Towards Scalable Verification of Commercial Avionics Software

Devesh Bhatt, Gabor Madl, David Oglesby, Kirk Schloegel
Honeywell International – Aerospace Advanced Technology
1985 Douglas Dr N, Golden Valley, MN 55422, USA

We describe a model-based approach for the automated verification of avionics systems that has been applied in Honeywell for the certification of complex avionics applications, such as flight controls and engine controls. The approach uses a symbolic analysis framework for MATLAB Simulink models, utilizing range arithmetic to represent test cases and equivalence-class transformations within a model diagram. Backwards search from a set of desired test-case values within the diagram is combined with forward-directed simulations from the inputs to resolve constraints and select values in the visited paths, leading to a set of diagram input/output values that produce the test case. Utilizing this approach, Honeywell has achieved $20 - 50\times$ reduction in certification costs compared to traditional analysis and testing methods, while maintaining scalability on complex real-life problems. As an example of the efficiency of this verification method, we describe a common design flaw that was uncovered in the early design phases of avionics software. We argue that finding such designs flaws is extremely hard by alternative methods such as directed or random simulations and traditional model checkers.

I. Introduction

Flight-critical avionics such as flight controls and engine controls must be certified to the highest level of assurance using the RTCA DO-178B software certification process[1]. Design and verification of complex systems requires significant effort, which has historically been accomplished using traditional design techniques and manual test creation from those designs.

With the increasing use of MATLAB Simulink[2] within the aerospace industry for the modeling and simulation-based evaluation of control algorithms, model-based verification plays an increasingly important role in the design and certification of aerospace systems software[3]. Current state-of-the-art model-based verification techniques and tools, however, fall short of handling the increasing complexity of these avionics systems and fully automating the achievement of the verification objectives. Thus, the increasing verification costs threaten to make the development of future avionics systems prohibitively expensive.

This paper presents a novel model-based approach for the verification of MATLAB Simulink designs and the resulting flight code that combines symbolic simulations with backwards reachability analysis, providing superior scalability compared to simulation-based verification and traditional model checking methods. This approach was implemented in the *Honeywell Integrated Lifecycle Tools & Environment (HiLiTE)*, and is being applied to the verification of several flight control, engine control, and perimeter control systems of commercial airliners. Using HiLiTE, Honeywell has reduced the cost and time of verification by a factor of $20 - 50$ compared to traditional methods in commercial avionics production programs.

This paper extends our earlier work on the model-based verification of safety-critical aerospace systems[4,5], as follows:

- We describe an approach for the symbolic analysis of MATLAB Simulink models that combines computation, control, and real-time properties in a unified analysis framework based on range arithmetic.
- We describe a class of design flaws that we have uncovered in the early design of an avionics system. The flaw results from a specific combination of real-time constraints, control flows, and input data, and cannot be detected when analyzing these properties independently.
- We summarize lessons learned from the application of our method to current and upcoming commercial avionics systems and discuss verification challenges that we anticipate in future avionics software.

II. Related Work

The verification of control algorithms in Simulink is commonly formulated as hybrid automata verification[6]. Reachability analysis on hybrid automata, however, is known to be undecidable in general[7]. Moreover, scalability is a major concern when applying hybrid automata verification to practical aerospace systems.

Alternative methods focus on identifying simulation traces of Simulink models that are equivalent with respect to a hybrid automata representation of the verification problem[8]. Practical flight control software, however, is too complex to be represented as formal hybrid automata models, as the verification must capture the data flow and the actual computations as well, not just the control flow. Moreover, to achieve coverage necessary for certification by repetitive simulations is overly expensive due to the large number of required simulations.

A model-based verification framework based on Simulink, and the LUSTRE[9] synchronous language was described in[10]. SCADE[11] is a commercial tool built on the LUSTRE language. Synchronous languages, however, employ aggressive abstractions and assumptions that are often overly restrictive in practical aerospace applications, thus limiting their application in practical design flows[12].

III. Verification Objectives for Avionics Software

The certification argument of an avionics system consists of successive levels of requirements-based analysis and verification, propagated from system-level requirements to lower-level requirements consisting of specifications and designs, including software designs. Certification guidelines such as DO-178B[1] require a verification effort to show that software designs and implementation satisfy a set of verification objectives.

The design of avionics software starts with the specification of high-level requirements for a software component. These requirements are based upon the system-level requirements that are allocated to the component. The next step is to capture the software design as one or more MATLAB Simulink models. Designs are commonly driven by control constructs such as filters, integrators, feedback loops, limiters, timers, latches. These constructs are represented by blocks or control blocks in the MATLAB Simulink model(s). The design is also referred to as low-level requirements and it includes the desired behavior and constraints for the particular block types used. The verification activity for a software component includes several steps:

1. Verifying that the software design satisfies its high-level and low-level requirements under normal and abnormal conditions.

2. Verify that the software design (low-level requirements) is accurate, consistent, robust, and verifiable.

3. Verifying that the object code that is deployed on the airplane (flight code) implements the design (low-level requirements) correctly and robustly. A by-product of this verification step is the structural coverage analysis on the object code.

In this paper we focus on the second and third activities. Key tasks that have to be performed include detecting potential division by zero, buffer/variable overflows, uncovering problems arising from interfacing floating point and integer arithmetic, detecting conflicting constraints in the control algorithm design, and finally verifying the functional behavior and robustness of each control construct on the flight code. In the following section we describe how HiLiTE addresses these challenges.

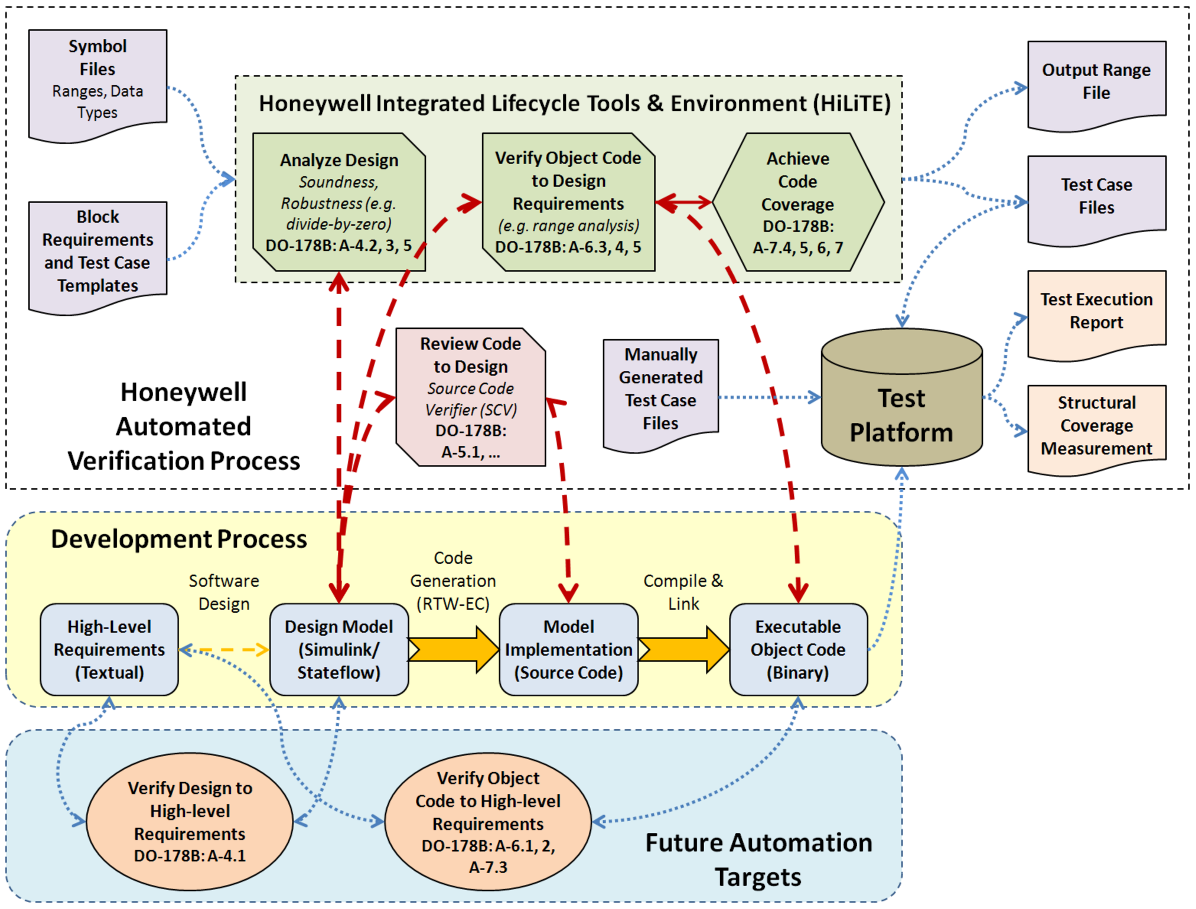
IV. Automated Verification for Certification of Avionics Applications

HiLiTE is a tool used within Honeywell for the requirements-based verification of aerospace systems that implements the proposed symbolic verification method for MATLAB Simulink models. HiLiTE has been qualified for use in DO-178B processes, and has been applied to the verification of thousands of practical MATLAB Simulink models that are derived from a wide variety of domains, such as flight control algorithms, discrete mode logic, built-in tests, hybrid perimeter controls, etc. The scalability of the HiLiTE approach has been well established for extremely large and complex Simulink models, resulting in $20 - 50\times$ cost and time savings in commercial avionics programs such as flight controls. In this section we describe the equivalence class framework that forms the basis of the symbolic analysis, and show how we utilize this framework to achieve the verification objectives including the detection of design flaws.

A. HiLiTE Overview

The objective of HiLiTE is to verify that the object code of a system conforms to the low level requirements represented by Simulink models. Figure 1 shows a typical development process and the parts that HiLiTE verifies. In our procedure the remaining verification problem is tackled by a combination of other tools and manual review.

At the conclusion of the development and V&V process, we have verified that the behavior of the object code conforms to the high level requirements. Additionally, we have verified the source code against the model as that is required by DO-178B[1] as the only place we can identify unintended behavior. HiLiTE is aware of the high level library blocks and their requirements, thus bringing verification to the level at which the developers are working.



Symbol Files
Ranges, Data Types
Block Requirements and Test Case Templates
Honeywell Integrated Lifecycle Tools & Environment (HiLiTE)
Analyze Design
Soundness, Robustness (e.g. divide-by-zero)
DO-178B: A-4.2, 3, 5
Verify Object Code to Design Requirements
(e.g. range analysis)
DO-178B: A-6.3, 4, 5
Achieve Code Coverage
DO-178B: A-7.4, 5, 6, 7
Output Range File
Test Case Files
Test Execution Report
Structural Coverage Measurement
Review Code to Design
Source Code Verifier (SCV)
DO-178B: A-5.1, …
Manually Generated Test Case Files
Test Platform
Honeywell Automated Verification Process
Development Process
Software Design
Code Generation (RTW-EC)
Compile & Link
High-Level Requirements (Textual)
Design Model (Simulink/ Stateflow)
Model Implementation (Source Code)
Executable Object Code (Binary)
Verify Design to High-level Requirements
DO-178B: A-4.1
Verify Object Code to High-level Requirements
DO-178B: A-6.1, 2, A-7.3
Future Automation Targets

Figure 1. Process Steps for DO-178B-compliant Automated Verification

HiLiTE uses MATLAB Simulink/Stateflow models, symbol files containing data-type and range information, and block requirements and test case template files as inputs. It generates *Test Generation Status Reports*, and *Test Case Files* as output. The test platform executes the test cases on the models executable object code and produces a *Test Execution Report*. Structural Coverage Measurement is performed using separate tools. Note that the test cases automatically generated by HiLiTE may need to be supplemented with hand-generated tests to fill requirements and robustness gaps. In our experience, HiLiTE generates more than 95% of required test cases, as described in Section VI..

B. Equivalence Class Framework

In the HiLiTE approach, each *test requirement* (for a control block) is represented by an *equivalence class* a set of input and expected output discrete-time-step sequences of ranges that can be used to verify the requirement. This equivalence class is expressed in terms of inputs and outputs signals of a block embedded within a model. The set of these signal values must be propagated backward to the model inputs and forward to an output using the semantics of the diagram and the blocks. This propagation results in a *test case*, expressed in terms of model's inputs and outputs, that maps the test requirement to the context of the model. The test case is then performed on the flight code to verify that the associated design requirement is satisfied by the code.

Forward and backwards propagation of sets of values through a complex model requires a search of the space of possible transformed value sets. The equivalence class is the *fundamental unit of the enumeration* of the search space within the diagram semantics. At each step in propagation, equivalence classes are transformed based on the particular (forward or backward) semantics of the incident block. So for example, for a given sequence of outputs, the analysis tries to infer the possible sequences of inputs, which requires calculation of inverse functions for the analyzed blocks. As a result, for a given output value, the set of corresponding input values can be infinite (e.g., back propagating a value of four through a two-input sum block results in an infinite number of possible input combinations).

Range arithmetic is used to make this analysis tractable. Range arithmetic is an extension of interval arithmetic[13] that also takes into account implementation-independent and -dependent type information. HiLiTE represents each element of an input and output sequence as a range. This approach groups together multiple (and possibly infinite) distinct input and output values for as long as they are indistinguishable from the perspective of verifying the particular requirement. Other techniques such as localized constraint solving, lazy evaluation, and practical heuristics techniques are also employed within HiLiTE to enable the verification of very large models in a scalable manner.

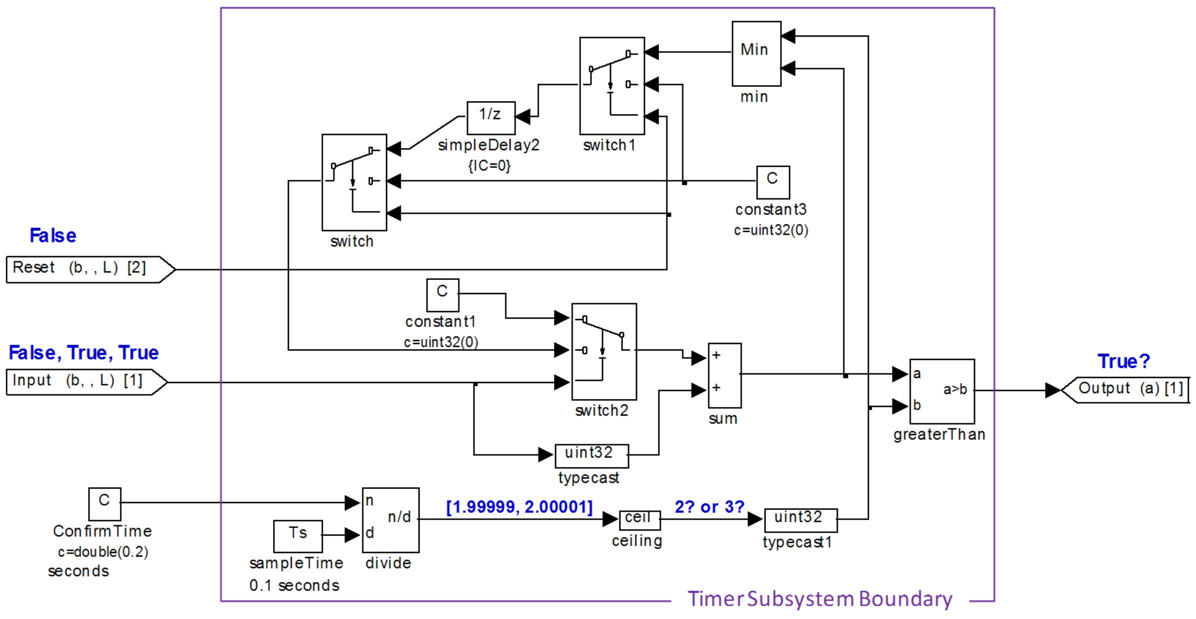


Figure 2. Design Ambiguity Defect Arising from Floating Point Representation Error

V. Detecting Ambiguities in Avionics Software Design

In addition to generating tests to be executed on the flight code, HiLiTE has also been successfully applied by control engineers in early design phases to detect remove such design defects such as overflow conditions (e.g. divide-by-zero), design ambiguity/conflicts, and un-testable conditions. Propagation of ranges within the diagram, based upon equivalence-class transformations and conflicts, is used uncover such defects in the model.

This section presents a case study excerpted from a set of real-world models that is representative of a class of design flaw arising from improper consideration of floating point representation error in computing time-dependent behavior. Figure 1 shows an actual subsystem design used to implement a timer. If the value on Input is true continuously for the duration of ConfirmTime, then Output becomes true. In this particular usage instance, constant 0.2 is connected to the ConfirmTime input of the timer, indicating that the output must become True when the Input has been true continuously for 0.2 seconds (or 2 time steps since sample time = 0.1 seconds). A problem of ambiguity occurs when ConfirmTime is an exact multiple of sample time Ts, due to the design construction using divide and ceiling blocks. Because of floating point representation error, the output of the divide block cannot be guaranteed to be exactly 2.0. Thus, if the output of the divide block is 1.99999, then the output of the ceiling block will be 2.0; if the output of the divide block is 2.00001, then the output of the ceiling block will be 3.0. This results in an ambiguity as to whether the timer will expire in 2 time steps or 3 time steps, and hence, leads to timing jitter.

During forward and backward propagation search to produce a True at Output, HiLiTE constructs equivalence classes around the divide and ceiling blocks that take the floating point representation error into account. This allows HiLiTE to automatically detect this ambiguity and report it as a design defect. We argue that finding such designs flaws is extremely hard by alternative methods such as directed or random simulations and traditional model checkers, since the floating point computation, the control flow, as well as timing properties all must be considered for the analysis.

VI. Practical Impact

HiLiTE was applied to the analysis of many large-scale safety-critical software designs used in several commercial airliners. Experience on these projects shows that HiLiTE achieves significant cost savings compared to traditional methods. Current deployments of the tool have reduced the costs of labor-intensive certification tasks (e.g. manual test generation) by a factor of 20 to 50 for certain *Integrated Modular Avionics (IMA)* applications, such as Boeing 787 flight controls. Table 1 provides an overview of HiLiTE performance metrics that have led to these savings.

An important objective of static and dynamic analysis of models is to catch problems early in the design cycle. To this end, Honeywell has begun the use of the HiLiTE tool suite to report design defects in models. The potential benefits of this capability have been established, and several Honeywell avionics product lines have now made this automated design defects analysis a standard part of the system engineering process. The current version of HiLiTE provides analysis and reporting of some of the key classes of defects as specified in Table 2.

The goal of HiLiTE from the outset has been to tackle real-world models. Over the years the complexity of both the target models and the analyses performed has grown, yet HiLiTE runtimes remain on the order of minutes for models

Application Domain	No. of Models	Types of Defects Found in Early Analysis	False Alarms	Requirements, Robustness Coverage	Code Coverage incl. MC/DC	Findings
Flight Controls	2000	overflow, underflow, anomalous behaviors, untestable/ unreachable constructs, floating point ambiguity	5–10%	95%	98%	protection mechanisms/ constraints elsewhere in the system mitigate many defects
Environment Control Systems	300	anomalous behaviors, untestable/ unreachable constructs	<5%	92%	–	still very early stages of design
Auxiliary Power Units Controllers	200	untestable/ unreachable constructs	<5%	95%	98%	defects were reduced with design maturity
Ground Engine Controllers	90	untestable/ unreachable constructs, divergent feedback loops	<5%	85%	90%	as above

Table 1. Usage of HiLiTE tool in Honeywell product lines for design analysis and test generation

Defect Name	Description	Example	Currently Detected	Effects
Numerical Overflow or Underflow	Range and error bounds on signals exceed data type limits or system physical limits	Result of a divide-by-zero	Yes	Overflow Exception
Frozen Requirements	The outputs of blocks are frozen to a constant value	Output of a zero-gain block	Yes	Code coverage holes
Untestable Conditions	Particular conditions cannot be exercised or mutually exclusive conditions hold	Over-specified transition guard	Some	Code coverage holes
Unreachable Constructs	Model structures lead to unreachable states, transition paths, or conditional statements	Default case of a switch stmt	Some	Unreachable constructs/code
Divergent Feedback Loops	Feedback signals diverge due to lack of design constraints	Open timer	Some	Overflow Exception
Anomalous Behaviors	Functional or robustness requirements of a function block are violated	Square root of a negative number	Some	Various
Floating-point Anomalies	Floating-point representation errors cause values to unintentionally cross thresholds	Ambiguous timer	No	Incorrect functional or timing behaviors

Table 2. Defects that need to be identified in flight-critical system models

on the order of hundreds of states or blocks and on the order of hours for the largest models we have encountered.

A key reason for this outstanding scalability is because HiLiTE first performs a suite of relatively fast and effective static analyses, such as type and range analysis, and uses the results of these to constrain the problem space to improve the performance and scalability of further dynamic analyses and automatic test case generation.

A. Analytical Evaluation of HiLiTE Savings

We now present an analytical evaluation of cost savings provided by HiLiTE based on the data presented in Table 1 and Table 2. The cost of certification using a traditional process can be described using the simple formula:

$$C_{traditional} = n \cdot c_m \tag{1}$$

where n is the number of models, and c_m is the average cost of generating tests and manually reviewing models. In contrast, in a HiLiTE-enabled design process, costs can be expressed as follows:

$$C_{hilite} = (1-p) \cdot n \cdot c_m + p \cdot n \cdot c_a \tag{2}$$

where n is the number of models, c_m is the average cost of generating tests and manually reviewing models, c_a is the cost of reviewing all models using an automated process, and $p = \frac{n_a}{n}$ is the fraction of the number of models that can be analyzed using the automated process (n_a), and the number of all models (n). The speedup compared to the traditional approach then can be estimated as follows:

$$S = \frac{C_{traditional}}{C_{hilite}} = \frac{n \cdot c_m}{(1-p) \cdot n \cdot c_m + p \cdot n \cdot c_a} = \frac{c_m}{(1-p) \cdot c_m + p \cdot c_a} \tag{3}$$

Analyzing a model using HiLiTE is several orders of magnitude faster than a manual review process. Honeywell has encountered many cases where 1000 hours of manual test generation could be replaced by automated analysis by HiLiTE in the order of 2 hours. More importantly, automated analysis by HiLiTE does not take up a verification engineer's time other than launching the analysis. Therefore, we can safely estimate that $c_a \ll c_m$. As the cost difference between automated and manual labor $(\frac{c_m}{c_a})$ approaches infinity, the overall savings than can be approximated as follows:

$$S \approx \lim_{\frac{c_m}{c_a} \to \infty} \frac{c_m}{(1-p) \cdot c_m + p \cdot c_a} = \frac{1}{1-p} \tag{4}$$

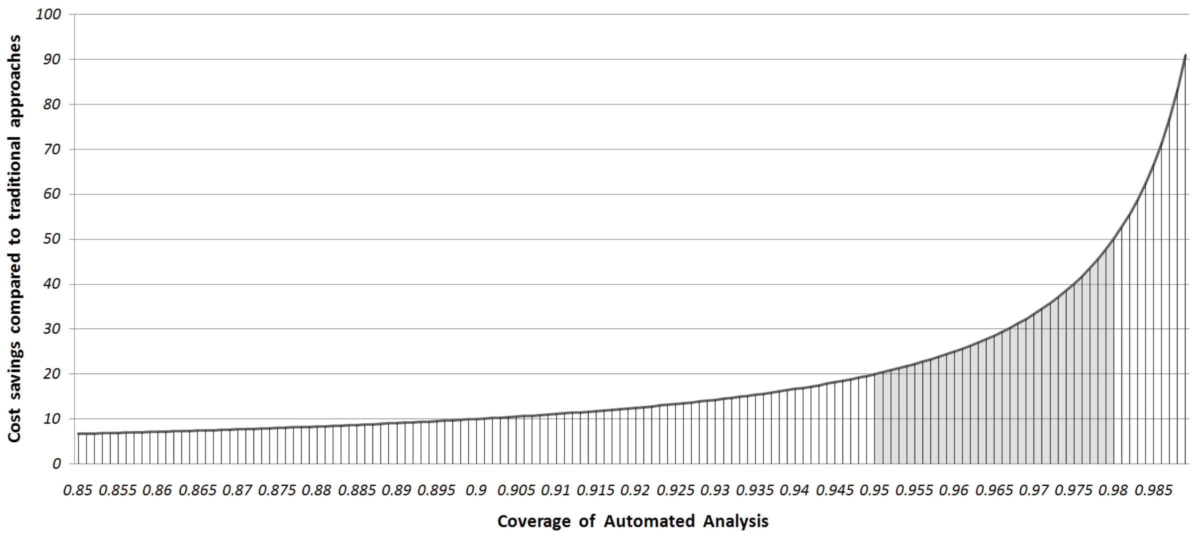


Figure 3. HiLiTE Savings as a Function of Test Coverage

Figure 3 demonstrates typical HiLiTE savings compared to traditional methods, as expressed in Equation 4. As the results show, coverage metrics are the key metric of speedups in certification processes. Given that HiLiTE reaches an average of $95 - 98\%$ coverage on typical production flight control systems, the typical savings are in the range of $20 - 50\times$, shown highlighted in Figure 3. Experience on the verification and validation of production commercial flight-critical systems have confirmed these results through countless examples.

VII. Lessons Learned in Model-based Verification of Avionics Software

Several testing groups across Honeywell have applied HiLiTE to support the verification of the most advanced and complex flight controls, engine control, and perimeter control system for commercial airliners. HiLiTE has automated the majority the effort associated with model analysis and generation of component tests. The following is a list of lessons learned from this experience:

- The cost benefits of verification automation can be effectively realized only when all specific steps to meet DO-178B verification objectives can be automated; otherwise manual-labor intensive reviews and translation are required.

- The certification arguments must be developed starting top-down from system level to software-component level, with design and verification artifacts at each stage and precise definition of exactly which verification objectives are automated by a tool. The tool must be qualified as a DO-178B verification tool in order to yield any cost benefits.

- The scalability of the verification method/tool is of utmost importance, since real avionics models can be very large and complex. If a verification tool does not scale to these models, then large parts of the automation benefits are lost. In addition to scalability to the size of models, the verification methods must work for different types of modeling patterns and types of algorithms.

- Detecting design flaws is early stages of development is very effective in achieving design assurance; the verification tools must provide a simple and fast method for designer to accomplish this.

- Some models contain complex structures for which it is extremely difficult to generate tests of upstream or downstream blocks. Sometimes these models cannot be broken down into simpler units due to performance or resource-constraint requirements. Hence, the test generation method must allow insertion of observation points and injection of secondary stimuli into the flight-code for thorough testing of the design requirements.

- While the widespread use of model-based design has enabled automated verification based upon the design, there are no viable methods to do the same for high-level requirements commonly captured in textual form. Formal specification of high-level requirements and use of requirements-patterns[14] are among some of the proposed approaches, indicating that significant technological and process issues need to be addressed.

References

[1] RTCA SC-167 / EUROCAE WG-12, DO-178B/ED12B, "Software Considerations in Airborne Systems and Equipment Certification," .

[2] The Mathworks, "MATLAB and Simulink," `http://www.mathworks.com`.

[3] Bhatt, D., Hall, B., Dajani-Brown, S., Hickman, S., and Paulitsch, M., "Model-based development and the implications to design assurance and certification," *Proceedings of Digital Avionics Systems Conference*, 2005.

[4] Bhatt, D., Hickman, S., Schloegel, K., and Oglesby, D., "An Approach and Tool for Test Generation from Model-based Functional Requirements," *Proceedings of 1st International Workshop on Aerospace Software Engineering*, 2007.

[5] Paulitsch, M., Ruess, H., and Sorea, M., "Non-functional Avionics Requirements," *Communications in Computer and Information Science*, Vol. 17, 2009, pp. 369–384.

[6] Agrawal, A., Simon, G., and Karsai, G., "Semantic Translation of Simulink/Stateflow models to Hybrid Automata using Graph Transformations," *Electronic Notes in Theoretical Computer Science*, Vol. 109, 2004, pp. 43–56.

[7] Henzinger, T. A., Kopke, P. W., Puri, A., and Varaiya, P., "What's decidable about hybrid automata?" *Journal of Computer and System Sciences*, Vol. 57, No. 1, 1998, pp. 94–124.

[8] Alur, R., Kanade, A., Ramesh, S., and Shashidhar, K., "Symbolic Analysis for Improving Simulation Coverage of Simulink/Stateflow Models," *Proceedings of EMSOFT*, 2008, pp. 89–98.

[9] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D., "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1305–1320.

[10] Tripakis, S., Sofronis, C., Caspi, P., and Curic, A., "Translating Discrete-time Simulink to Lustre," *ACM Transactions on Embedded Computing Systems (TECS)*, Vol. 4, 2005.

[11] Esterel Technologies France, "SCADE Suite," `http://www.esterel-technologies.com`.

[12] Durrieu, G., Laurent, O., Seguin, C., and Wiels, V., "Formal Proof and Test Case Generation for Critical Embedded Systems Using Scade," *Building the Information Society*, Vol. 156/2004, 2004, pp. 499–504.

[13] Moore, R. E., *Interval Arithmetic and Automatic Error Analysis in Digital Computing*, Ph.D. thesis, Stanford University, 1962.

[14] Zhou, C., Kumar, R., Bhatt, D., Schloegel, K., and Cofer, D. D., "A Framework of Hierarchical Requirements Patterns for Specifying Systems of Interconnected Simulink/Stateflow Modules," *Proceedings of the Nineteenth International Conference on Software Engineering and Knowledge Engineering*, 2007.