

# Model Based Analysis and Test Generation for Flight Software

Corina S. Păsăreanu, Johann Schumann, Peter Mehlitz, Mike Lowry  
NASA Ames Research Center, Moffett Field, CA 94035-1000  
NAME@nasa.gov

Gabor Karsai, Harmon Nine, Sandeep Neema  
ISIS, Vanderbilt University  
{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<sup>1</sup> 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<sup>2</sup> 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 8). 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 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

<sup>1</sup><http://www.mathworks.com>

<sup>2</sup><http://www.matrixx.com>

model based development frameworks.

We use model checking for the automated analysis of the models. While this technology has shown great promise of being cost-effective for finding defects in software, it suffers from the well known state-space explosion problem: for real-size models, the systematic analysis of all the model states runs quickly out of time or memory resources. Therefore, the model representation used by our framework should be abstract enough to exhibit a smaller state space, allowing thorough automated analysis, but detailed enough to reflect the model behavior (that will be later represented in the generated code). Furthermore, the model representation should be generic enough to allow representation of various model features represented in different formalisms.

To address these issues, our framework uses *two* levels of abstraction for representing the models: (i) One model-checker specific representation is designed specifically for modeling of various *statechart* formalisms. Its main purpose is to provide an abstract execution semantics for state machines, while aligning the diagram and representation state space as closely as possible. The representation is designed so that it can represent many representations (including Stateflow and UML) and it can be *adapted* to specific statechart dialects and associated execution semantics. (ii) The second representation is less abstract and closer to the actual production code (e.g., the code that is auto-generated from the models). This abstraction is used for representing mixed formalisms (i.e., both Simulink and Stateflow models). While the first abstract representation is more amenable to thorough, model-level verification, the second one can reveal problems that are more related to the actual production code. For example, this second representation can be used for the generation of test cases for testing the actual code.

The framework's characteristics are summarized as follows:

**Intermediate Common Representation.** The framework uses a common representation that is executable, analyzable, and restricted to a safe language subset (e.g., no dynamic memory allocation, no un-bounded arrays). Graph transformation [9] techniques are used for translating Simulink, and Stateflow models, while customized code generators are used to translate Embedded Matlab scripts and UML-style statecharts.

**Different Levels of Abstraction.** The framework uses two levels of abstraction as discussed above for the intermediate representation.

**Model Analysis.** Model analysis is performed with the Java PathFinder (JPF) [6] model checker. JPF is a highly customizable explicit state model checker with particular focus on finding bugs (concurrency related errors, run-time errors, assertion violations). Our framework uses JPF's built-in capabilities to address the state explosion problem,

e.g., partial-order and symmetry reduction, heuristic state space search, abstract state matching, etc. Furthermore, our framework uses JPF's statechart library (Section 5.2) for modeling different statechart formalisms.

**Test vector and test sequence generation.** Our framework uses and extends Symbolic (Java) PathFinder (SPF) [17] for model based test case generation. SPF uses symbolic execution and constraint solving techniques to generate test cases that achieve user specified test coverage (e.g., state, transition, path coverage). Test cases encode both the inputs and the expected outputs (the so called test oracles) to allow testing of the production code against the models. Since the models that we need to analyze perform extensive mathematical computations and make use of external libraries, SPF has been extended with precise modeling of Math functions (via specialized decision procedures [3]) and with mechanisms for modeling embedded code.

Our framework consists of a set of loosely coupled components and is easily *extensible* toward different modeling formalisms (e.g., a new flavor of UML) or different analysis tools. The analysis and test case generation components (including the generic statechart library) are available for download [6] (and we are making efforts to make the model transformation component also available).

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 3), followed by a detailed description of the framework components: model transformation (Section 4), model analysis (Section 5) and test case generation (Section 6). We then describe the application of our framework (Section 7), related work (Section 8) and conclusions (Section 9).

## 2. Modeling Languages and Associated Tools

Mathworks' Simulink is a data-flow oriented modeling tool, which has been specifically developed for the design and analysis of continuous- and discrete-time (control) systems. Hierarchical models contain operational blocks (e.g., mathematical functions, signal routing, integrators and delay elements), and links between the blocks denote the data flow. Whereas the underlying continuous-time semantics is suitable for simulation and analysis, most flight-code relevant models use a discrete-time semantics, where time is incremented in discrete, fixed time steps (e.g., in 12.5ms increments). Simulink diagrams can contain 'delay elements'

that store the state of the diagram, or the diagrams can be purely functional, i.e., stateless. For fixed time step models, efficient executable code can be generated, e.g., by using Mathworks' RealTime Workshop code generator. For our framework we assume that all models use fixed time steps.

In order to facilitate modeling of event-driven, state-based systems Stateflow can be used, which is integrated into Simulink. Stateflow is a variant of the Harel's statechart notation and provides hierarchical finite state machines with several extensions, e.g., history nodes. Stateflow models describe event-driven systems that undergo state transitions upon the occurrence of specific events. In the combined Simulink/Stateflow models events are typically generated by the Simulink models (e.g., when a signal reaches a certain threshold value). Stateflow models could also be used in a standalone manner, where they could describe complex controller logic. Stateflow semantics has been formally developed by Rushby et al. [7].

In general, traditional block-oriented Simulink models are used to model the continuous-time and mathematical components of the flight software (e.g., large portions of navigation and control software) and the system architecture, whereas Stateflow is used for discrete, state-oriented models, e.g., the guidance system. With the advent of advanced data structures (e.g., buses) and efficient code generators (RealTime workshop), large parts of modern flight software is developed in a model-based fashion and production code is auto-generated. Additionally, mathematical (i.e., matrix) algorithms could be implemented as scripts in Embedded Matlab (eML), which is a restricted and safe subset of the general Matlab scripting language, suitable for generation of safe code.

In contrast to Simulink/Stateflow, UML modeling systems offer more support for high-level and architectural design. Discrete, state-based components can be modeled using UML statecharts (which have a slightly different semantics from Stateflow diagrams). Modern tools with code generators are used for the modeling and automated generation of flight software.

### 3. Framework for Model Based Analysis and Testing

The overall framework for model based analysis and test case generation is illustrated in Figure 1. Various models (Simulink/Stateflow/eML and also UML statecharts) are translated into a common representation (a safe subset of Java) that is suitable for simulation and (exhaustive) analysis with the JPF model checking tool.

As mentioned, we have two abstract intermediate representations: one for different statechart formalisms (denoted "SC" in the figure) and a second one that is closer to production code (denoted ".java"). The SC representation factors

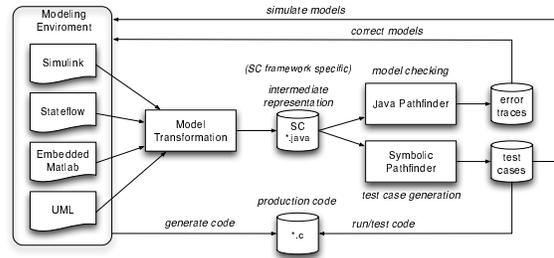


Figure 1. Model Based Analysis and Testing

the semantics of the model into a class hierarchy, that (i) follows the state hierarchy of the original statechart model, (ii) contains the 'actions' from the model in specific methods of the classes, and (iii) uses a generic, reusable execution engine that operates on the class hierarchy and implements the behavior of the corresponding model by calling the appropriate methods. The SC class hierarchy is a straightforward transcription of the statechart model. These two representations allow us to perform different kinds of analyses for the same models and provide additional confidence in the translation tools.

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 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.

## 4. Model Based Transformation

The model based transformation component of our framework is used to translate various models into a common Java representation that is suitable for analysis. In this paper, we focus on the the graph-based model transformation tools for translating Simulink/Stateflow/Matlab models. Other translation tools for xUML for Kennedy Carter (KC) [2] using a customized version of KC’s code generator is also being developed.

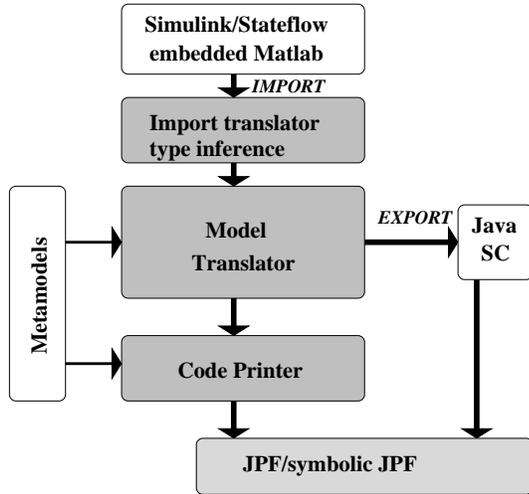


Figure 2. Model-Integrated Transformation Tools

### 4.1. Model Integrated Computing

At the heart of our model transformation component lies 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]). In the following sections, we will describe the principles of graph transformations used for this translation as well as the individual steps of the translation process.

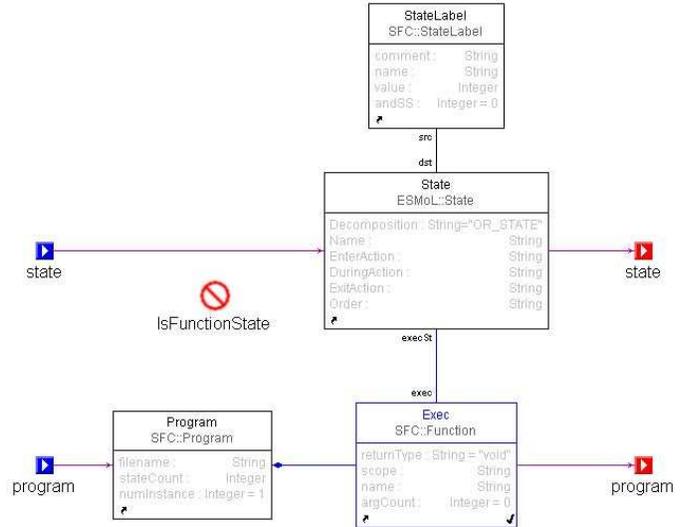


Figure 3. Example graph transformation rule

### 4.2. Graph Transformations and Rewriting

The translation program is a graph transformation [9] system that rewrites the input model, a graph, into the target model, which is another graph.

The transformation consists of explicitly sequenced rewriting rules (taking advantage of the features of the transformation language used) that traverse the input model depth-first (based on the containment hierarchy) and incrementally construct the target tree. Figure 3 shows an example for a transformation rule for Stateflow, which is constructed using the graphical GME editor. The rule applies to a State (in the input model) and a Program (which is part of the output model). The matches if the State has an associated StateLabel, and checks if the state is a 'function state' (i.e., not a pseudo-state like a history state) via the 'is-FunctionState' guard expression. If the rule matches and the guard evaluates to true then the rule creates an Exec function node and attaches it under the Program.

### 4.3. Simulink/Stateflow Translation

Our translation tool chain includes the following elements (see Figure 2): 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. The *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.

A different code printer can translate SFC into C code, which satisfies the requirements of the MISRA[15] guidelines, a coding standard for embedded C code.

A separate tool translates Stateflow models into an equivalent statechart (SC) representation in Java, suitable for analysis by the Java Path Finder (JPF). This translation is a simple text generation from the model, as the generated Java code here is a hierarchical collection of classes, representing the states, and methods representing the state transitions (see Section 5.2).

#### 4.4. Model Analysis during Transformation

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.

#### 4.5. Semantics of the Translated Models

We describe briefly here the semantics of the translated models, for the two abstract representations (.java and SC).

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

the 'discrete-time' with 'fixed-stepsize' assumption. Each Simulink subsystem is translated into two functions: one is used to initialize the 'state' (i.e., the delay elements) of the Simulink model, and another one that executes a single computational update step defined by the Simulink diagram. Similarly, the Stateflow models are translated into two main functions (one for initialization, one for the update), and for each state in the diagram we generate three functions that implement the necessary actions upon entering, exiting, and executing in the state, respectively. This approach is similar to the one used in RealTime Workshop.

For the SC representation, the SC class hierarchy is a straightforward transcription of the Stateflow model and the execution semantics is captured in the 'JPF SC engine' that operates on the class objects (see Section 5.2). The JPF SC approach gives a better analysis of the original model, while analyzing the generated production code can give assurances about the executable code that will be deployed in the final system.

## 5. Model Analysis

We use the Java PathFinder (JPF) model checker for model analysis. JPF also provides an adaptable, highly optimized, state chart (SC) framework that we use for translating models written using various state-chart formalisms. We describe here both JPF and the SC framework.

### 5.1. 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].

### 5.2. The JPF State-chart (SC) Framework

JPF includes a framework to analyze state machine models. The framework supports a broad range of state ma-

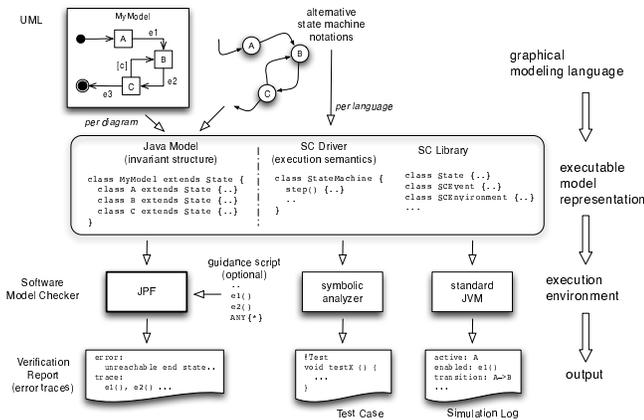


Figure 4. JPF's state-chart extension

chine types and execution environments. The major goal of this framework is to provide an execution semantics for a given state machine. This is achieved by translating the model into Java code that intuitively reflects the state machine structure, and builds upon a layer of extensible library classes that encapsulate concrete execution semantics. By choosing different driver applications, the resulting program can be run either under the JPF model checker, or in stand alone simulation.

The SC framework is implemented using JPF's extension mechanisms and it consists (Figure 4) (i) a translation scheme from state machine models into SC specific Java classes, (ii) basic building blocks like `State` and `SCEvents`, and (iii) a special `StateMachine` class that provides a *driver* that captures execution semantics of a given modeling language (like UML).

The model-to-Java translation scheme is primarily aimed at representing the original diagram structure in a readable form, i.e., without adding execution related overhead which obfuscates the hierarchical composition and transitional structure of the model. It translates diagrams into a set of nested classes. Each class represents a model `State`, defining methods for all events/triggers this state can react to. Transition guards are Java boolean expressions inside trigger methods, while various actions are ordinary Java expressions inside trigger methods. Details of the translation rules are outside the scope of this paper, and can be found in [14] (for UML state machines).

Albeit the framework is flexible in terms of state machine types and execution policies, it assumes one fundamental constraint: state composition and transitions are invariant (i.e., there is no dynamic creation of states and/or transitions). This is reflected by the structure of classes and the set of defined methods per class, which both cannot be changed at runtime. The main reason for this constraint is to enable deep analysis of the state machine structure by

means of the Java type system (using reflection). This in turn is key to avoiding verification related overhead in the resulting program representing the model.

hierarchical state composition
active state sets
begin/end states
actions when entering/exiting a state
events and transitions
transition actions
transition guards
domain specific state categorization
specialized and universal environments (event sources)
prioritized event queues and explicitly sent events
concurrency (independently executing machines)
event based synchronization
extensible execution semantics

Table 1. JPF SC features

Based on this assumption, the our framework supports numerous state machine features (Table 1), in particular:

**Hierarchical State Composition.** Complex models often exceed the size that can be displayed in one diagram. A standard technique to solve this problem is to use hierarchical composition into super- and sub-states, keeping the details of composed states in separate diagrams. UML is the most prominent example of a hierarchical state machine notation, but almost all other systems (like Stateflow) do support them. The SC framework implements state hierarchies by means of class nesting, leaving class inheritance to be available for model domain specific state categorization.

**Environments.** The framework allows to separate the reactive part of the system (the state machine) from the event generator (the environment). Environments can be either script based or use Java reflection to compute events. The script language is mostly declarative, but provides also simple control structures like bounded and unbounded loops. This mode is mostly useful for simulation, or to guide execution into specific states before deeper model analysis is started. The reflection based environment represents a *universal environment* that computes all events which could possibly cause transitions or actions, given a set of active states. This environment type is especially useful for model checking.

**Prioritized Event Queues.** Some state machine languages do not assume environments, but rely on explicit modeling of event generation. For such languages, the JPF framework provides a `sendEvent()` method, which is the front end for state-specific, prioritized event queues. It is even possible to mix environments and explicitly sent events, the latter ones taking priority over environment generated events.

**Concurrent State Machine Execution.** Following the concept of orthogonal regions in UML, the framework can model non-preemptive parallel execution of state machines. This is achieved by composite states with more than one initial state, executing within the composite state until all

regions are in their respective end states. While this is sufficient to implement orthogonal regions, it should be noted that this concurrency model is coarse, and would require additional verification capabilities to detect race conditions and deadlocks. This feature is therefore mostly used to model truly independent automata, or state machines that only communicate via sending events.

**Event Based Synchronization.** Some languages for concurrent state machines do require explicit synchronization mechanisms, which can be seen as dynamic guards over input alphabets of active states. For this purpose, the JPF framework includes a generic `receiveEvent(eventName)` that populates state specific wait sets, effectively suspending event processing of this state until an event from the wait set becomes available.

**Extensible Execution Semantics.** JPF’s state machine framework can be adapted to different execution semantics by extending its `StateMachine` class (the `step` method).

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.

JPF can be used for checking various properties of the translated statecharts, such as built-in Java properties (e.g., no unhandled exceptions), modeling language specific properties (e.g., check for ambiguous transitions), and more general safety properties, encoded as assertions or as safety monitors (using JPF’s Listeners).

## 6. Test Case Generation

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 multithreading and input pre-conditions. We describe symbolic execution, the Symbolic PathFinder tool, and its extensions for model based test case generation in more detail below.

```
[1] if ((pressure < pressure_min) ||
[2]     (pressure > pressure_max)) {
[3] ... /* abort */
      } else {
[4] ... /* continue */
      }
```

**Figure 5. Example for symbolic execution.**

### 6.1. Symbolic Execution

Symbolic execution [12] is a form of 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. The paths followed during the symbolic execution of a program are characterized by a *symbolic execution tree*.

To illustrate the difference between concrete and symbolic execution, consider the example in Figure 5. The code checks if the value of *pressure* (input variable *pressure*) is within *min* and *max* allowed values (input variables *pressure\_min* and *pressure\_max*). In concrete execution (e.g. testing) one executes the code on given concrete inputs. For example, for *pressure* = 460, *pressure\_min* = 640, *pressure\_max* = 960, only one path through the code will be executed, corresponding to the first disjunct in the `if` statement being true. In contrast, symbolic execution starts with symbolic input values (*pressure* = *Sym1*, *pressure\_min* = *MIN* and *pressure\_max* = *MAX*). Symbolic execution will analyze three paths through the program and it will generate three path conditions, according to different possibilities in the code:

$$\begin{aligned} PC_1 &: Sym_1 < MIN, \\ PC_2 &: Sym_1 > MAX, \\ PC_3 &: Sym_1 \geq MIN \wedge Sym_2 \leq MAX \end{aligned}$$

Concrete values for the inputs that satisfy (“solve”) the path conditions are then found with the help of a constraint solver and those solutions are used as concrete test inputs that are guaranteed to give full path coverage for this code.

## 6.2. Symbolic PathFinder

Symbolic PathFinder implements a non-standard interpreter 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).

## 6.3. Test Vectors and Test Sequences

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 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.

Per default, Symbolic PathFinder prints the generated test vectors and test sequences in a tabular HTML format and as text. 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.

## 6.4. Modeling Math Functions and External Function Calls

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 can be used to handle native code embedded in the models.

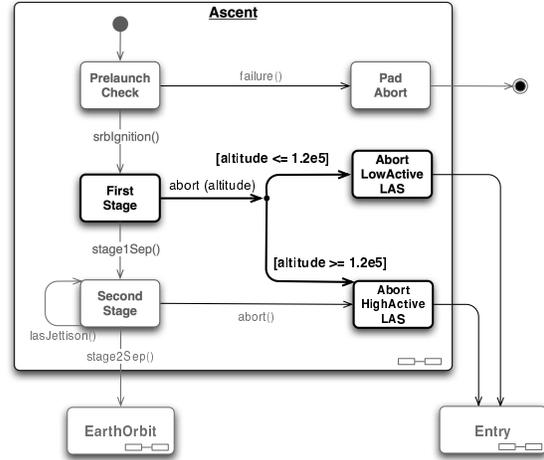


Figure 6. Model of the *Ascent* and *EarthOrbit* flight phases of a spacecraft

## 6.5. Example

We illustrate model based test case generation using the state-machine model of the *Ascent* and *EarthOrbit* flight phases of a spacecraft (Figure 6), 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 amenable to simulation testing (that would fix the the event sequence a priori), 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.

## 7. Applications

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

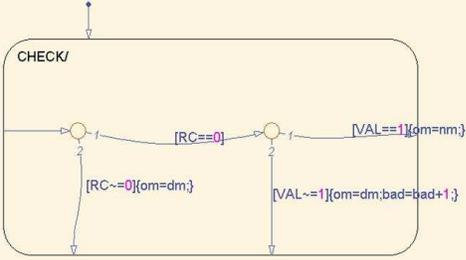


Figure 7. Stateflow example

### 7.1. Analysis of a Sampling Port

We illustrate some of the features of our framework with a simple Stateflow example (see Figure 7). 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 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 2 shows the test cases that are generated to achieve branch coverage.

In order to run the test cases from Table 2, 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 8: 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.

Table 2. Generated test cases for model in Figure 7

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 8. Measuring coverage on generated code

### 7.2. Analysis of Flight Software

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

The entire GN&C software has been modeled using Mathwork’s Simulink/Stateflow system, and large portions of the flight code are automatically generated using Mathwork’s RealTime Workshop. This model has a highly hierarchical structure and contains Stateflow diagrams, Simulink blocks for continuous calculations and sig-

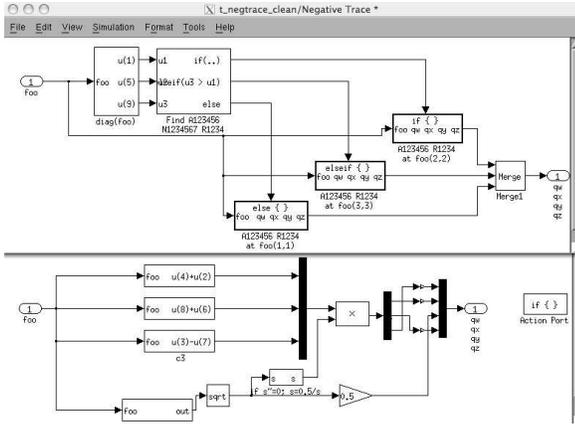


Figure 9. Simulink Model

nal routing, as well as some embedded Matlab scripts. The entire system consists of roughly 25,000 Simulink blocks, 100 Stateflow diagrams of 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 9 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  $[-50, \dots, 50]$ , 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

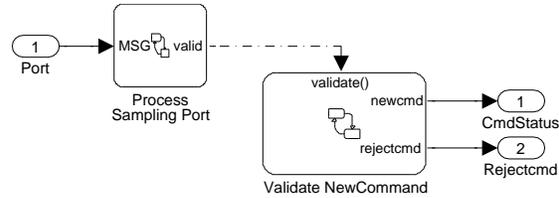


Figure 10. Synchronizing SF diagrams

large SF diagrams, which contain embedded Matlab code and which synchronize by recursive calls (Figure 10) as well as other advanced Simulink/Stateflow features, like buses.

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.

## 8. 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<sup>3</sup> 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

<sup>3</sup><http://www.t-vec.com>

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<sup>4</sup> 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].

## 9. 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.

## 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 Statechart 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. Mehltitz. 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. Mehltitz, 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. ISSA'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.

<sup>4</sup><http://www.reactive-systems.com>