Model Based Analysis and Test Generation for Flight Software

Corina S. Păsăreanu, Johann Schumann, Peter Mehlitz, Mike Lowry
NASA Ames Research Center
Email: name@nasa.gov

Gabor Karsai, Harmon Nine, Sandeep Neema
ISIS, Vanderbilt University
Email: {gabor|hnine|sandeep}@isis.vanderbilt.edu

Abstract

We describe a framework for model-based analysis and test case generation in the context of a heterogeneous *model-based development paradigm that uses and combines MathWorks and UML 2.0 models and the associated code generation tools. This paradigm poses novel challenges to analysis and test case generation that, to the best of our knowledge, have not been addressed before. The framework is based on a common intermediate representation for different modeling formalisms and leverages and extends model checking and symbolic execution tools for model analysis and test case generation, respectively. We discuss the application of our framework to software models for a NASA flight mission.*

1. Introduction

This paper reports on an on-going project at NASA Ames, whose goal is to develop automated techniques for error detection in the flight control software for the next manned space missions. Such software needs to be highly reliable. The developers of the flight software chose an innovative *heterogeneous* model-based paradigm that combines model-based design using MathWorks[1] with UML 2.0 statechart models, together with the associated code generation tools. The MathWorks tools are used to develop math-intensive control software, while the UML-based tools are used for the rest of the software, including flight, ground, and simulation software.

The flight software will be complex, where errors can be caused by interactions among many components, whose dynamic behavior will be described using different modeling formalisms. The model-based approach not only provides leveraged generation of code for current and future platforms, but also enables early life-cycle (design stage) detection of errors because the software models are both formal and abstracted from some details of the target code.

In the past two decades the avionics software community has increasingly applied model-based software engineering, where models are used to specify software designs, and often executable code is generated automatically from the models. The models are expressed in domain-specific modeling languages with higher-level abstractions that are well-known and convenient for domain engineers. Flight control software have been developed for various vehicles using Matrix-X[2] and MathWorks' Simulink/Stateflow, which supports models based on dataflow diagrams and hierarchical finite state machines.

In spite of the popularity of model-based software engineering (in the style of the two leading products mentioned above), the current approaches to the Verification and Validation of model-based software are still very limited (see e.g., MathWorks' DesignVerifier and Section 5). Furthermore, the particular characteristics of the model-based paradigm using *heterogeneous* models poses the additional challenges of handling the different semantics of the modeling formalisms, while keeping the analysis tractable, and providing means of validating the model analysis results on the code that is generated from the models. To the best of our knowledge, these challenges are not addressed by any existing approaches or tools.

In order to study integration issues between components described using different modeling formalisms, we have developed a framework that is based on a common *intermediate representation* for different models and that leverages existing formal verification and test case generation technologies developed at Ames [6], [17]. The framework aims to provide *automated* techniques for analysis and test case generation for UML and Simulink/Stateflow models of mission-critical systems and to provide seamless integration with model based development frameworks.

We describe how we applied our framework to parts of the flight control software that is being developed for a NASA mission. Although we make our presentation in the context of a NASA project, we believe that our work should be relevant to other complex, safety critical model-based software that is built from heterogeneous components.

The rest of the paper is organized as follows. In the next section we give some background on modeling languages and associated tools. We then describe the our model based analysis and test case generation framework (Section 2), followed by a detailed description of the framework components: model transformation (Section 2.1), model analysis (Section 2.2) and test case generation (Section 3). We then

1. http://www.mathworks.com

2. http://www.matrixx.com

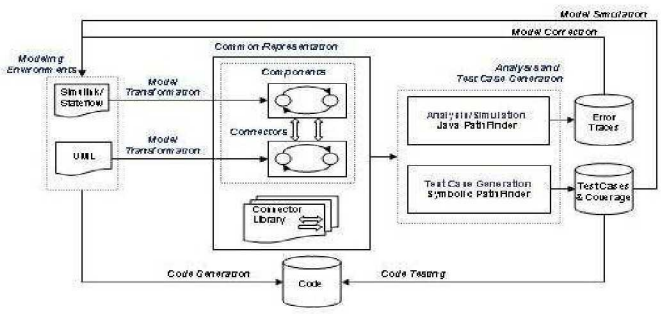

Figure 1. Model Based Analysis and Testing

describe the application of our framework (Section4), related work (Section 5) and conclusions (Section 6).

2. Framework for Model Based Analysis and Testing

Figure 1 depicts our framework that takes models created using different modeling environments and enables their analysis (model checking [?] and test-case generation) in a common, neutral environment. Our framework targets Mathwork's Simulink/Stateflow tool and Rhapsody's UML modeler, because this heterogeneous combination of modeling tools is used within our NASA project and it is also a good representative of a heterogeneous modeling environment. We focus in this paper on the analysis and test case generation of hierarchical state-machine models (i.e. Simulink/Stateflow and UML), since they form some of the most complicated parts of the flight software that we analyzed (e.g. mode logic).

We note however that our framework also provides automated support for translating, executing and analyzing Mathworks'Simulink data-flow models and Embedded Matlab (eML) code (out-of-scope for the current paper).

The framework is based on a *common intermediate representation* (a safe subset of Java) for flight-software models produced using different modeling environments. This representation is meant to bridge the gap between the semantics of different modeling formalisms via translating into a common format; this representation is executable and amenable to simulation, visualization and analysis using existing powerful V&V and test-case generation technology developed at NASA Ames.

The framework uses *model transformation techniques* [] to translate models into the common representation, based on meta-models built for Simulink/Stateflow and UML state-machines respectively.

The analysis of heterogeneous models is driven by the software architecture that defines the system in terms of components and connectors. Components, modeled using different formalisms, are running concurrently andhave well defined, typed interfaces (encoding the services provided/required by a component to/from its environment). The common representation is equiped with a communications infrastructure to support component interactions. An extensible library of connectors captures the different types of communication policies of interest in the target flight-software systems.

As mentioned, the common representation is specially tailored for analysis and it is tightly integrated into the JPF verification tool-set [?]. This enables us to leverage the verification technologies of JPF to validate and check for integration problems in heterogeneous models.

Properties to be checked with JPF are given in terms of assertions or safety monitors encoding software requirements, flight rules, etc. The error traces and the debugging information reported by JPF are used by the developers to correct the models.

Once the developers have enough confidence in the models, they can generate test cases (test vectors and test sequences) encoding the input values and the expected outputs. The user can also specify the desired code coverage criterion to be achieved by the test cases (e.g., state, transition, path coverage, or some other, user-specified coverage criteria). Also test cases for testing of user defined, domain-specific properties can be generated.

Test cases can be fed back to model simulators (e.g., Matlab's simulator) or can be used to test the actual code generated from the models. The code does not need to be auto-generated, but we assume a close correspondence between models and code.

Test cases can be used for the following activities: testing the code, validating the model transformation (e.g. by running them against Matlab's simulator), and validating the code generators. The model based test cases can reveal problems such as un-covered code, undesired discrepancies between models and code, etc. We believe that such testing should complement other analysis and testing activities at the code level.

Model execution semantics are implemented in the `StateMachine` class that is used for analysis or simulation. `StateMachines` have to provide a `run()` method that implements a driver for the state machine, that usually loops until appropriate end conditions are detected. The driver maintains a set of active states and a set of enabling events, and it systematically goes through the set of events to advance the state machines to the next set of active states, using the `step` method. One can obtain different execution semantics by customizing this driver. Off-the-shelf, the framework includes two implementations. The first one is suitable for model checking, and is designed to keep model and program states as closely aligned as possible. The second implementation allows stand alone execution outside JPF, and can be used for - possibly interactive - simulation. The framework also has preliminary support for running

multiple (homogeneous and heterogeneous) state machines concurrently.

We are working on equipping the common representation environment with a communications infrastructure to support interactions between heterogeneous components. Interactions will take place through explicit, typed component interfaces. Specific communication policies governing these interactions will be captured by connectors that will instantiate the generic communications infrastructure as required by a targeted application. Since connectors provide application-independent interaction mechanisms, we will equip the framework with an extensible library of connectors that can be reused across applications. For example, we have already added support for connectors that model procedure calls and event-based synchronization. We plan to extend the conncetor library towards domain-specific protocols and standards, such as ARINC 653, an RTOS API Specification with support for space and time partitioning in an Integrated Modular Avionics architecture, that will be used in the context of our NASA project. This work is in a preliminary phase and it will be performed in close collaboration with the developers of the Flight software, since the definition for component inter-communication is still under active debate among the development project.

2.1. Model Transformation

The model transformation component of our framework is used to translate various models into a common Java representation that is suitable for analysis. Model transformation is based on Model-Integrated Computing (MIC) [11], a technology for building domain-specific software development tools, which is supported by a tool suite [10] that includes a metaprogrammable model editor GME, a model transformation tool GReAT, and a software infrastructure for integrating model-based software development tool chains, called OTIF.

We have used the MIC tool infrastructure to build a translation tool chain whose main task is to bridge the gap between the analysis tools and the source Simulink/Stateflow models. The model translators have been implemented as graph transformation programs, where the input models are treated as typed, attributed graphs. The type system of the graphs is defined by a metamodel, which is constructed as a UML class diagram (for details see [10]).

Our translation tool chain includes the following elements: an *import translator* converts Simulink/Stateflow models into a format compatible with the MIC tools. This translator uses Matlab's API to access all necessary details of the Simulink and Stateflow models, and transcribes them into an equivalent model for the translation process. This approach avoids the necessity to develop a parser to directly read Mathwork's own and ever changing internal format.

The models, as imported from Simulink/Stateflow, do not contain sufficient information for the translation. In particular, data types of internal signals are missing. A *type inference analyzer* calculates this information. It starts from the input 'ports' of the toplevel model, which must be typed, and propagates their type through the dataflow operators used in the Simulink model. Every elementary operator in the Simulink diagram is well-defined, so the output data type of the operator instance can be easily determined. By forward tracing the dataflow graph our algorithm computes the data type for each intermediate 'signal'.

Three *model translators*, for Simulink, Stateflow and Embedded Matlab, respectively, translate the imported models into a language-independent executable format, SFC (a data structure similar to Abstract Syntax Trees used in compilers). The first two of the generators were implemented using graph transformations, as discussed above. Finally, a *code printer* converts the SFC data structures into a safe subset of Java.

The translated Simulink/Stateflow models follow the semantics as specified in the language documentation from Mathworks.

We note here that the model transformation component can perform several preliminary analyses on the imported models, which are complementary to the analyses performed by JPF and SPF. Specifically, we validate that the models follow the MAAB guidelines [13] that constrain the models to make them suitable for generating safe and efficient embedded code. We also analyze the call graph of the generated code and verify that there is no infinite recursion (which would lead to unbounded stack growth during execution, thus a catastrophic failure). The analysis takes advantage of the fact that recursive calls (if generated at all), are always protected with conditions. Other verification activities are also possible, as the tool chain is built using open interfaces, and XML is used for interchanging information between the elements of the toolchain.

For UML, we are working on defining new model transformations using the XML Metadata Interchange (XMI), an OMG standard that is commonly used for UML models.

2.2. Model Analysis with Java PathFinder

JPF is an explicit state software model checker for Java bytecode programs, and includes its own Java Virtual Machine (JVM) implementation that supports state storing and matching. Given the well known scalability problem of software model checking, JPF is focused on finding defects and producing and analyzing respective error traces. Defects can refer to non-functional properties like deadlocks and data races, or can be defined by user-provided, application or domain specific property modules.

The primary design goal of JPF is its extensibility, especially to achieve the required scalability. In addition to

mechanisms like partial order reduction and heap symmetry, JPF provides an array of extension mechanisms to define alternative search strategies, implement complex properties, abstract standard libraries using the Model Java Interface (MJI), observe system-under-test execution, define state space branches, and to implement different bytecode execution semantics. For details see, e.g., [6].

3. Test Case Generation with Symbolic PathFinder

For model-based test case generation we use Symbolic PathFinder [17], a recent extension to JPF that combines symbolic execution and constraint solving for automated test case generation. Symbolic PathFinder implements a symbolic execution framework for Java byte-code. It can handle mixed integer and real inputs, as well as multi-threading and input pre-conditions.

Symbolic execution [12] is a well-known program analysis that uses symbolic values instead of actual data as inputs and symbolic expressions to represent the values of program variables. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. The state of a symbolically executed program includes the (symbolic) values of program variables, a path condition (PC), and a program counter. The path condition is a boolean formula over the symbolic inputs, encoding the constraints which the inputs must satisfy in order for an execution to follow the particular associated path. These conditions can be solved (using off-the-shelf constraint solvers) to generate test cases (test input and expected output pairs) guaranteed to exercise the analyzed code.

Symbolic PathFinder implements a non-standard interpretor for byte-codes on top of JPF. The symbolic information is stored in attributes associated with the program data and it is propagated on demand, during symbolic execution. The analysis engine of JPF is used to systematically generate and explore the symbolic execution tree of the program. JPF is also used to systematically analyze thread interleavings and any other forms of non-determinism that might be present in the code; furthermore JPF is used to check properties of the code during symbolic execution. Off-the-shelf constraint solvers/decision procedures `choco` and `IASolver` [3] are used to solve mixed integer and real constraints. We handle loops by putting a bound on the model-checker search depth and/or on the number of constraints in the path conditions. Furthermore we have extended Symbolic PathFinder to handle input arrays of fixed size (in addition to inputs of primitive type).

By default, Symbolic PathFinder generates vectors of test cases, each test case representing input-output vector pairs. In order to test looping, reactive systems, such as the state-chart models in our model-based framework, we have extended Symbolic PathFinder to also generate test

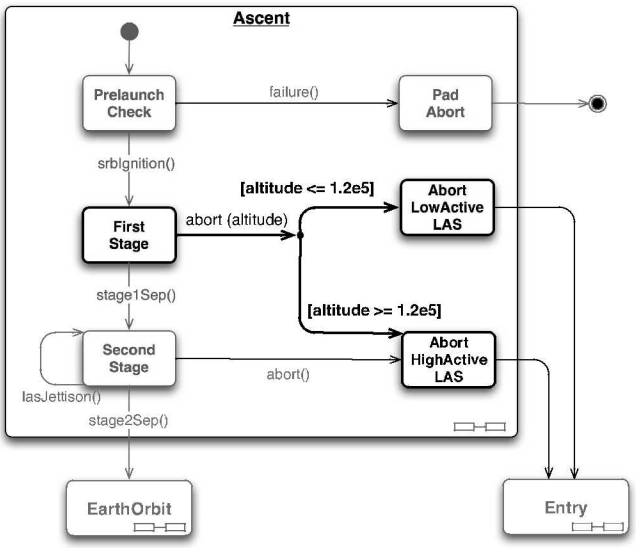

Figure 2. Model of the *Ascent* and *EarthOrbit* flight phases of a spacecraft

sequences (i.e., sequences of test vectors) that are guaranteed to cover states or transitions in the models (other coverages such as condition, or user-defined are also possible). This works by instructing Symbolic PathFinder to generate and explore all the possible test sequences up to some user pre-specified depth (or until the desired coverage is achieved) and to use symbolic, rather than concrete, values for the input parameters.

We have also customized SPF to print the generated test cases in terms of test drivers (for testing the auto-generated code) and in terms of simulation scripts; SPF's output can be customized easily for such purposes.

The models that we need to analyze perform complex mathematical computations. To generate test cases for them Symbolic PathFinder uses JPF's *native peers* mechanisms for modeling native libraries, i.e., to capture `math` library calls and to send them to the constraint solvers. The same mechanism was used to handle native code embedded in the models.

3.1. Example

We illustrate model based test case generation using the state-machine model of the *Ascent* and *EarthOrbit* flight phases of a spacecraft (Figure 2), where transitions are labeled with both events and guards on event parameters. The model has an error: there is an ambiguous transition going from state *First Stage* on an *abort* event when the value of the *altitude* is exactly `1.2e5`. Exposing this error requires a test sequence *srbIgnition(); abort(1.2e5)* that depends on both event and parameter choice, i.e., it is not

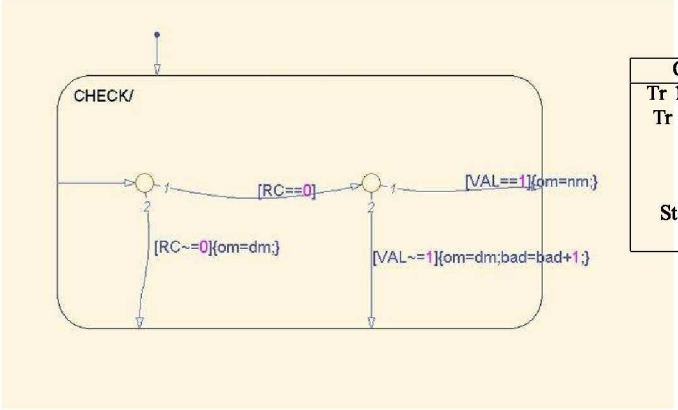

Figure 3. Stateflow example

amenable to simulation testing (that would fix the the event sequence apriori), to random testing, or to purely explicit state model checking techniques (that can not "guess" the exact value of the abort parameter that leads to error). However, the combination of explicit state model checking (to systematically explore all the methods sequences up to a given depth) with symbolic execution (to discover the right partitions on input values) allows us to discover such sequences automatically. We believe that the analysis of every realistically complex, reactive model with a data acquisition part requires such combined analysis tools.

4. Case Study

In this section, we will present some results of a case study, which applied our framework on safety-critical models for NASA flight software.

4.1. Analysis of a Sampling Port

We illustrate some of the features of our framework with a simple Stateflow example (see Figure 3). This model is a simplified version of one of the flight software components that we have analyzed. This diagram implements a sampling port. At each cycle of execution, the component first checks to see if a new message ($RC == 0$) is present. If not, the output (om) is set to a default message (dm) and the component waits for the next cycle. If a new message is present and it is valid, the output is set to the new message data ($om = nm$). If the new message is not valid, we increment variable *bad* (representing the port status) and set the output to the default message.

Although very simple, this example illustrates some of the problems discovered during the analysis of the real flight software component. We used JPF to perform simulations of the model and to check for properties, extracted from the informal documentation provided by the developers of the

Table 1. Generated test cases for model in Figure 3

Coverage	VAL_{in}	RC_{in}	dm_{in}	nm_{in}	bad_{in}	om_{out}	bad_{out}
Tr 1 2 VAL!=1	0	0	0	0	0	0	1
Tr 2 1 RC!=0	0	1	0	0	0	0	0
Tr 1	0	0	0	0	0	0	1
Tr 1 2	0	0	0	0	0	0	1
Tr 1 1	1	0	0	0	0	0	0
St CHECK2	0	0	0	0	0	0	1
Tr 2	0	1	0	0	0	0	0

```
{
  boolean_T sf_guard1 = false;
  if (sf_junct2_DWork.is_active_c1_sf_junct2 == 0) {
    sf_junct2_DWork.is_active_c1_sf_junct2 = 1U;
    sf_junct2_DWork.is_c1_sf_junct2 = (uint8_T)sf_junct2_IN_CHECK;
  } else {
    sf_guard1 = false;
    if (sf_junct2_U.In1 == 0.0) {
      if (sf_junct2_U.In2 == 1.0) {
        sf_junct2_B.om = sf_junct2_P.Constant3_Value;
      } else if (sf_junct2_U.In2 != 1.0) {
        sf_junct2_B.om = sf_junct2_P.Constant2_Value;
        sf_junct2_B.bad = (uint16_T)(sf_junct2_B.bad + 1);
      } else {
        sf_guard1 = true;
      }
    } else {
      sf_guard1 = true;
    }

    if (sf_guard1 == true) {
      if (sf_junct2_U.In1 != 0.0) {
        sf_junct2_B.om = sf_junct2_P.Constant2_Value;
      }
    }
  }
}
```

Figure 4. Measuring coverage on generated code

models. For example, JPF runs out of memory on this small example, the reason being that variable *bad* is unbounded, since it is being incremented without ever being reset. Thus, eventually an integer overflow error can occur. Interestingly, several other models that we have analyzed exhibited similar problems of missing resets.

We also used SPF to generate test cases for this model. Table 1 shows the test cases that are generated to achieve branch coverage.

In order to run the test cases from Table 1, we used RealTime Workshop to generate code and used a simple test harness. Code coverage was measured using `gcov`, which is a part of the GNU C compiler. While running these test cases on the code did not reveal any discrepancies between code and model in terms of expected output, we did discover some code statements that were not covered (Figure 4: unreachable statement is highlighted). Such examples of unreachable code in general poses a big problem in the development of flight code, as no dead code is allowed.

4.2. Analysis of Flight Software

We have applied our framework to several flight softwar components written in Matlab's Simulink/Stateflow. Thes models were built for the Launch Abort System (LAS) – on of the most important safety features of the new Orion spac capsule and the ARES rocket. In particular, we analyze the Guidance, Navigation, and Control (GN&C) part of th software that will be flight tested in the near future.

The entire GN&C software has been modeled using Math work's Simulink/Stateflow system, and large portions of th flight code are automatically generated using Mathwork' RealTime Workshop. This model has a highly hierarchica structure and contains Stateflow diagrams, Simulink block for continuous calculations and signal routing, as well a some embedded Matlab scripts. The entire system consists o roughly 25,000 Simulink blocks, 100 Stateflow diagrams o various sizes and complexity and more than 200 embedded Matlab scripts.

Since none of the tools (inhouse and commercial) could handle the entire system at once, we selected a number of representative subsystems for this case study. These examples included pure Simulink parts (to analyze the tool's capabilities for handling continuous and hybrid parts and signal flow), Stateflow diagrams (Statecharts), and subsystems with embedded Mathscript. The extraction of the subsystems under consideration proved to be far from trivial, because data types and signal dimensions were not encoded with all signals of the model; rather they were automatically inferred by the Simulink system. In total, we applied our analysis and test case generation tools to 6 selected small subsystems (Simulink, Stateflow, embedded Mathscript) and two larger subsystems, which mainly consisted of mode logic modeled by several Stateflow statecharts. For each of the models, we generated test vectors and test sequences (where applicable) with the goal of obtaining state, transition, and path coverage.

Figure 5 shows one of the analyzed Simulink models, which encodes some mathematical operations on quaternions with 5 inputs and 4 outputs. Besides various mathematical operations (e.g., inverse, square root), this model contains several if-then-else and merge blocks. With a range restriction on the inputs of $[-50, \ldots, 50]$, our tool generated 11 testcases (and in several cases our constraint solver gave warnings that it could not find solutions).

When executing these testcases on the corresponding generated code, only a code coverage of appr. 95% was obtained (analysis of the coverage revealed un-reachable code).

We analyzed several Simulink/Stateflow diagrams, ranging from the simple model in the previous section to two large SF diagrams, which contain embedded Matlab code and which synchronize by recursive calls (Figure 6) as well as other advanced Simulink/Stateflow features, like buses.

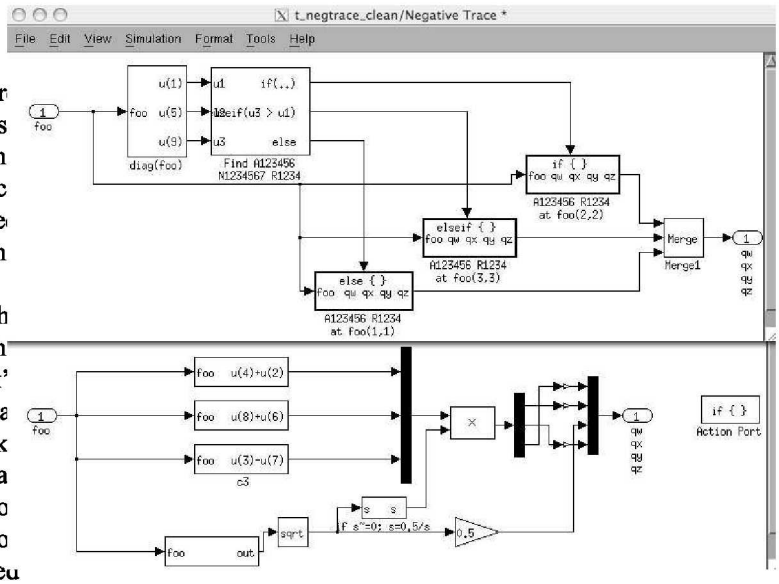

Figure 5. Simulink Model

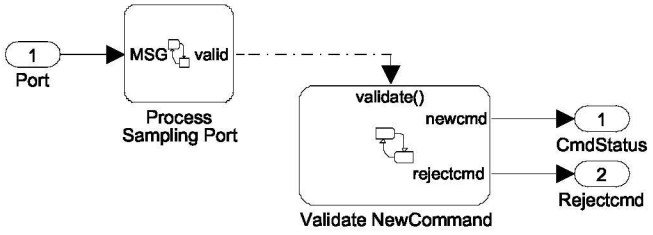

Figure 6. Synchronizing SF diagrams

For these models, we analyzed properties encoded as assertions; these assertions were derived from the informal model documentation (e.g., "On entry to state *Parachute*, assert that the Reaction Control System (RCS) control is disabled"). In addition to the problems related to integer overflow and unreachable code described above, our analysis revealed several errors in the models (e.g. assertion violation for the above property due to underconstrained environment).

It still remains for us to study the interaction between heterogeneous models and we are working with the developers of the code to define such interactions. However, we believe that our framework will provide good support for this study.

5. Related Work

The work related to this paper is vast and for brevity, we only highlight here some of the most relevant one.

The automatic generation of test cases from Simulink/Stateflow is the subject of several approaches. In particular, we have performed some experiments to investigate the applicability of two commercial tools,

T-VEC and Design Verifier, in the context of our case study. The tool T-VEC[3] is a commercial tool for testcase generation based on Simulink/Stateflow diagrams; it uses constraint solving technology.

We had been able to run the submodels from our case study through T-VEC. Although T-VEC supports a large subset of Simulink blocks and Stateflow, the translator has problems with processing large diagrams and complex statecharts, and unlike our framework, it does not support embedded Matlab. Furthermore, in order to produce test sequences, T-VEC has to work with multiple copies of the diagram, thus severely limiting its scalability.

Design Verifier is a tool by Mathworks, which is even closer integrated with the Simulink/Stateflow system. It also translates the models into a logic representation and the uses the Prover technology for analysis and generation of test cases. The current version has a relatively limited functionality, as it cannot handle nonlinear functions (e.g., sqrt, trigonometric functions), Simulink bus objects, or recursive functions. However, both T-VEC and Design Verifier are under active development, so it is expected that the above limitations will be soon overcome.

Another commercial tool, Reactis[4] is a toolset for model-based testing and validation of Simulink/Stateflow models. It uses random and heuristic search to exercise the behavior of the models to reach a certain coverage.

None of the above tools attempt to address the analysis of heterogeneous models.

There are many approaches for automatically verifying model-based specifications (e.g., [8]). The most closely related to ours are the ones targeting multi-formalisms template semantics and analysis tools (e.g., [1], [18]). However, such approaches target only multiple state machine representations. In the future, we plan to investigate the applicability of the template semantics in the context of our SC framework.

Model based generated test cases can be used to ensure that the translation (code generation) from the model to the code is working properly, as automatic code generators or manual implementation is not necessarily error free. Many approaches address the problem of making code generators and/or compilers trustworthy. Such approaches range from verifying model transformations [16] and verifying compilers/proof-carrying code [4] to instance-based verification, e.g., the AutoCert system [5].

6. Conclusion

We described a framework for model based analysis and test case generation based on Simulink/Stateflow and UML representations. We applied our framework to the analysis of various safety-critical parts of the flight code for NASA Orion. Our analyses and test cases revealed various deficiencies in the models (e.g., ambiguity in statechart transitions, potential integer overflows) as well as problems in the code generation phase (e.g., dead code).

Although this tool chain is currently used for Simulink/-Stateflow and UML models, the underlying framework for translation and analysis is very flexible and could be customized to handle other formalisms (e.g., multiple statechart semantics).

In the future, we plan to make the framework more robust and to apply it further to the analysis of heterogeneous models.

3. http://www.t-vec.com
4. http://www.reactive-systems.com

References

[1] J. M. Atlee and J. Gannon. State-based model checking of event-driven systems requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, 1993.

[2] Kennedy Carter. http://www.kc.com

[3] Choco Constraint Solver. http://choco.sourceforge.net.

[4] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. PLDI 2000*, pp 95–107, 2000. ACM Press.

[5] E. Denney and S. Trac. A software safety certification tool for automatically generated guidance, navigation and control code. In *IEEE Aerospace*, 2008. IEEE.

[6] Java Path Finder. http://javapathfinder.sourceforge.org.

[7] G. Hamon and J. Rushby. An operational semantics for Stateflow. In *Proc. 7th FASE*, vol 2984 *LNCS*, pp 229–243, 2004. Springer.

[8] D. Harel and A. Naamad. The Statemate Semantics of Statecharts. *ACM TOSEM*, 5(4):293–333, 1996.

[9] R. Heckel. Graph transformation in a nutshell. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/16.

[10] G. Karsai, A. Ledeczi, S. Neema, and J. Sztipanovits. The model-integrated computing toolsuite: Metaprogrammable tools for embedded control system design. In *2006 IEEE International Symposium on Computer-Aided Control Systems Design*, pp 50–55, 2006.

[11] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. In *Proceedings of the IEEE*, volume 91, pp 145–164, 2003.

[12] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[13] Control algorithm modeling guidelines using Matlab, Simulink, and Stateflow - Version 2.0. Mathworks Automotive Advisory Board.
http://www.mathworks.com/industries/auto/maab.html.

[14] P. Mehlitz. Trust your model - verifying aerospace system models with Java pathfinder. In *Proc IEEE Aerospace*, 2008.

[15] Guidelines for the use of the C language in critical systems. The Motor Industry Software Reliability Association. http://www.misra.org.uk/.

[16] A. Narayanan and G. Karsai. Using semantic anchoring to verify behavior preservation in graph transformations. *Electronic Communications of the EASST: Graph and Model Transformation 2006*, 4, 2006.

[17] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proc. ISSTA'08 (to appear)*, 2008.

[18] M. Pezzè and M. Young. Constructing multi-formalism state-space analysis tools. In *Proc. ICSE*, pp 239–249. ACM Press, 1997.