Improving an Industrial Test Generation Tool Using SMT Solver

Hao Ren¹, Devesh Bhatt^{2(✉)}, and Jan Hvozdic³

¹ Department of Electrical and Computer Engineering, Iowa State University,
Ames, IA 50014, USA

² Honeywell Aerospace Advanced Technology, 1985 Douglas Dr N,
Golden Valley, MN 55422, USA
`devesh.bhatt@honeywell.com`

³ Honeywell spol. s.r.o. - Aerospace Engineering, 100 Turanka,
627 00 Brno, Czech Republic

Abstract. We present an SMT solving based test generation approach for MATLAB Simulink designs, implemented in the HiLiTE tool developed by Honeywell for verification of avionic systems. The test requirements for a Simulink model are represented by a set of behavioral equivalence classes for each block in the model, in terms of its input(s) and output. A unique feature of our approach is that the equivalence class definitions, as well as the upstream subgraph of a block under test, are translated as constraints into SMT expressions. An SMT solver is called at the back-end of HiLiTE to find a satisfiable solution that is further augmented into an end-to-end test case at the model level.

1 Introduction

As the industry practices engage model-based design increasingly, model-based verification and testing [1] techniques emerge to keep up with the trends. In avionics area, comprehensive testing methods and tools are required to assure that safety-critical systems like flight controls are certified to the guidelines established by standard processes such as the DO-178C [2].

At Honeywell, researchers have developed the *Honeywell Integrated Lifecycle Tools & Environment (HiLiTE)* suite of tools for the automated verification of avionics applications developed using MATLAB Simulink/Stateflow. HiLiTE performs automatic test generation [3] on Simulink models based upon the low-level requirements (LLRs) expressed by the model elements. The tests are then applied to the executable object code generated from the model to verify that the code complies with the LLRs in the design model. HiLiTE has been qualified as a DO-178C verification tool and deployed in several avionics product certifications to deliver significant cost savings in the verification effort.

This paper presents an SMT solving technique to extend the earlier heuristics-based test case generation approaches implemented in HiLiTE, providing improved performance on models with complex constraints or non-linear

This research was supported in part by NASA Contract NNA13AC55C.

arithmetic computations. SMT solving is the decision procedure of determining whether a formula in first-order-logic is satisfiable and finding a concrete solution if it is. SMT Solvers, such as Z3 [4], Yices [5], etc., have rapidly matured over the last 5 years and have been used in various areas including automated test case generation [6, 7]. In our SMT solving based approach, each LLR equivalence class for a block type is represented by a set of constraints, applied on block-level input(s) and expected output. Meanwhile, test space is also constrained by the subgraph environment that the block under test (BUT) is embedded in. The collection of constraints can be formulated as an SMT problem and expressed in a standard format by HiLiTE in an automatic fashion. SMT solver is then called to generate the satisfiable solution once for all ports in the related subgraph. The solution is merged back to the entire graph for a complete model-level input-to-output test case. With the integration of heuristics and SMT solving techniques, HiLiTE has been successfully used to generate requirement-based test cases for a great range of large-scale complex constrained avionics models.

Section 2 describes the HiLiTE normal test case generation approach and the need for improvements. Section 3 describes the formalized language of equivalence classes of block's behaviors and SMT solving based test case generation approach. Finally, the conclusion and future work are discussed in Sect. 4.

2 HiLiTE Test Generation Approach

HiLiTE generates specific tests at the model level to exercise the equivalence classes of the behavior of each block embedded in the model. In the original HiLiTE tool, each equivalence class of a block's behavior is represented by a set of *test case templates*, each of which uses heuristics to select a specific combination of values for the block under test (BUT) input(s) and output that satisfy this equivalence class. Backward and forward propagation search through the computations of other blocks in the model generates a test vector in terms of model inputs and outputs to ensure controllability of the BUT inputs and observability of the expected BUT output. Figure 1 shows a Simulink model extracted from a complex industrial model to illustrate this.

Fig. 1. Test case generation for **product** block.

When the **product** block is the BUT, the test case template (Fig. 2) assigns the two inputs with non-zero values 2 and 4 respectively. After the backward and forward propagation search, the generated test vectors are given in Fig. 2 where blue column heading denotes model input and green denotes model output.

				Block Under Test Ports				
Test Case Name	Test Case Description	Test Requirements Covered	No. of Time Steps	Normal Range	Input[1]	Input[2]	Output	Step Description
				min →	0.44	0	0	
				max →	2	4	8	
Product Operation (Using OpMax)	Test product operation ,avoid 0 and 1 at the input.	Functional: Non Zero Output	1		2	4	8	Set all inputs to maximum value,avoid 0 and 1 at input

signal name → zone 2 multiplier equalTo.01 AdjustPct gainTable2.01 switch.01 GainAdj									
Test Case Name	Time Steps	range	Input[2]						Output
		min →	0	2	0	0	0	0.44	
		max →	10	2	4	1	100	2	
Product Operation (Using OpMax)	1		2	2	4	1	0.000000E+	2	8

Fig. 2. Test case template (top) and test vector for the **product** block in Fig. 1.

When the **switch** block is the BUT, the equivalence class requires different values at its data inputs (**FalseIn**, **TrueIn**) to verify unique impact of an input on the block’s output. One test case template assigns 44 to **FalseIn** and 46 to **TrueIn**, but this leads to a conflict at the model input **AdjustPct** after backward computation through the two look-up tables since their data points are in the same range. HiLiTE then further tries several alternative templates based on heuristics, yet all result in search failure. The root cause is that HiLiTE templates heuristics in the equivalence class domain prematurely pick block’s local input values, while this problem involves taking into account constraints imposed by the look-up table blocks driven by the same input **AdjustPct**.

3 Applying SMT Solving in Test Case Generation

Test generation difficulties such as those noted above can be addressed by an approach that solves computational constraints of the upstream subgraph of BUT in conjunction with the constraints on BUT inputs imposed by the behavior equivalence class. SMT can be thought of the constraint satisfaction problem expressed in Boolean formulas, linear/nonlinear arithmetic in integer/real domain, bit-vectors and so on. In HiLiTE, we added SMT solving based approach that embodies formulating test case generation constraints from both equivalence classes, constraints related to upstream source ports and the subgraph computations upstream of the BUT into an SMT problem. Therefore, constraints can be solved together to find a satisfying solution which excludes any conflicts.

3.1 Formal Specification of Equivalence Classes of Block Behaviors

An equivalence class of a block behavior, which represents a test requirement, is now expressed in HiLiTE with formalized rules on the block’s input and output

Table 1. Equivalence class definitions for switch block.

	Equiv. Class Definition for each input and output			
Equiv. Class Name	TrueIn	FalseIn	Control	Output
Verify TRUE Input	Exists	NEQ(FalseIn, TrueIn)	EQ(Control, 1)	Valid
Verify FALSE Input	Exists	NEQ(FalseIn, TrueIn)	EQ(Control, 0)	Valid

ports. These rules are expressed in a language as shown in Table 1 for **switch** block with rule names in blue. Each rule is automatically translated, based upon formal definitions, into SMT logic formula in a straightforward way. For example, “Exists” for port **TrueIn** is evaluated to “true” if any value is present, “NEQ(FalseIn, TrueIn)” of port **FalseIn** is interpreted as “FalseIn $\neq$ TrueIn”, and “EQ(Control, 1)” of port **Control** is interpreted as “Control = true” since **Control** has a Boolean type. The overall SMT logic formula for an equivalence class is the conjunction of individual formulas translated from equivalence class rules for each block port. E.g., the equivalence class “Verify TRUE Input” corresponds to “(FalseIn $\neq$ TrueIn) $\wedge$ (Control = true)”.

3.2 SMT Logic Formula for the Blocks’ Computation

SMT logic formula for a block captures the block’s mathematical computation for each time step; block formulas are stitched together to yield a subgraph formula. Let m be the number of time steps tried in test case generation. Examples:

- **Sum**: $\bigwedge_{j=0}^{m-1} (\text{Out}_j = \sum_{i=1}^n \text{In}_i)$.
- **Comparator**: $\bigwedge_{j=0}^{m-1} (\text{Out}_j = \text{In}_1_j \sim \text{In}_2_j), \sim \in \{=, \neq, >, <, \geq, \leq\}$.
- **Switch**: $\bigwedge_{j=0}^{m-1} (((\neg \text{In}_3_j) \wedge (\text{Out}_j = \text{In}_1_j)) \vee (\text{In}_3_j \wedge (\text{Out}_j = \text{In}_2_j)))$.
- **1D Look-up Table**: $\bigwedge_{j=0}^{m-1} ((\bigvee_{i=1}^n ((\text{In}_j \in \text{Range}_i) \wedge (\text{Out}_j = f_i(\text{In}_j))))$, where f_i is a linear function of In_j given the value of In_j in Range_i .
- **UnitDelay**: $(\text{Out}_0 = \text{initial_constant}) \wedge \bigwedge_{j=1}^{m-1} (\text{Out}_j = \text{In}_{j-1})$.

Note: support for time-dependent blocks (e.g., **UnitDelay**) also allows us to explore feedback loops in the model for bounded number of steps.

3.3 Formulated SMT Problem

HiLiTE explores the upstream subgraph of the BUT to ensure all constraints imposed by the subgraph computations on the test case generation are included. The subgraph exploration uses depth-first search, starting from the inputs of the BUT identified in its equivalence class, all the way back to the model inputs. Once the SMT-available subgraph is obtained, HiLiTE loops through each block in it to collect its SMT logic formulas. SMT logic formulas are further translated into expressions in SMT-LIB 2.0 standard format, recognized by popular SMT solvers like Z3, with actual port names as variables. Additionally, to specify

the block connections, each input in the formula is replaced by its source block output. For instance, the SMT expression for the *switch* block in Fig. 1 is *(assert (and (not (= gainTable 1.O1 gainTable 2.O1)) (= equalTo.O1 true)))*. Finally, the variables in the SMT expressions are substituted by a short form y_{i-j} (time step subscript $_j$ is omitted if there are no time-dependent blocks), where each block is assigned with a unique index i as shown in Fig. 3.

Fig. 3. Input file for SMT solver Z3 of test generation for *switch* block in Fig. 1.

The SMT logic formula is built initially with the number of time steps m determined by the equivalence class of the BUT: if the result of SMT solving is “unsat”, the formula is then updated with $m \leftarrow m + 1$. The process is repeated until either SMT solver returns “sat” or a pre-defined time step limit is reached. In the worst case, m may become very large before a value at some point (such as the output of a timer/integrator/counter) of the model is accumulated to satisfy the constraints, in which case SMT solver may break down or return “unknown”. To bypass blocks causing over-sized formulas, and certain mathematical blocks (e.g., *sin*) not supported by SMT solving, HiLiTE identifies those blocks, records them as *pending*, and explores the neighbor paths, resulting in an incomplete subgraph. The pending blocks and their upstream blocks are excluded from the solution returned by SMT solver. HiLiTE normal method then picks up from here to further propagate the pending values. An improvement can be done if the backward search goes through a switch block, only one data input of which has pending block(s) on its upstream. The values on the branch with pending block(s) do not matter if we force the switch block to disable that branch. This is done by modifying the SMT logic formula of that switch block. Suppose port *In*₁ of block *switch* is to be disabled, then the SMT logic formula becomes $\bigwedge_{j=0}^{m-1} (\text{In}_{3j} \wedge (\text{Out}_j = \text{In}_{2j}))$.

Fig. 4. Architecture of SMT solving in test case generation.

3.4 Tool Architecture

The SMT solving based test case generation is implemented by HiLiTE as a fully automated process shown in Fig. 4. Test cases needing SMT solving based approach are identified by the complexity of relationships detected during model analysis. For these, the test generation module formulates a collection of SMT expressions and writes them into a .smt2 file as described in Sect. 3. The SMT solver is called as a back-end, generating a solution which is then merged into test generation search space. HiLiTE normal method takes over from here to propagate the `switch` block output through the forward path to the model output `GainAdj` via the intervening `product` block, using a non-zero value for the second input of `product` block to ensure observability. This process results in valid test cases (Fig. 5) for the `switch` block in Fig. 2.

signal name →			zone	2	multiplier	equalTo.O1	AdjustPct	gainTable1.O1	gainTable2.O1	switch.O1	GainAdj
Test Case Name	Time Steps	range				Control		FalseIn	TrueIn	Out	
		min →	0	2	0	0	0	0.44	0.44	0.44	0
		max →	10	2	4	1	100	2	2	2	8
True Input Max Value to Output	1		0	2	2	0	60	0.9	2	0.9	1.8
False Input Min Value to Output	1		2	2	2	1	60	0.9	2	2	4

Fig. 5. Test vector generated for the `switch` block in Fig. 2 via SMT solving.

3.5 Nonlinear Applications

Modern SMT solvers are capable of solving a great range of non-linear problems used be computational intractable. Figure 6 shows a simple two-variable 2nd-order polynomial model. Z3 returns an answer for this case (as shown in Fig. 6) as in many other nonlinear problems.

Fig. 6. A polynomial model and the test vector for the `product` block via SMT solving.

4 Conclusion and Future Work

We extended the HiLiTE test generation capability with an SMT solving based approach for solving certain complex constrained problems. The improved tool combines HiLiTE normal search method and SMT solving, and has been successfully applied on many large-scale industrial models. HiLiTE is also being extended to derive invariant bounds on the number of time steps (e.g., for a timer) that will help bound the array size. We are also applying SMT solving to support such invariant generation, in which each condition-guarded path that captures a certain pattern of model behavior can be validated by SMT solving.

Open Access. This chapter is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, duplication, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, a link is provided to the Creative Commons license and any changes made are indicated.

The images or other third party material in this chapter are included in the work's Creative Commons license, unless indicated otherwise in the credit line; if such material is not included in the work's Creative Commons license and the respective action is not permitted by statutory regulation, users will need to obtain permission from the license holder to duplicate, adapt or reproduce the material.

References

1. Bhatt, D., Madl, G., Oglesby, D.: System Architecture Driven Software Design Analysis Methodology and Toolset. In: SAE International (2012)
2. RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification, RTCA Inc. (2011)
3. Bhatt, D., Madl, G., Oglesby, D., Schloegel, K.: Towards scalable verification of commercial avionics software. In: Proceedings of the AIAA Infotech @ Aerospace Conference, April 2010
4. Z3Prover. <https://github.com/Z3Prover/z3/wiki/>
5. The Yices SMT Solver. <http://yices.csl.sri.com/>
6. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating Tests from Counterexamples. In: ICSE (2004)
7. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 298–312. Springer, Heidelberg (2011)