DISTRIBUTION STATEMENT A – Approved for Public Release. Distribution is unlimited.
Ref: LM Aero PIRA AER200910006

A Model Checking Example

Solving Sudoku Using Simulink Design Verifier

ABSTRACT. This paper presents an easy-to-understand application of formal methods—specifically, model checking. Through an example based on the popular game Sudoku, I demonstrate the power and simplicity of this technology as implemented within Simulink®—a development environment for Model-Based Design. The overarching theme to consider is an analogy of the game to real-world constraint problems. The intent is to show a transition of the technology to real-world engineering problems and how model checking can be used in a full-scale system development process.

Copyright © 2009 – Lockheed Martin Corporation.

A Model Checking Example – Solving Sudoku Using Simulink Design Verifier

By Walter A. Storm, Lockheed Martin Aeronautics Company

Abstract. This paper presents an easy-to-understand application of formal methods—specifically, model checking. Through an example based on the popular game Sudoku, I demonstrate the power and simplicity of this technology as implemented within Simulink®—a development environment for Model-Based Design. The overarching theme to consider is an analogy of the game to real-world constraint problems.

INTRODUCTION

Sudoku is a logic-based number-placement puzzle. The objective is to populate a 9x9 grid so that each column, row, and 3x3 box contains a single instance of the digits 1-9. The game starts with a partially completed grid, and the solution to the puzzle is the arrangement of digits that meet the single-instance criteria.
The Sudoku grid is analogous to the complex finite state machines (often implemented as hybrid control automata) that are responsible for executing the modes and behaviors of emerging software systems. As the grid is populated, the temporary switching and storing of digits is representative of the various states and modes that the system can enter at any given time. The alteration of the grid is a result of the environment in which the system operates.

The strategy behind using a model checker to solve a Sudoku puzzle is this: formulate a logical proposition that suggests, given an initial state, no cases exist that meet all Sudoku requirements. The resultant counterexample is a solution to the puzzle.

Formalizing the Requirements

Our approach to the Sudoku example is to first formalize the requirements of the game as a graphical model in Simulink (Figure 1). This formalization consists of an initial board and an input vector that represents the puzzle's environment—essentially, all the blank spaces to which a digit can be assigned.

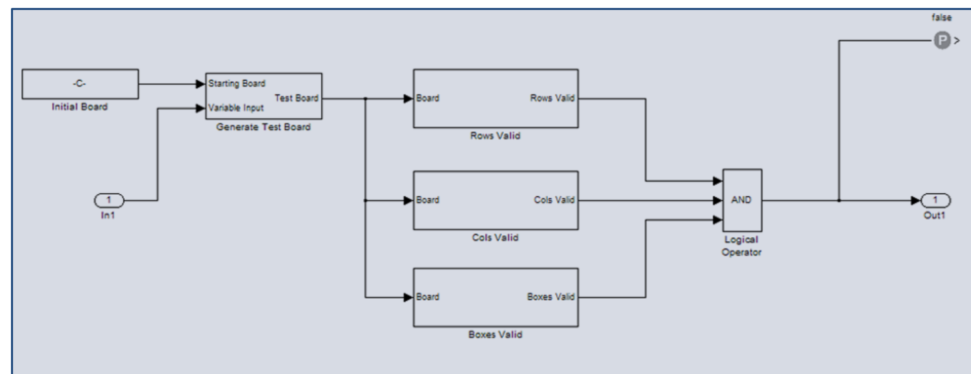

Figure 1 - Model of Sudoku Requirements.

The initial board is blended with the environment (the In1 port) to produce a resultant state—the *test board*. The resultant state is checked for valid rows, columns, and 3x3 boxes. If all three conditions are met, then the given state is a valid solution to the Sudoku. The *P* block is a formal property (or logical proposition) that claims the output of the *AND* block is **false** for all possible cases.

Formally Defining Valid Rows, Columns, and Boxes

Each row, column, and box is passed through a *Valid Set* subsystem that simply checks for a single instance of the digits 1-9 in a set. This is done by extracting a particular row, column, or box from the *test board* (Figure 2).

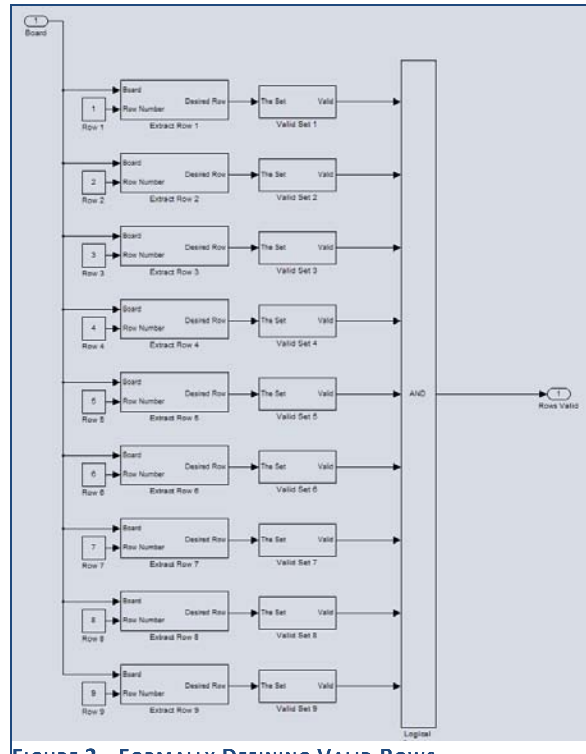

Figure 2 - Formally Defining Valid Rows.

This process is analogous to graphically representing the valid states of a system relative to the environment in which the system operates. For example, a *Valid Set* for an autonomous landing sequence would include an explicit requirement that the landing gear is down.

Note that there is no attempt here to enumerate or anticipate the potential states and modes of the system as it is excited through environmental manipulation—only the requirements of the system are graphically defined. That is, we are not implementing any tricks or techniques that may be particular to Sudoku solvers—we are defining just the requirements of the game in a formal, structured way.

This process of formally defining system requirements is an entirely different paradigm from that of traditional testing. Through propositional logic, one does not attempt to define specific input conditions and enumerate the expected output—rather, the objective is to define specific *properties* of the system output that must hold throughout *all* possible input conditions.

Running the Model Checker

The underlying proof system in Simulink Design Verifier™ is the Prover Plug-In® from Prover Technology. This engine is built upon Gunnar Stålmarck's patented proof procedure for propositional logic[1],a technique based on efficient use of Boolean Satisfiability. Understanding the theory and mechanics of this technology involves a solid background in computer science and demands a good understanding of proof systems in general; however, through Simulink Design Verifier the ultimate implementation of the technique is boiled down to a single mouse click (Figure 3).

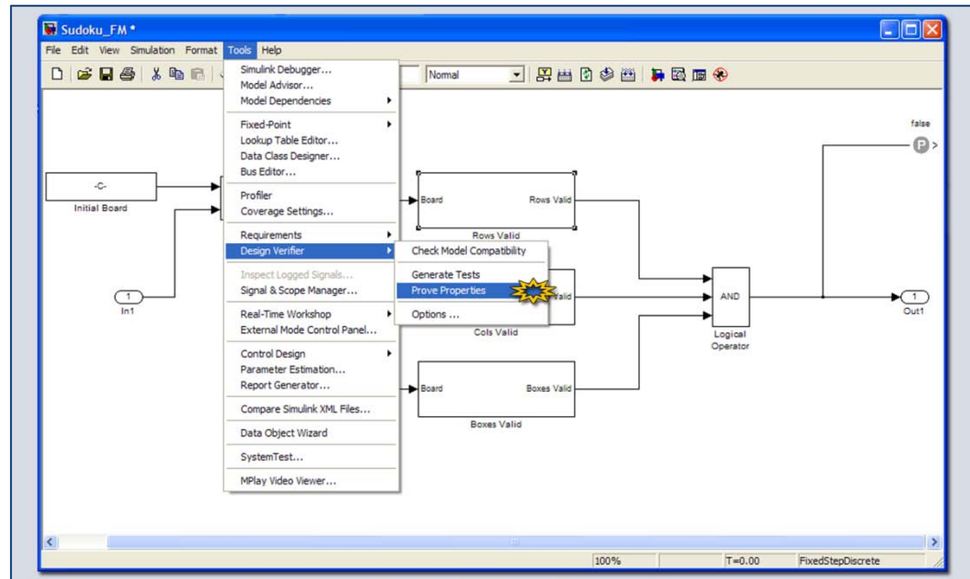

Figure 3 - Simulink Design Verifier Implementation of the Prover Plug-In.

Once invoked, the model checker symbolically explores the entire state-space of the system in search of a violation to the logical proposition (i.e. property). If a violation of the property is identified, the model checker returns a counterexample that enumerates all the system states leading up to the violation—in essence, a test case (Figure 4).

Initial Board

					3		8	5
		1		2				
			5		7			
		4				1		
	9							
5							7	3
		2		1				
				4				9

Counterexample with Values for In1

9	8	7	6	5	4	3	2	1
2	4	6	1	7	3	9	8	5
3	5	1	9	2	8	7	4	6
1	2	8	5	3	7	6	9	4
6	3	4	8	9	2	1	5	7
7	9	5	4	6	1	8	3	2
5	1	9	2	8	6	4	7	3
4	7	2	3	1	9	5	6	8
8	6	3	7	4	5	2	1	9

Figure 4 - A Violation of the Sudoku Property.

This violation, or counterexample, is representative of what is commonly referred to as a *sneak circuit*, or an overall *bad day*. The model checker was given a logical proposition that this

[1] *M. Sheeran and G. Stålmarck,* "A tutorial on Stålmarck's proof procedure for propositional logic ," *Proceedings of the 2nd Intl. Conf. on Formal Methods in Computer-Aided Design*, FMCAD'98, Palo Alto, CA, USA, 4--6 Nov 1998.

combination of states *shall never* occur—a proposition that turned out to be false. In this example, the violation is actually the solution to the Sudoku.

THE COUNTEREXAMPLE

A counterexample is a frame-by-frame account of the values assigned to the specific variables responsible for falsifying a proof objective. In essence, it is a test case that is known to violate the requirement. The implementation of the counterexample in Simulink Design Verifier is through an HTML report. The counterexample for the Sudoku example is presented in Figure 5.

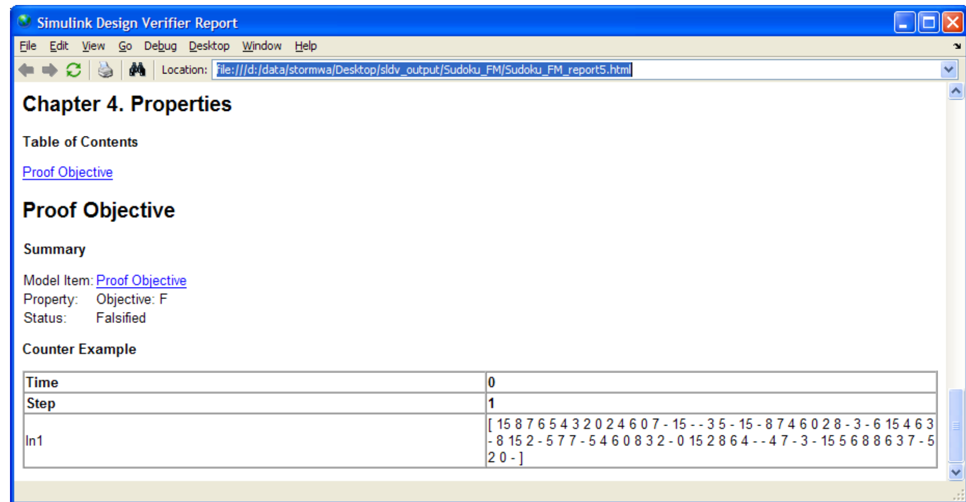

FIGURE 5 - SUDOKU COUNTEREXAMPLE FROM SIMULINK DESIGN VERIFIER

Notice that the property was violated in the initial time step. If the graphical requirements had any state information or history, or if the logical proposition had inherent temporal connectives, then there would be a sequence of time steps involved in the counterexample—the formalization of a chess match, for example.

Note also that the counterexample shown in Figure 5 is not as clear as the interpretation shown in Figure 4. Due to the symbolic representation of the system within the proof engine, the input values of the system are able to assume any possible value unless specifically constrained. Therefore, some interpretation is needed to fully understand the counterexample. In this instance, the In1 values must be blended with the initial board to produce a meaningful interpretation (Figure 6).

Let B:= Initial Board Value
Result=IF(B=0,IF(In1>=9,9,IF(In1<=1,1,In1)), B)

Row 1	Initial Board	0	0	0	0	0	0	0	0	0
	In1	15	8	7	6	5	4	3	2	0
	Result	9	8	7	6	5	4	3	2	1
Row 2	Initial Board	0	0	0	0	0	3	0	8	5
	In1	2	4	6	0	7	-	15	-	-
	Result	2	4	6	1	7	3	9	8	5
Row 3	Initial Board	0	0	0	0	0	3	0	8	5
	In1	2	4	6	0	7	-	15	-	-
	Result	2	4	6	1	7	3	9	8	5

FIGURE 6 – OUTPUT INTERPRETATION.

FIXING THE PROBLEM

When faced with a counterexample, one must decide on a course of corrective action. The first step is to verify that the counterexample is, in fact, a real system problem. In the Sudoku example, perhaps the sequence in the counterexample has already been identified as a possible outcome (i.e. a cohort may have solved the puzzle weeks ago but never documented the results). Fair enough—simply iterate on the formal requirements, add the known solution as an environmental assumption, and run the model checker again (Figures 7 and 8).

FIGURE 7 – DOCUMENTING A KNOWN SOLUTION AS AN ADDITIONAL CONSTRAINT.

With the known solution as an additional constraint, the property is now proven valid—meaning that no additional cases meet the constraints of the formal requirements. Therefore, the unique solution to this Sudoku is now formally documented in the model.

Had another solution existed, the analysis process would repeat,building additional confidence in the design and requirements.

FIGURE 8 - A VALID PROPERTY.

CONCLUSION

This example illustrates the power, flexibility, and simplicity of emerging formal verification techniques. The fact that this process could accompany early design activities means that the problems identified are much easier and less expensive to fix than they would have been if caught in the lab or the field.

Model checking is not a replacement for testing, and not all problems can be formalized. There are, however, many areas where this technology is both applicable and highly beneficial--specifically, verifying the state and mode transitions of hybrid control automata, complex switching and Boolean mode logic (finite-state machines), and solving many constraint problems (such as the Sudoku example).

I believe that many disciplines of systems modeling and development can benefit from this technology, and I encourage you to consider how model checking can be used in your environment.

91880v00 8/10