Formal Validation of Fault Management Design Solutions

Corrina Gibson
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91011
1-626-660-5107
corrina.l.gibson@jpl.nasa.gov

Robert Karban
European Southern Observatory
Karl-Schwarzschild-Str. 2
85748 Garching, Germany
+498932006542
rkarban@eso.org

Luigi Andolfato
European Southern Observatory
Karl-Schwarzschild-Str. 2
85748 Garching, Germany
+498932006796
landolfua@eso.org

John Day
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91011
1-818-354-2026
john.c.day@jpl.nasa.gov

ABSTRACT
The work presented in this paper describes an approach used to develop SysML modeling patterns to express the behavior of fault protection, test the model’s logic by performing fault injection simulations, and verify the fault protection system’s logical design via model checking. A representative example, using a subset of the fault protection design for the Soil Moisture Active-Passive (SMAP) system, was modeled with SysML State Machines and JavaScript as Action Language. The SysML model captures interactions between relevant system components and system behavior abstractions (mode managers, error monitors, fault protection engine, and devices/switches). Development of a method to implement verifiable and lightweight executable fault protection models enables future missions to have access to larger fault test domains and verifiable design patterns. A tool-chain to transform the SysML model to jpf-Statechart compliant Java code and then verify the generated code was established. Conclusions and lessons learned from this work are also described, as well as potential avenues for further research and development.

Categories and Subject Descriptors
D.2.2 [SOFTWARE ENGINEERING]: Design Tools and Techniques – State diagrams
D.2.3 [SOFTWARE ENGINEERING]: Software/Program Verification: Assertion checkers, Model checking, Formal methods, Validation

General Terms
Design, Theory, Verification.

Keywords
Model Checking, Java Pathfinder, SysML, Statechart, Fault Protection

1. INTRODUCTION
The Soil Moisture Active Passive (SMAP) will provide global measurements of soil moisture and its freeze/thaw state. These measurements will be used to enhance understanding of processes that link the water, energy and carbon cycles, and to extend the capabilities of weather and climate prediction models. SMAP data will also be used to quantify net carbon flux in boreal landscapes and to develop improved flood prediction and drought monitoring capabilities [8].

Highly complex systems, such as the SMAP Fault Protection system [2], are difficult to develop, test, and validate using traditional methods – Fault protection design has been prone to human error and subject to limited multi-fault, multi-response testing. Traditionally, responses are designed individually because it is not feasible for humans to incorporate all combinations of fault protection events in design or test without a model. It is also expensive to use high fidelity test beds, limiting the scope of the possible combined-response tests that can be performed. To explore new model-based methods of testing and validating fault protection, SMAP Fault Protection logical designs were used to architect a representative SysML behavioral model that was used to exploit fault injection testing and model checking capabilities. Model checking provided a basis for checking fault protection design against the defined failure space and enabled validation of the logical design against domain specific constraints (for example, during ascent the receiver should be on and the transmitter should be off).

The model is transformed to run simulations, create artifacts to be model checked, and to produce the final software implementation. In order to gain confidence in the validation and verification of the model based design and its implementation the following questions must be addressed:

- Does the model represent the system?
- Do the generated artifacts for model checking represent the model?
- Do the generated artifacts for model checking represent the final software system implementation?

In the context of this paper, simulation is used to validate the model against requirements. It is also assumed that the generated artifacts for model checking represent the model. However, this could be mitigated by comparing the simulation results with the execution of the generated code for model checking. Finally, the code used for model checking is not part of the final software system implementation. Simulation of the model caught (initial modeling and design translation) errors and provided the ability to inject a variety of inputs to test many aspects of the model, leading to confidence in the logical design of the model. It became clear, as more error monitors and responses were added to the model, that it would not be possible to manually run simulations for all of the possible sequences of the model [1] – a model checker is necessary to formally and exhaustively verify the model for all possible sequences.

2. TOOLCHAIN
The tool-chain consists of: a UML modeling tool (MagicDraw 17.0.4) with SysML plugin and simulation environment (Cameo Simulation Toolkit 17.0.4 which is based on Apache SCXML Engine 0.9), a model-to-text transformation tool (COMODO), and a model checker (JPF6 and JPF7).

MagicDraw is used to model and represent the system in terms of collaborating Statecharts according to SysML 1.3. The model is
exported in UML2 XMI 2.x format and then processed by COMODO.

COMODO [4] is a platform independent tool to generate text artifacts from SysML/UML models using Xpand/Xtend technology. For example COMODO can transform SysML State Machine models to Java code which is compliant with JPF’s Statechart project (jpf-Statechart) and the final software implementation for different platforms.

Statecharts XML (SCXML) is a W3C notation for control abstraction defining the syntax and semantic for Statecharts execution [7]. Apache Commons SCXML is one implementation of SCXML.

3. MODELING FOR MODEL CHECKERS
An overview of SMAP model is provided in the following figure.

Model checking state space is a valuable indicator for the complexity of the model. Model checkers are computation and memory intensive. After initial model checking runs were found to take days to exhaustively check a small subset of the SMAP Fault Protection system, it became apparent that model patterns should aim at decreasing the state space. In an attempt to reduce the state space, adjustments were made to the model architecture and the Statechart representation of the fault protection system’s response tiers and response queue.

Response tiers define the sequence of actions performed by fault protection system responses. Each subsequent tier of a given response attempts to mitigate a fault with different sets of actions. If a response has more than one tier, subsequent tiers will not be performed until prior tiers fail to mitigate the fault. The error monitor that detects the fault must be re-tripped between each subsequent tier. In cases where all tiers are executed and the fault still exists, the response resets and re-executes its tiers (assuming the response has not been masked).

When system responses are tripped by error monitors, they are placed in the fault protection response queue based on priority: High priority responses are placed in the front and low priority responses are placed in the back. A set of activation rules evaluates the response in the front of the queue and either allows the response begin executing its tiers or, if the activation rules do not pass, denies response execution and places the response back into the queue.

The adjustments made to the model in order to reduce the state space include: 1) Use of composite Statecharts with orthogonal regions to define each behavior. It was verified that the state space of a Statechart with orthogonal regions is equivalent to a flat state machine since Statecharts are only a notational enhancement of the state machines. However the Statechart representation is more compact and the model is more readable; 2) Enumerations were used for incrementing response tiers rather than integer values that are incremented (the value of the tier variable is set in the effect (opaque behavior) of the transitions that lead to each tier’s steps, instead of the behavior of the tier state. Then guards on those transitions ensure the correct tier is transitioned to based on the tier variable value. This method, as opposed to an incrementor, prohibits infinite counting (no tier ++) in JPF. An incrementor essentially adds new states for each iteration in JPF so the verification will fail due to memory shortage; and 3) Guards were placed on transitions wherever possible. The difference in computation time for a model with few guards compared with another model of triple size with many guards was found to be a factor of 5000 (see Table 2). Adding guards to most transitions reduces the complexity of the system to be checked, limiting the number of paths making it quicker to check.

The following simple Statechart with orthogonal regions provides a straightforward example of a model checking application. The correctness property inserted is to ensure that state B and state E are never active at the same time: assert(inState(B) && !inState(E)).

Model checking state space is a valuable indicator for the complexity of the model. Model checkers are computation and memory intensive. After initial model checking runs were found to take days to exhaustively check a small subset of the SMAP Fault Protection system, it became apparent that model patterns should aim at decreasing the state space. In an attempt to reduce the state space, adjustments were made to the model architecture and the Statechart representation of the fault protection system’s response tiers and response queue.

Response tiers define the sequence of actions performed by fault protection system responses. Each subsequent tier of a given response attempts to mitigate a fault with different sets of actions. If a response has more than one tier, subsequent tiers will not be performed until prior tiers fail to mitigate the fault. The error monitor that detects the fault must be re-tripped between each subsequent tier. In cases where all tiers are executed and the fault still exists, the response resets and re-executes its tiers (assuming the response has not been masked).

When system responses are tripped by error monitors, they are placed in the fault protection response queue based on priority: High priority responses are placed in the front and low priority responses are placed in the back. A set of activation rules evaluates the response in the front of the queue and either allows the response begin executing its tiers or, if the activation rules do not pass, denies response execution and places the response back into the queue.

The adjustments made to the model in order to reduce the state space include: 1) Use of composite Statecharts with orthogonal regions to define each behavior. It was verified that the state space of a Statechart with orthogonal regions is equivalent to a flat state machine since Statecharts are only a notational enhancement of the state machines. However the Statechart representation is more compact and the model is more readable; 2) Enumerations were used for incrementing response tiers rather than integer values that are incremented (the value of the tier variable is set in the effect (opaque behavior) of the transitions that lead to each tier’s steps, instead of the behavior of the tier state. Then guards on those transitions ensure the correct tier is transitioned to based on the tier variable value. This method, as opposed to an incrementor, prohibits infinite counting (no tier ++) in JPF. An incrementor essentially adds new states for each iteration in JPF so the verification will fail due to memory shortage; and 3) Guards were placed on transitions wherever possible. The difference in computation time for a model with few guards compared with another model of triple size with many guards was found to be a factor of 5000 (see Table 2). Adding guards to most transitions reduces the complexity of the system to be checked, limiting the number of paths making it quicker to check.

The following simple Statechart with orthogonal regions provides a straightforward example of a model checking application. The correctness property inserted is to ensure that state B and state E are never active at the same time: assert(inState(B) && !inState(E)).
When JPF was run, it instantly found a counterexample to the assertion and output the error trace and performance statistics shown in the following figure. Following trace #1, error #1 was found because trace #1 defines an existing path that leads to B and E being active together. The statistics show that there is no elapsed time needed to perform this very basic model-checking task.

The initial attempt at architecting the Fault Protection model did not consider the limitations of the tool chain. The initial model used complex elements and diagrams such as sequence diagrams, nested logic (hidden If statements), complex state machine model elements (e.g. decision nodes), and global variables. Simulation artifacts began overtaking the model because of the complex elements and nested logic. Additionally, the current version of the SysML to Java code transformation tool is limited to interpreting elements and nested logic. The challenge was to find an adequate representation of the system and properties to be checked.

The fundamental drivers of the modeling task are patterns and practices that lead to efficient model checking. Efficient in the sense that the state space is reduced; the most important factor in exhaustively checking a model within a reasonable time as memory and computation requirements grow with the state space. The goal is to make checking of large system models a standard practice that is accessible to a wider audience of engineers, is automated and does not require highly specialized skills in order to produce an optimal representation of the system and properties to be checked.

Currently, JPF ignores internally generated signals so there is no limit of the signals that can be sent from a state. JPF inherently checks all combinations of events so no cases of the modeled behavior are missed due to internally generated signals being ignored in JPF, but many paths irrelevant for the specification of the system behavior are explored. Guards are critical in keeping the state space limited, but the ability for JPF to interpret signals would be ideal for limiting the state space even more. Thus, a goal to substantially reduce the state space is to further develop jpf-Statecharts to take into account the internal signals that are sent in behaviors. One possible solution is to add boolean logic for signals as guards on transitions so JPF doesn’t explore impossible paths (Note: This solution can only be done at run-time and the model checker would have , for every state configuration , only a limited number of events available when generating events during model checking).

This demonstrates good modeling for model-checking practice: Use guards to encode the knowledge of the model about internal events. Without guards, the model checker would exhaustively explore the complete state space, whereas in the final run-time system the entire state space would never be checked since only a limited number of events will occur at any given moment in time. Adding guards that limit the number of possible transitions results in a two-fold advantage. First, the state space that is explored is drastically reduced, decreasing the time and memory used for model checking. Second, including the guards in the final implementation ensures the system will never end up in an “unexpected” state if an out of order event occurs. In both cases the complexity of the system is reduced, allowing the amount of involved testing to be reduced and off-nominal behavior to be limited (it is worthwhile to investigate how the introduction of model checking affects traditional test strategies). It is important to note that guards also help when doing an initial validation and verification of the model using simulation. For example, In multiple circumstances, the model could not be simulated after adding a guard, which pointed out a model error or bug. The guards also validate the modeler’s assumption on the behavior of the model.

Multiple versions of the fault protection system’s response queue were modeled in order to find a pattern that could be executed and model checked. The response queue was initially modeled using opaque behaviors with ‘if inState’ code inside of states – this actually represents implicit Statechart states that cannot be considered by jpf-Statecharts but only by JPF-Core, therefore increasing the state space. Now the response queue is modeled explicitly with multiple nested states and ‘inState’ guarded transitions. It was found that both methods work for simulation and JPF (because JPF can interpret the code inside of opaque behaviors); however because the latter method is explicit (it does not have hidden guards in the code) and limits the state space with guarded transitions, it was chosen for the queue pattern.

4. MODEL TRANSFORMATION

During the model-checking phase of this project, the tool chain from the SysML model to Java Pathfinder (JPF) model checker was extended to support collaborating Statecharts, guards on composite states, and opaque behaviors (transformed as commented code).

The challenge was to find an adequate representation of the SysML model in jpf-Statecharts terms. In SysML the behavior of the system is described by a set of collaborating Statecharts interacting via SysML ports, sending signals to each other, and referencing other Statecharts’ states in guarded transitions. In jpf-Statecharts, states are represented by Java classes, nested states by nested Java classes and transitions by class methods. Parallel regions are mapped to additional initial nodes and not to a separate Java classes.

COMODO was modified to map the behavioral part of the SysML model to a single system Statechart containing one orthogonal region for each collaborating Statechart. Thus, the guards that reference states in separate Statecharts in the SysML model will reference states in an orthogonal region in the transformed model.

Due to the property that events are broadcast to all orthogonal regions in a Statechart, a merged Statechart of orthogonal regions accurately represents a collection of collaborating Statecharts.
Consequently, the modeler must ensure that distinct Statecharts do not use the same signal to trigger transitions unless they intend for all of the transitions in the model with that signal to be triggered independently of where the signal is intended to be consumed. Additionally, the inState guards require the fully qualified name so JPF knows where, in its merged Statechart, to find individual states.

The following chart illustrates the process of model checking from SysML model to JPF’s result.

The current COMODO transformation to jpf-Statechart requires manual intervention in the generated Java code. For example any code from the model’s opaque behaviors are manually converted to Java. Ports are commented because there is no concept of interfaces between state machines in the transformed, merged Statechart. Different syntax required to specify guards in MagicDraw/Cameo and in jpf-Statecharts currently imposes a limitation therefore the guard transformation has to be done manually. The properties to be checked (in the form of assertions) are also added manually and have, at the moment, no representation in the SysML model.

If Java is the target platform for the final software implementation and its structure is based on Statechart semantics, the JPF model checker can be run on the final production code. For rigorous validation and verification of the system it is both efficient and less prone to error if the model checker supports the final implementation language. If an intermediate representation is checked or the model is transformed to a model checking specific language (for example Promela), then one would have to verify that the model checked representation is consistent with the final implementation.

5. **SYSTEM’S PROPERTIES**

The following assertions are a subset of the properties to check that were input into the transformed model (SMAPFaultProtectionBehavior_v37) prior to running JPF. The goal of model checking is to ensure that none of these logical statements can be violated in the fault protection design. This small set of assertions was formulated from rules stated in the SMAP Mode Manager Functional Design Description (FDD) [3], Mode Manager configuration tables, and SMAP Fault Protection FDD.

### Table 1: Example Model Checking Assertions

<table>
<thead>
<tr>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>If in Idle Acs mode state, then in Test system mode state OR Prelaunch system mode state</td>
</tr>
<tr>
<td>If in Safe RCS response state, then in Activation rules state</td>
</tr>
</tbody>
</table>

Assertions are a simple way to check logical behavior of the system when executing a entry/so/exit behavior. Assertions are limited to expressions that are valid at a given point in time in the current state configuration. It would be useful to check temporal expressions that can be formulated for the overall Statechart and can contain operators to express temporal constraints.

With the currently available ‘isInActiveStates’ operator of jpf-Statecharts, assertions on the current state configuration can be formulated. However, we cannot assert certain paths or sequences of Statechart execution, in particular when guards are used. If a Statechart is created without guards the sequence of states is determined by its transitions. The problem arises with multiple guarded transitions that allow a state to be reached in different ways. The objective is to validate that, in certain conditions, a state can only be reached via a certain path. If the transitions are guarded the path cannot be simply determined by the transitions. In order to be able to assert certain paths, an additional operator, ‘waslnActiveStates’ can be introduced that queries the previous state configuration and enables assertions to be made about the path that leads to a certain state. For example, in Figure 6, S3 can be reached from S5 and S6 via transitions. A ‘waslnActiveStates’ assertion could be put in a behavior of S3 that ensures that the previous state was either S5 or S6, depending on some other condition like being in another state in yet another region.

Asserting statements manually into the generated Java code, it is preferred to have the assertions and potential temporal logic expressions modeled as constraints in the SysML model and generate them during the model transformation. This would ensure that all relevant information is in a single place, as suggested by a model driven approach.

**Figure 5: Model Checking in Java Pathfinder**

**Figure 6 Multiple Paths to reach a state**
6. Model Checking Results

The following results show the performance of JPF model checkers for different sized models.

<table>
<thead>
<tr>
<th>Config.</th>
<th>Notes</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core2 Quad CPU Q9400 @ 2.66GHz 2.67GHz, 8GB, Windows 7, 64Bit, Java 1.6.0_45 64Bit, JPF6</td>
<td>Covered Statecharts: SysRop, RW1A1, ErMon, mode manager. No errors detected.</td>
<td>Elapsed time: 0.000719 seconds; state: new=49728, visited=17970, choices=45189, max live=2319, gc-cycles=221189 instructions: 1853380973 max memory: 335MB loaded code: classes=252, methods=1307</td>
</tr>
<tr>
<td>Intel Core2 Quad CPU Q9400 @ 2.66GHz 2.67GHz, 8GB, Windows 7, 64Bit, Java 1.6.0_45 64Bit, JPF6</td>
<td>Covered Statecharts: SysRop, RW1A1, ErMon, mode manager.</td>
<td>Elapsed time: 0.00028 seconds; state: new=12357, visited=21281, backtracked=33638, end=0 search: maxDepth=35, constraints hit=0; choices=18464 (signal=0, lock=1, refId=1792) instructions: 7349849581 backtracked=2552715791 end=0 search: maxDepth=3391, constraints hit=0; choices=1132462106. Max memory: 662MB; Loaded code: classes=220, methods=1097</td>
</tr>
<tr>
<td>Intel Core2 Quad CPU Q8400 @ 2.66GHz 2.67GHz, 8GB, Windows 7, 64Bit, Java 1.6.0_45 64Bit, JPF6</td>
<td>The SMAP model v.37,</td>
<td>Elapsed time: 0.000028 seconds; state: new=12357, visited=21281, backtracked=33638, end=0 search: maxDepth=35, constraints hit=0; choices=18464 (signal=0, lock=1, refId=1792) instructions: 7349849581 backtracked=2552715791 end=0 search: maxDepth=3391, constraints hit=0; choices=1132462106. Max memory: 662MB; Loaded code: classes=220, methods=1097</td>
</tr>
</tbody>
</table>

*Note: There is a significant difference when checking the SMAP model with JPF 6 and 7. This requires further investigation.

An estimate of the worst case state space was computed for Statecharts given in the above table simply by building the product of the number of leave states of every region. This means that every state in a region would have a transition to any other state in the same region.

7. JVM Configuration

There are many applications where Java is used in High Performance Computing (HPC) with often only marginal reductions in performance w.r.t C or C++ [6]. However, the Java Virtual Machine (JVM) requires some application specific tuning in order to exploit its best performance. In particular in relation to Just In Time compilation and the selected Garbage Collector strategy. Model Checking can be considered as an HPC application, in particular for large models with large state spaces.

At the moment the run-time options for Java used for JPF are:

djava -server -XX:+UseParallelOldGC -XX:CompileThreshold=10000 -XX:+PassiveLibCf -XX:+UseCompressedOops -XX:MaxNewSize=8g -XX:MaxNewSize=8g -Xmx26g -Xms26g -jar jpf-core/build/RumJPF.jar +shell.port=4242 ..SUT.jpf -show -log

There was an attempt to run JPF on JPL’s Castor supercomputer which consists of multi Intel Itanium CPUs and a Linux operating system. Already with a small model, which took about 10 hours on a Windows quad core machine, the execution on Castor took about 40 hours. The main reason (which can also be found in various internet forums) seems to come from a competition between the Java garbage collector and the Linux paging mechanism.

8. CONCLUSIONS AND FUTURE WORK

In addition to increasing the capabilities of the transformation tool to make the SysML to text code process more automatic, improvements to JPF-SC have been noted that would speed up the model checking verification time as well as allow for a wider range of SysML models to be checked.

The Cameo Simulation Toolkit, used to execute the model in this project uses the Apache implementation of SCXML. jpf-Statechart uses its own semantics for Statecharts and would need to be rewritten to be compliant with SCXML so the suite of tools used for this project is better aligned. jpf-Statechart should also be 64 bit compliant, have parallelized (multi-threaded) state space exploration capabilities so multi-core machines can be used efficiently for model checking. Moreover, strategies to reduce the state space (using information on internal and external events) shall be investigated, as well as the capability to validate specific paths and sequences in the Statechart execution (wasInActiveStates). jpf-Statecharts should also support temporal logic expressions which would allow to validate future state configurations. For example, if the Statechart enters a Failure Mode state it shall eventually end up in a Response state. Finally a catalog of modeling patterns has to be created to support efficient model checking.

9. ACKNOWLEDGMENTS

G. Holzmann, G. Watney[5], C. Havelund, P. Meakin, and P. Mehltz. This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

10. REFERENCES


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Conference/Meeting Month – Year, City, State, Country.

Copyright 2010 ACM 1-59593-084-8/10/01...

5