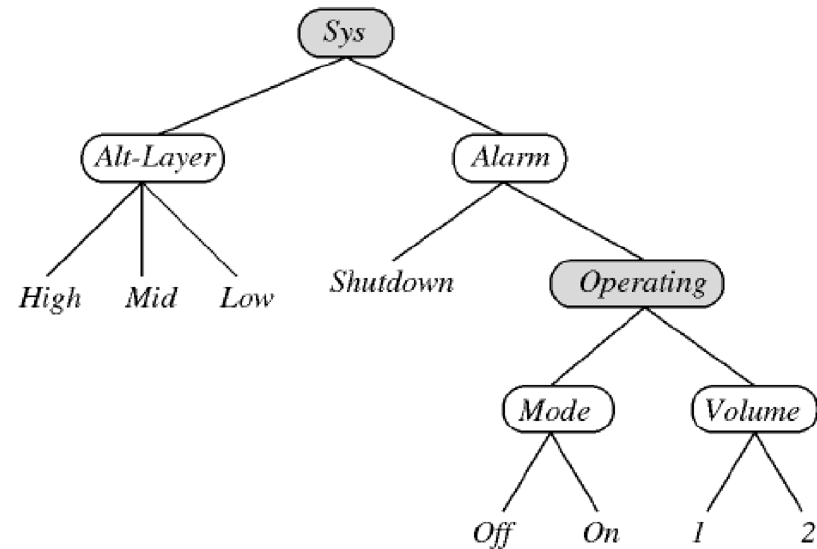

Fig. 3. The state hierarchy drawn as a tree. The shaded nodes represent and-states, unshaded nodes represent or-states, and the leaves are atomic states.

3.1.2 Inputs and Events

- The example contains two input variables from the environment, namely *alt* (an integer) and *switch* (up, down, or test).
  - The input *alt* represents the altitude of the aircraft, and *switch* is controlled by the pilot.

- States in RSML are synchronized by events, which are broadcast to the entire system.
  - *u*, *v* : generated by the environment and are called ***external events***
  - *w* : generated by the machine for ***internal synchronization*** (in this example only)

3.1.3 Transitions

- A transition is represented as an arrow originating from a *source* state to a *destination* state.

$$id:\ trig[cond]\,/\,acts$$

- The idea is that if the machine is in the source state, the trigger occurs, and the guarding condition is true (it is considered true if absent), then the transition is enabled.

- Synchrony Hypothesis
  - *External events → cascading of microsteps → becomes stable → a step*
  - During a step, no new external event may occur and the values of the inputs remain unchanged.
  - In other words, the machine runs infinitely faster than the environment. Once the machine is stable, inputs can change and external events can again occur.

3.1.4 AND/OR Tables

- The guarding condition *c* of transition t_{10}, too complex to fit in Fig. 2, is shown in Fig. 4 as an AND/OR table, one of the features that distinguish RSML.

Transition(s): *Off* ⟶ *On*

Location: *Mode*

Trigger Event: w

Condition:

AND		OR		
Alt-Layer **in state** Low	T	T	T
$alt < 1000$	T	.	.
$alt < 1500$	.	T	.
$\text{PREV}(alt) < 1500$	.	T	.
$t \geq t(\text{Exited}(Mid)) + 5$	.	.	T

Output Action:

Fig. 4. Transition from *Off* to *On*.

3.2 Translating the Example

- In Section 3.2, we translate the RSML example above to SMV code.
- The complete SMV program is shown in Appendix A.
  - SMV Variables
  - RSML Transitions
  - Inputs
  - Prev and Timing Constraints

```
MODULE main
VAR
  u: boolean;
  v: boolean;
  w: boolean;
  switch: {up, down, test};
  alt: 0..20000;
  prev-alt: 0..200000;
  Alt-Layer: {High, Mid, Low};
  Alarm: {Shutdown, Operating};
  Mode: {Off, On};
  Volume: {12};
  time-Mid: 0..5;
DEFINE
  stable := !(u|v|w);
  in-Sys := 1;
  in-Alt-Layer := in-Sys:
  in-High := in-Alt-Layer & Alt-Layer = High;
  in-Mid := in-Alt-Layer & Alt-Layer = Mid;
  in-Low := in-alt-Layer & Alt-Layer = Low;
  in-Alerm := in-Sys;
  in-Shutdown := in-Alarm & Alarm = Shutdown;
  in-Operating := in-alarm & alarm = Operating;
  in-Mode := in-Operating;
  in-Volume := in-Operating;
  in-Off := in-Mode & Mode = On;
  in-1 := in-Volume & Volume = 1;
  in-2 := in-Volume & Volume = 2;
```

```
t1 := in-High & u & alt >= 9950;
t2 := in-Mid & u
      & 1950 <= alt & alt <= 10050;
t3 := in-Low & u & alt <= 2050;
t4 := in-Mid & u & alt > 10050;
t5 := in-High & u & alt < 9950;
t6 := in-Low & u & alt > 2050;
t7 := in-Mid & u & alt < 1950;
t8 := in-Shutdown & u & switch=up;
t9 := in-Shutdown & u & switch=down;
t10 := in-Off & w & c;
t11 := in-On & w & in-Mid;
t12 := in-1 & v;
t13 := in-2 & v;
t14 := in-Shutdown & u & switch=test;
c := in-Low &
       (alt<1000
        |(alt<1500 & prev-alt<1500)
        |time-Mid >= 5);
```

```
ASSIGN
  init(Alt-Layer) := Mid;
  next(alt-Layer) :=
    case
      t1|t4    : High;
      t2|t5|t6: Mid;
      t3|t7    : Low;
      1        : Alt-Layer;
    esac;
  init(Alarm) := Shutdown;
  next(Alarm) :=
    case
      t8|t14: Operating;
      t9     : Shutdown;
      1      : Alarm;
      esac;
  init(Mode) := Off;
  next(Mode) :=
    case
      t10|t14: On;
      t8|t11 : Off;
      1      : Mode;
    esac;
  init(Volume) := 1;
  next(Volume) :=
    case
      t8|t13|t14: 1;
      t12       : 2;
      1         : Volume;
    esac;
  init(w) := 0;
  next(w) := t1|t2|t3|t4|t5|t6|t7;
  next(u)  :=
    case
      stable: {0,1};
      1      : 0;
    esac;
  next(v) :=
    case
      stable: {0,1};
      1      :  0;
    esac;
  next(switch) :=
    case
      stable: {up, down, test};
      1    : switch;
    esac;
  next(alt) :=
    case
      stable: 0..20000;
      1: alt;
    esac;
  next(prev-alt) :=
    case
      stable: alt;
      1      : prev-alt;
  esac;
  next(time-Mid) :=
    case
      t2|t4|tt7     : 0;
      stable & time-Mid < 5: time-Mid + 1;
      1                    : time-Mid;
    esac;
```

4. Translation Rules

- To explain the translation from RSML to SMV more generally and precisely, we first formally define an RSML machine as a state transition system given in Section 2.1, based on the operational semantics of RSML by Leveson et al.

4.1 RSML Machines as State Transition Systems

- RSML States
- Global States
- Initial Global States
- RSML Transitions
- Global Transitions

4.2 Translate Global States

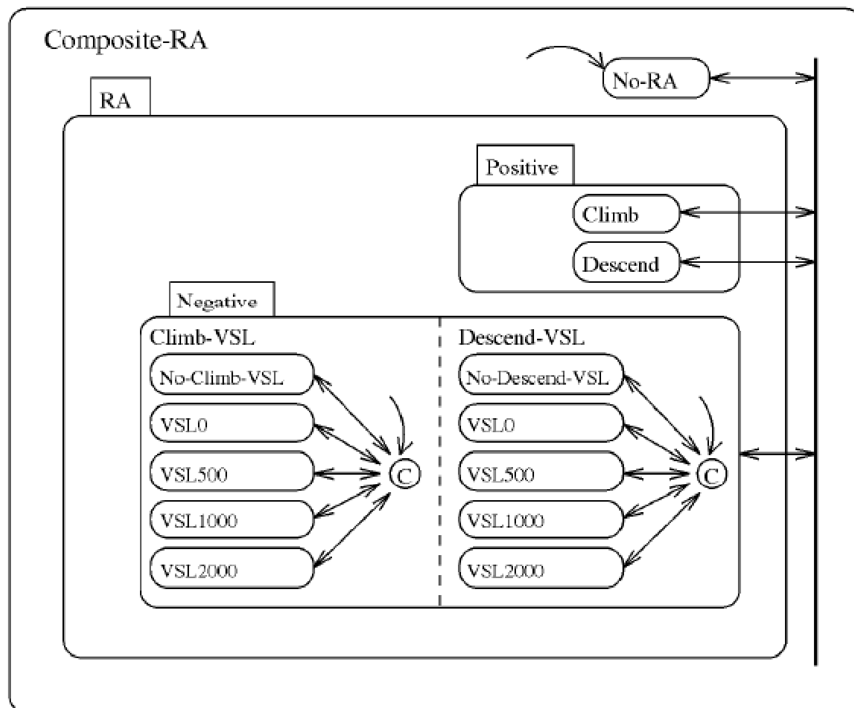

Fig. 5. Nested or-states in TCAS II.

```
VAR
  Composite-RA: {No-RA, Climb, Descend,
                 Negative};
  Climb-VSL: {No-Climb-VSL, VSL0, ... };
  Descend-VSL: {No-Descend-VSL, VSL0, ... };
DEFINE
  in-RA := in-Positive | in-Negative;
  in-Positive := in-Climb | in-Descend;
  in-Climb := in-Composite-RA
              & Composite-RA = Climb;
```

4.3 Translate Deterministic Transitions

```
1. For each p ∈ O:
   VAR
     p: Followers(p);
   ASSIGN
     init(p) := default*(p);
2. DEFINE
     in-root := 1;
3. For each s ∈ A:
   DEFINE
     in-s := in-leader(s) & leader(s) = s;
4. For each and-state or atomic state s ∉ A with
   parent p:
   DEFINE
     in-s := in-p;
5. For each s ∈ O with parent p:
   DEFINE
     in-s := in-p;
6. For each or-state p ∉ O:
   DEFINE
     in-p := ⋁_{s∈Children(p)} in-s;
7. For each e ∈ Events:
   VAR
     e: boolean;
8. For each e ∈ Events − External:
   ASSIGN
     init(e) := 0;
9. For each input variable y:
   VAR
     y: Range(y);
```

Fig. 6. Rules for declaring and initializing SMV variables for RSML machines.

```
10. For each tr ∈ Trans:
    DEFINE
      tr-enabled := in-src(tr) & trig(tr) & cond(tr);
      tr-taken := tr-enabled;
11. For each p ∈ O:
    ASSIGN
      next(p) :=
        case
          • For each tr ∈ Trans with
              Followers(p) ∩ Enters(tr) ≠ ∅,
            let s be the unique state in the set:
          tr-taken: s;
          • For the default branch:
          1: p;
        esac;
12. For each e ∈ Events − External:
    ASSIGN
      next(e) := ⋁_{tr: e∈acts(tr)} tr-taken;
13. For each e ∈ External:
    ASSIGN
      next(e) :=
        case
          stable: {0,1};
          1: 0;
        esac;
14. For each input variable y:
    ASSIGN
      next(y) :=
        case
          stable: Range(y);
          1: y;
        esac;
15. DEFINE
      stable := ¬⋁_{e∈Events} e;
```

Fig. 7. Rules for translating deterministic RSML transitions.

4.4 Translate Nondeterministic Transition

$$
\begin{array}{ll}
10'. & \textit{For each } tr \in \textit{Trans:} \\
& \texttt{DEFINE} \\
& tr\texttt{-enabled := in-}src(tr)\ \texttt{\&}\ trig(tr)\ \texttt{\&}\ cond(tr); \\
& tr\texttt{-taken := } tr\texttt{-enabled} \\
& \qquad\qquad\quad \texttt{\&}\ \bigwedge_{s \in Enters(tr)} \texttt{next(in-}s\texttt{)} \\
& \qquad\qquad\quad \texttt{\&}\ \bigwedge_{e \in acts(tr)} \texttt{next(}e\texttt{);} \\
11'. & \textit{For each } p \in \mathcal{O}\textit{:} \\
& \texttt{TRANS} \\
& \quad (p = \texttt{next(}p\texttt{)}) \vee \\
& \quad \bigvee_{Followers(p) \cap Exits(tr) \neq \emptyset} tr\texttt{-taken} \\
16'. & \textit{For each } tr \in \textit{Trans:} \\
& \texttt{TRANS} \\
& \quad \neg tr\texttt{-enabled} \vee \\
& \quad \left(\neg tr\texttt{-taken} \leftrightarrow \bigvee_{tr' \in Conflict(tr)} tr'\texttt{-taken} \right)
\end{array}
$$

Fig. 8. Rules for translating a class of nondeterministic RSML transitions.

4.5 Translate Timing Constraints

- Since time grows without bound, the underlying state transition system in general has an infinite number of global states and BDD based model checking becomes inapplicable.

- Fortunately, many common cases can be handled.

```
VAR
  θ: 0..kθ;
ASSIGN
  next(θ) :=
    case
      ∨_{tr∈T_θ} tr: 0;
      stable & θ < kθ: θ + 1;
      1: θ;
    esac;
```

4.6 Translate Prev

- When the value of PREV(*y*) for some input *y* is needed, we use the following code:

```
VAR
prev-y: Range(y);
ASSIGN
  next(prev-y) :=
    case
    stable: y;
    1: prev-y;
    esac;
```

4.7 Miscellaneous

- Other RSML Constructs
- Granularity of Global Transitions
- Alternative Semantics

5. Obstacles

- After we derived the translation rules in the previous section, we had to overcome a number of obstacles to make model-checking the TCAS II specification feasible.

- TCAS II
  - The first obstacle to analysis was its sheer size.
  - Own-Aircraft has close interactions with another state machine called Other-Aircraft.

- BDDs
  - To use BDDs, we had to assume that these inputs are bounded integers.

- SMV
  - BDD size and linear arithmetic
  - Counterexample search

6. Results of Analysis

TABLE 1
RESOURCES USED IN ANALYSIS

Properties	Result	Time (sec)	No. of BDD Nodes	Memory Allocated (MB)
Building the transition relation	N/A	46.6	124,618	7.1
Transition consistency	False	387.0	717,275	16.4
Function consistency	False	289.5	387,167	11.5
Step termination	True	57.5	142,937	7.4
Descend Inhibition	True	166.8	429,983	11.8
Increase-Descend Inhibition	False	193.7	282,694	9.9
Output agreement	False	325.6	376,716	11.6

6.1 Transition Consistency

- Transition consistency
  - **AG ! (t9 & t12)**
- Soundness of the Analysis

LEMMA 1. *Given two state transition systems* $M_1 = \langle Q, R_1, I \rangle$ *and* $M_2 = \langle Q, R_2, I \rangle$ *with identical state spaces and initial states. Define*

$$N = \{q \in Q \mid \exists q'.\ \ (q, q') \in (R_2 - R_1) \cup (R_1 - R_2)\}.$$

The set N is reachable in M_1 *if and only if it is reachable in* M_2.

6.2 Function Consistency

- The value of the function Displayed-Model-Goal, shown in Fig. 10, is displayed to the pilot when an event called Composite-RA-Evaluated-Event occurs.

Displayed-Model-Goal =

Value	Condition	Case
0	**if** Composite-RA **not in state** Positive	/* case 1 */
Max(Own-Track-Alt-Rate, PREV(Displayed-Model-Goal), 1500 ft/min)	**if** (New-Climb **or** New-Threat) **and** **not** New-Increase-Climb **and** **not** (Increase-Climb-Cancelled **or** Increase-Descend-Cancelled) **and** Composite-RA **in state** Climb	/* case 2 */
Min(Own-Track-Alt-Rate, PREV(Displayed-Model-Goal), −1500 ft/min)	**if** (New-Descend **or** New-Threat) **and** **not** New-Increase-Descend **and** **not** (Increase-Climb-Cancelled **or** Increase-Descend-Cancelled) **and** Composite-RA **in state** Descend	/* case 3 */
2500 ft/min	**if** New-Increase-Climb	/* case 4 */
−2500 ft/min	**if** New-Increase-Descend	/* case 5 */
Max(Own-Track-Alt-Rate, 1500 ft/min)	**if** Increase-Climb-Cancelled **and** **not** New-Increase-Climb **and** Composite-RA **in state** Positive	/* case 6 */
Min(Own-Track-Alt-Rate, −1500 ft/min)	**if** Increase-Descend-Cancelled **and** **not** New-Increase-Descend **and** Composite-RA **in state** Positive	/* case 7 */
PREV(Displayed-Model-Goal)	Otherwise	/* case 8 */

```
AG (Composite-RA-Evaluated-Event ->
    !((Case-1 & Case-2) |
      (Case-1 & Case-3) |
              ⋮
      (Case-6 & Case-7)))
```

Fig. 10. Definition of Displayed-Model-Goal.

6.3 Step Termination

- AG EF stable
- which means that the machine is stable infinitely often. In other words, it can only stay unstable for a finite number of microsteps.

6.4 Inhibition of Resolution Advisories

6.5 Output Agreement

7. Related Work

- Case Studies
- Approaches to Fighting State Explosion
- Consistency and Completeness
- Hybrid Systems

8. Discussion

- Feasibility
  - Restriction to Finite States
  - Regularity
  - Scale
- Model Checking as a Design Tool
  - Understanding and Documentation
  - Iterative Development
- Tool Integration
- Properties to Check
- Nonlinear Arithmetic
- More Case Studies

9. Conclusion

- We have shown how to translate part of a large system requirements specification into input to a symbolic model checker, and check several nontrivial properties.
- Our approach to analyzing the specification iteratively, by modeling some components nondeterministically and then refining them, proved to be powerful.
  - These are critical steps towards realizing symbolic model checking as an effective tool in the process of analyzing and developing software specifications.
- We believe that this investigation contributes to an increase in optimism that symbolic model checking can overcome predicted impediments and thus be successful in the analysis of realistic software specifications.