

# Trust Your Model - Verifying Aerospace System Models with Java™ Pathfinder

Peter C. Mehltz  
Perot Systems Government Services

NASA Ames Research Center, M/S 269-2  
Moffett Field, CA 94035-1000  
(650) 604-1682  
pcmehltz@email.arc.nasa.gov

*Abstract*—Model Driven Development (MDD) is rapidly becoming a mainstream practice for the development of complex aerospace systems. UML has emerged as the de facto standard for modeling languages, supporting a wide range of modeling aspects and refinement levels. As a consequence, models can easily become too complex for manual verification and simple static analysis.<sup>12</sup>

This paper describes an approach to using the Java™ Pathfinder (JPF) software model checker to systematically verify UML state charts. While state machines in general are amenable to model checking, embedded actions and guards in UML state charts are not, since they require execution and analysis of a full programming language to cover the whole model behavior. Many UML development systems can produce code from diagrams, but this code is usually aimed at production systems, and is not suitable for software model checkers.

Our approach is based on a specific translation scheme from UML state charts into Java code that (a) is highly readable, (b) shows close correspondence between diagram and program, (c) provides a 1:1 mapping between model and program states, and (d) imposes no restrictions about aspects and actions that can be modeled.

We have demonstrated scalability and efficiency of this approach on hierarchical state charts with up to 1000 states, including verification of incomplete models by means of guided model checking. This paper provides an overview of the method based on an exemplary spacecraft model.

## TABLE OF CONTENTS

<b>1. INTRODUCTION</b> .....	<b>1</b>
<b>2. TOOLCHAIN</b> .....	<b>2</b>
<b>3. UML TO JAVA TRANSLATION</b> .....	<b>3</b>
<b>4. UML MODEL CHECKING WITH JPF</b> .....	<b>5</b>
<b>5. MODEL PROPERTIES</b> .....	<b>7</b>
<b>6. GUIDED MODEL CHECKING WITH SCRIPTS</b> .....	<b>9</b>
<b>7. CONCLUSIONS AND OUTLOOK</b> .....	<b>11</b>
<b>ACKNOWLEDGEMENTS</b> .....	<b>11</b>
<b>REFERENCES</b> .....	<b>11</b>
<b>BIOGRAPHY</b> .....	<b>11</b>

<sup>1</sup> 1-4244-1488-1/08/\$25.00 ©2008 IEEE

<sup>2</sup> IEEEAC paper #1197, Version 2, Updated October 24, 2007

## 1. INTRODUCTION AND BACKGROUND

### *The MDD approach – benefits and caveats*

Since its introduction by OMG in 2001, Model Driven Development (MDD [1]) has found widespread use in Aerospace Engineering. Originally meant to be a software development approach, it turns out to be especially useful for the design of complex missions. Since it can cover both hardware and software functionality, MDD is suitable for creating complete mission models that not only form a basis for later development phases, but also enable validation of mission concepts long before the actual system is built.

As more projects rely on MDD as a crucial development tool, and models become larger, there is increased need for consistency of the model itself. This requires a formal, standardized notation for all modeled aspects. The Unified Modeling Language (UML [2]) provides such a set of graphical notations that try to meet this requirement by striking a balance between well-defined constructs and flexible extensions that target ease-of-use. While UML does define diagram languages, it does little to specify what checks have to be performed to ensure that diagrams are consistent. This task is left to 3<sup>rd</sup> party UML development system vendors, is usually focused on simple static analysis, and has always been subject to vendor specific interpretation. This situation is particularly insufficient to verify consistency of behavioral models.

### *UML Statecharts – Power and Pitfalls of Embedded Code*

UML state diagrams are essentially Harel statecharts [3], supporting

- hierarchical composition of states
- some notion of concurrency (orthogonal regions)
- completion-, signal- and time- triggers
- entry and exit actions for states
- guard expressions and actions for transitions

The embedded code in guards and actions makes it very convenient to capture non-statechart logic. Together with hierarchical composition it forms the basis for scalability of

UML statechart models, providing efficient mechanisms for refinement.

The downside of actions and guards is that model behavior might not be comprehensible anymore by looking at the diagram – the code has to be “executed” in order to understand the dynamics of the model.

Consider the following example: a spacecraft earth orbit flight phase is modeled by a composite state *EarthOrbit*:

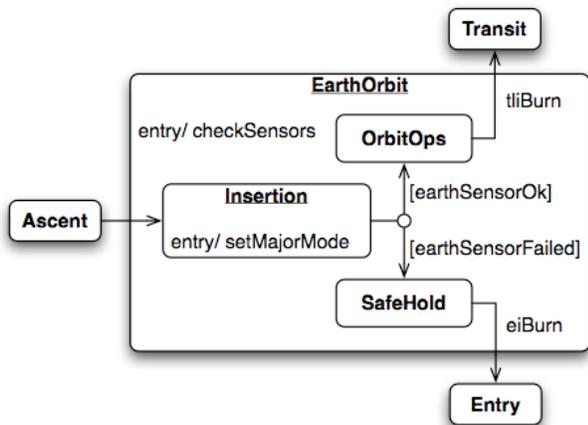


Figure 1 -- EarthOrbit Model

From the diagram perspective, this model looks fine – as soon as the *EarthOrbit* state is entered, the earth sensor is checked. If the sensor works, the system automatically transitions through the Insertion state into the *OrbitOps* state, which is the interface for the next nominal flight phase. If the earth sensor failed, the spacecraft cannot enter a sustainable orbit and proceeds into the *SafeHold* state, which is the interface for an off-nominal entry.

The problem is that the *checkSensors()* and *setMajorMode()* entry actions can represent arbitrarily complex functions. Assuming that *checkSensors()* is modeled so that the earth sensor failure state is treated as a random variable, a software model checker should explore both possible transitions out of the *Insertion* state.

```

checkSensors() { ...
    if (random.nextBoolean())
        failures.add(earthSensor)
    ...
}
  
```

However, the *Insertion* state of the example also features an entry action *setMajorMode()* which might be subsequently incorrectly refined so that it actually reverts the effect of *checkSensors()*, thus causing the system to always enter the *OrbitOps* state no matter what the simulated outcome of the sensor check was:

```

setMajorMode() {
    ... resetSubsystems() ...
}
...
resetSubSystems() {
    ... resetFailures() ...
}
  
```

A typical consistency requirement for our model might be that all its states are reachable, but in order to check for compliance, we have to try all possible combinations of

- external stimuli (even sequences)
- internal reactions (action outcomes)

With growing model complexity, this might quickly become infeasible to perform manually, especially with nested states or actions. Using our approach, the check can be fully automated:

```

>jpf gov.nasa.jpf.StateMachine
+jpf.listener=.tools.sc.Coverage
+sc.required=earthOrbit...

===== error #1
required earthOrbit.safeHold NOT COVERED
...
  
```

This demonstrates only one possible model property (reachability), but shows the value of systematically exploring all possible model behaviors (including code). The rest of this paper describes the steps and tools required to achieve this level of automated verification, and how the approach can be applied to other types of properties.

## 2. TOOLCHAIN

In general, our approach requires three steps to verify UML statecharts:

- (1) translate the UML model into a Java program, using a specific framework and translation scheme
- (2) choose model properties to verify, and configure verification tools accordingly
- (3) optionally provide a guidance script that represents the environment of the model (event sequences)

Figure 2 represents the flow of data. Step (1) is mandatory, since our model checker only works on Java bytecode, i.e. binary programs. Step (2) can vary in terms of effort, from using only generic non-functional properties like “no unhandled exceptions” that do not require any configuration, explicit assertions in the model code (representing safety properties), all the way up to extending the model checker with classes that implement functional, domain specific properties. Step (3) is optional, and only required to (a) verify incomplete models, or (b) reduce the

JPF state space. This is achieved by providing imperative event sequences representing the model environment, driving it into a state where the model checking should start. We will look at each of these steps in order.

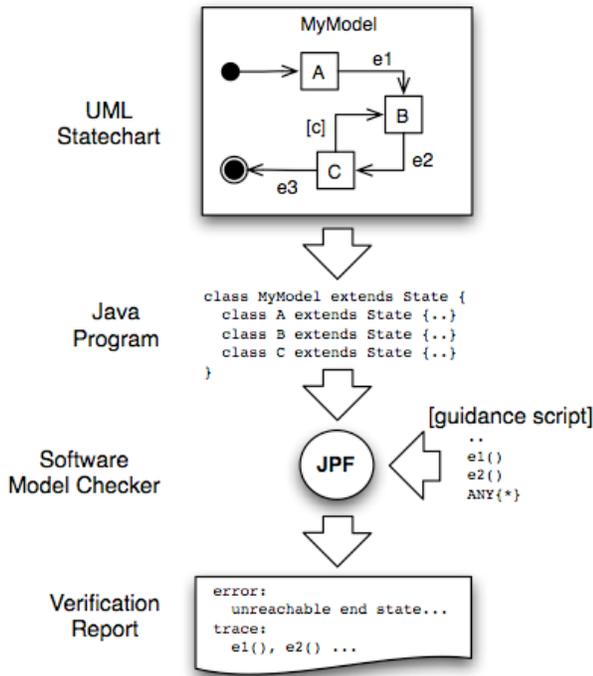


Figure 2 - toolchain

### 3. UML TO JAVA TRANSLATION

This step includes two aspects: (A) defining the format that diagrams are translated into, and (B) automating this process.

#### (A) Target Format

The format of the Java sources that are generated from UML diagrams is motivated by three major requirements:

- (1) Readability – generated sources should be human readable and should map diagram elements on a 1:1 basis.
- (2) No execution policy – the sources should reflect only structural information from the diagram (state composition, transitions), and leave execution policy (like order of trigger invocation) to a configurable runtime system.
- (3) Low model checker overhead – since the goal is to apply a model checker to the generated program, there should be no constructs causing state space explosion. Model and program state space should be closely aligned.

The format is based on a UML modeling library that is part of the *Java Pathfinder* (JPF [4]) distribution, effectively encapsulating the interface to the model checker.

Sources are generated according to the following simplified list of rules, which is also shown in Figure 3:

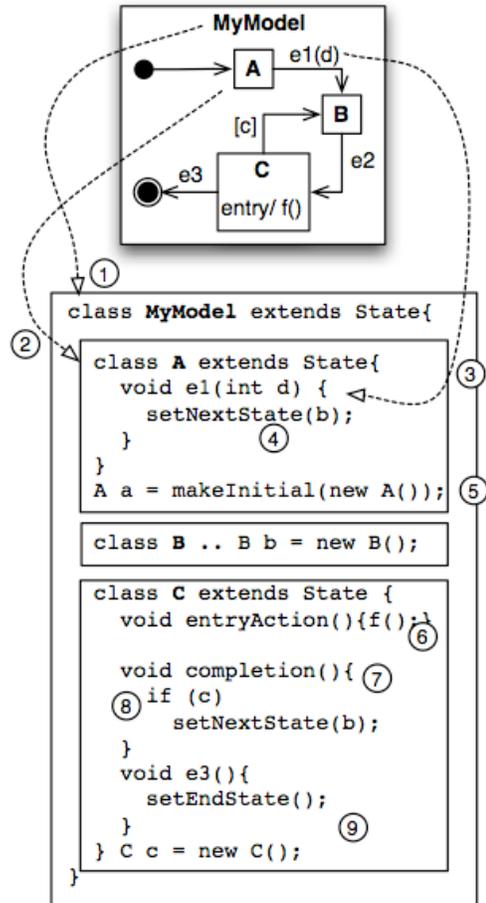


Figure 3 -- Diagram to Java Translation

- (1) each hierarchical diagram system is translated into one Java class (e.g., *MyModel*)
- (2) each composite or simple state of the diagram system is translated into a Java class that extends the library class *gov.nasa.jpf.statechart.State*. Sub-states of a composite state are translated into nested classes (e.g. *MyModel.A*)
- (3) each trigger is translated into a void method of the source state (e.g. *e1* → *A.e1(..)*), possibly taking parameters of restricted types (*int*, *double*, *String* etc.).
- (4) transitions are represented by calling *setNextState(targetState)* from inside of trigger methods. Trigger actions are implemented inside the trigger method bodies

(5) each *State* class has a corresponding field inside its encapsulating class, which is instantiated by using a default constructor. Instantiation of initial states is wrapped into a *State.makeInitial(newState)* call.

(6) *entry/* and *exit/* actions are translated into *entryAction()* and *exitAction()* methods of the corresponding state.

(7) completion triggers are implemented as *completion()* methods of the source state

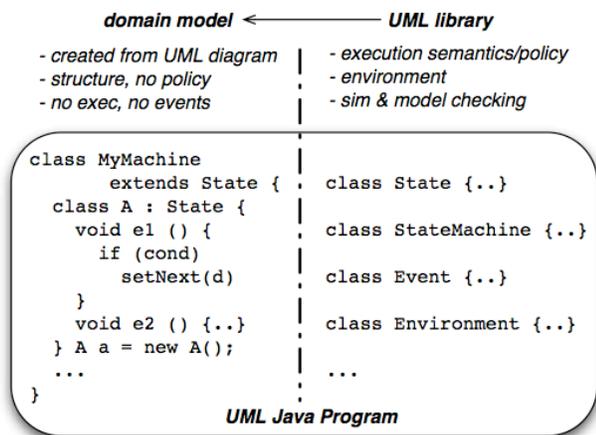
(8) trigger guards become boolean Java expressions inside of trigger methods

(9) end states are represented by calling *State.setEndState()* from inside of their corresponding trigger methods.

There is no explicit target construct for orthogonal regions, which are simply represented by having more than one sub-state field initialized with *State.makeInitial()*.

Target states are referenced in calls to *State.setNextState(target)* by using their corresponding field names (e.g. *class B → b*).

The resulting program consists of two layers: (a) the domain model generated from the diagram, and (b) the UML library that is part of the model checker distribution and interfaces the domain model with the software model checker.



**Figure 4 Program Structure**

The domain model closely corresponds to the UML diagram(s) and contains no execution policy. It is strictly focused on the invariant information found in the diagram.

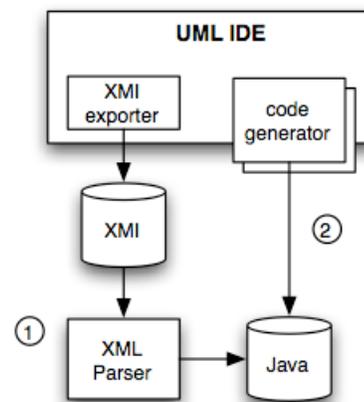
The UML library is highly model checker specific and hides all the required interfacing from the user. Its main purpose is to align the diagram and program state space as closely as possible, and provide the capability to adapt the verification to specific UML dialects and associated execution policies.

### (B) Automated Translation

Due to the separation of the Java program into these two layers, the model part can be kept almost free of implementation overhead. It remains concise and readable, and hence it is possible to create it manually. However, this is not the preferred mode of operation – ideally, the whole UML verification process becomes so automated that the user does not even have to be aware of the Java program.

This requires automatic generation of the domain model part, which preferably should be done in a generic way. Depending on the UML integrated development environment (IDE) in use, two solutions seem appropriate:

- (1) using the XML Metadata Interchange (XMI [4]) as an IDE independent intermediate diagram export format, together with an IDE external XML-to-Java translator
- (2) using configurable code generator features of the IDE to directly produce the domain model program



**Figure 5 -- code generation**

Endorsed as an OMG standard, XMI has been established as a (mostly) platform independent format to store and exchange UML documents. Since most UML IDEs support XMI export, and there exist various efficient libraries for XML parsing (e.g. Apache Xerces), creating an XMI-to-Java translator is the most portable solution to produce the domain model sources. We have used this approach together with the Unimod [5] UML development environment.

The challenge with XMI based translation is the platform specific storage of embedded code (guard expressions, actions). Usually, this is only stored as an XML attribute (i.e. a string literal) and requires an embedded expression parser.

Provided the UML IDE uses a well defined language for guards and actions, this limitation can sometimes be

overcome by directly utilizing code generation facilities of the IDE itself. Many IDEs support code generation based on an internal data model (e.g. the Eclipse EMF), and allow the user to configure and adapt the code generation process with less effort than writing a separate XMI parser. We are currently implementing this approach based on the iUML IDE [6,7].

#### 4. UML MODEL CHECKING WITH JPF

Before we look at what kind of properties we can verify in UML diagrams, we have to briefly discuss the underlying verification technology, which is *model checking* [8]. Since we translate UML diagrams into Java programs, we use the *Java Pathfinder* (JPF [9]) software model checker for model verification.

##### The JPF model checker

JPF is a highly configurable software model checker for Java bytecode programs, which was developed at the NASA Ames Research Center. It can be thought of as a drop in replacement for a normal Java virtual machine (VM). Unlike a normal VM, which only executes one path through the program, depending on input data and scheduling choices, JPF systematically explores all possible data and scheduling combinations. JPF can store program states and detect if states are equivalent, in which case it backtracks to a previously stored state and continues execution from there.

If JPF finds a *property violation* (defect), it not only reports the nature of the defect, but also the complete *program trace* – the sequence of operations leading to this defect.

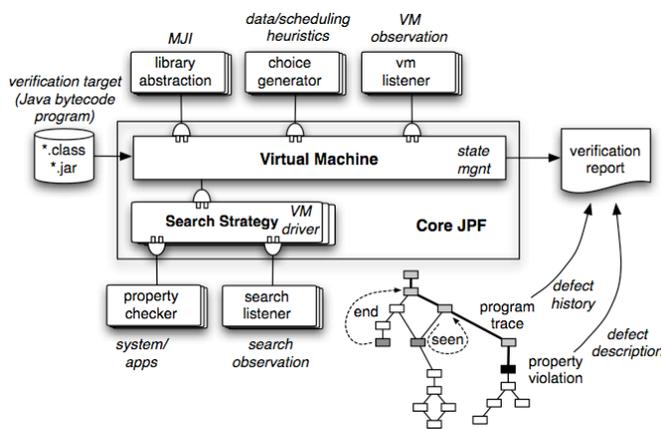


Figure 6 - The JPF Program Model Checker

A detailed description of JPF is neither possible nor required in the scope of this paper. The system is open sourced, and available from its public website [9], which also contains documentation.

##### Software Model Checking

Here, we only give a short exemplary introduction into software model checking, and refer to [10] for a better foundation. Consider the following program:

```
public class Test {
    public static void main (String[] args){
        Random random = new Random(42); // (1)

        int a = random.nextInt(2);      // (2)
        System.out.println("a=" + a);

        //...

        int b = random.nextInt(3);      // (3)
        System.out.println(" b=" + b);

        int c = a/(b+a -2);             // (4)
        System.out.println("    c=" + c);
    }
}
```

Depending on the random seed in line (1), executing this program with a normal Java runtime picks random values for variables *a* (2) and *b* (3), and then computes the variable *c* (4) based upon these random choices:

```
> java Rand
a=1
b=0
c=-1
```

If we depict possible variable value combinations in a tree, it becomes obvious that our simple program test only yields one possible execution path, missing variable values for *a* and *b* that would cause exceptions (e.g. *a*=0, *b*=2).

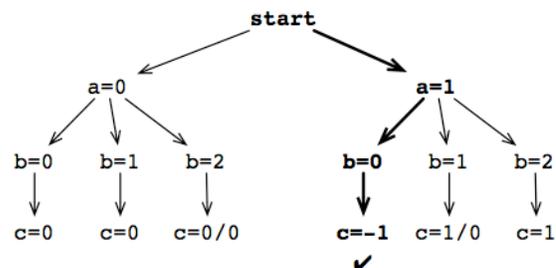


Figure 7 - testing

Executing the same program with a model checker like JPF explores all possible choices, not just one set, and hence finds the possible defects:

```
> bin/jpf +vm.enumerate_random=true Test

JavaPathfinder v4.1 - (C) 1999-2007
RIACS/NASA Ames Research Center

===== system under test
application: Test.java
```

```

===== search started: 5/23/07 11:49 PM
a=0
  b=0
    c=0
  b=1
    c=0
  b=2
===== error #1
NoUncaughtExceptionsProperty
ArithmeticException: division by zero
  at Rand.main(Test.java:15)
...

```

Looking at the above output, we see that if the model checker reaches the end of the program (displays the computed  $c$  value), it does not stop execution like a normal Java runtime, but automatically looks for other, unexplored choices. If it finds any, the model checker backtracks to the corresponding program state (reverting variable values and program counters), picks the next choice and continues from there. This process is repeated until no choices are left, or a property violation is detected:

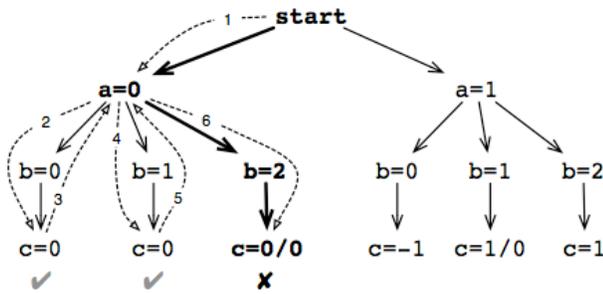


Figure 8 - model checking

Theoretically, model checking is a strict formal method that is guaranteed to find any defect that can occur due to our data and scheduling choices.

#### UML execution semantics

In the context of model checking UML statecharts, our primary choices are not thread context switches or random data input, but *enabling events*, i.e. the external stimuli for our model. Each time there is a choice between several possible events, we want to explore all of these choices recursively, to make sure that we cover all possible event sequences.

The program that is model checked by JPF is a generic class *gov.nasa.jpf.sc.StateMachine*, which is part of JPFs UML modeling framework. The generated model class is provided as an argument when running this program. Our execution semantics of UML statecharts are mostly defined by the implementation of the *StateMachine* class.

Each execution step starts by computing the set of enabling events, either by inspection of the classes of the currently active states, or by consulting a guidance script, which we will introduce in section 6 of this paper. The model checker then proceeds by processing each event of this set, looking at each active state to see if it defines a corresponding trigger method, and if it does, executes this method. In case the trigger causes a state transition, the new target state is stored, and subsequently added to the next set of enabling events. At the end of each execution step, the active state set is swapped with the next set, and the process is repeated until there either are no active states anymore, no more events to process, or a defect is found.

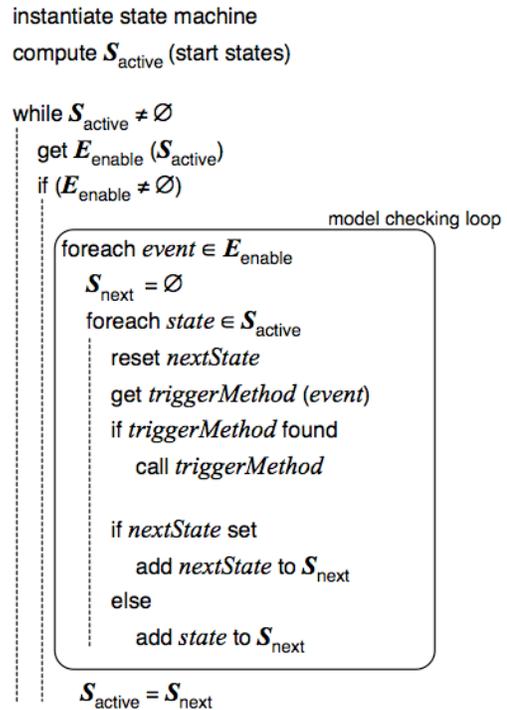


Figure 9 – UML statechart execution semantics

A detailed discussion of the framework implementation is not required in the context of this paper, since it is mostly concerned about aligning the UML model states with the Java program states, i.e. avoiding any overhead that makes it hard to map a given program state back into the model. Here, we are more interested in what we can verify about a UML statechart.

## 5. MODEL PROPERTIES

There are three different types of properties that can be checked by JPF in UML statechart programs:

- (1) built-in JPF properties
- (2) generic, domain specific properties implemented in the UML library
- (3) application specific properties implemented as separate JPF modules (listeners)

We will give examples for each of these property classes.

### (1) Built-in JPF Properties

This category is the most basic one that does not require any specific JPF knowledge. It includes a generic property

(P1) "no unhandled exceptions"

which holds if the program execution does not explicitly or implicitly cause any exception that is not handled within the program itself. As generic as this property is, it is very useful to specify domain and application specific properties as *assertions*, which are Boolean expressions evaluated at runtime, throwing *AssertionError* exceptions if the condition is violated (*AssertionErrors* are not supposed to be caught by applications).

Assertions can be used to specify application specific safety properties, i.e. events that are never allowed to occur. Assume the following model of the *Ascent* and *EarthOrbit* flightphases of a spacecraft

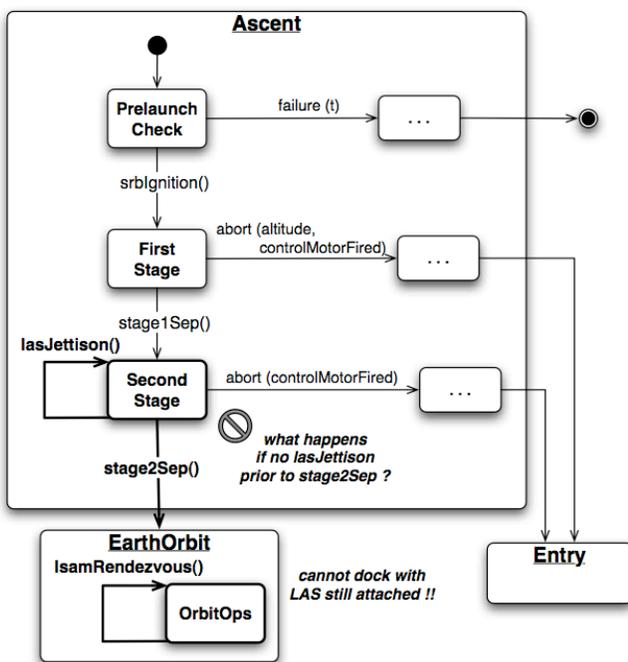


Figure 10 -- Explicit Safety Property Example

The *lasJettison* event represents a command that separates a launch abort rescue system from the top of the spacecraft stack, and is modeled as a self-transition (i.e. leading back into the same *SecondStage* state).

We further assume that the composite state *EarthOrbit* is specified in a different diagram, and contains an event *IsamRendezvous*, which represents a docking maneuver with another spacecraft. It is obvious this docking maneuver has to fail if the spacecraft did not execute a previous *lasJettison*, but our diagram model contains no provisions to enforce this. Moreover, the *Ascent* and *EarthOrbit* models might be done by different developers, not being aware of implicit assumptions of the other model part (the problem is not visible at the top level containing both composite states).

Model defects like this can easily be prevented by adding assertions to the code, in this case the *IsamRendezvous* trigger action:

```

class OrbitOps {...
    void IsamRendezvous(){...
        assert !spacecraft.contains(LAS) :
            "IsamRendezvous with LAS attached"
        ...
    } ...
}
    
```

This safety property does not require knowledge of preceding model transitions or JPF internals, and provides an efficient safeguard when executing the model with JPF:

```

...
===== error #1
NoUncaughtExceptionsProperty
AssertionError:
    IsamRendezvous with LAS attached
    at ...
===== choice trace #1
srbIgnition()
stage1Separation()
stage2Separation()
IsamRendezvous()
...
    
```

The (abbreviated) JPF output, which is configurable itself, shows not only the encountered defect, but also the sequence of events that caused the error.

### (2) Domain Specific Properties of the UML Library

The second category of properties targets UML specific defects. The corresponding checks are implemented in the UML library that comes with JPF, and do not require any specific code in the model. Examples are ambiguous transitions, illegal exits from orthogonal regions, and occurrence of events without corresponding triggers ("unhandled events"). Since JPF is most useful in the

context of state charts with non-trivial actions, we will use ambiguous transitions to demonstrate UML specific properties.

Consider a slightly more detailed version of the *Ascent* state from the previous example:

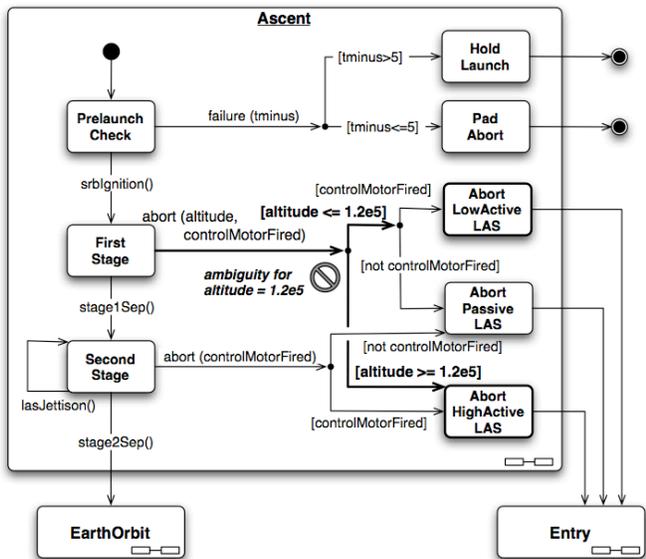


Figure 11 -- UML Specific Property Example

Although this diagram has just a modest level of complexity, it shows how quickly it can become difficult to manually find problems in large scale, real world models, even if the defect is visible within the same document.

The property we are interested in can be stated as

(P2) “no single trigger execution can lead to more than one transition”

which is synonym to the requirement that no trigger method execution of the model with a given set of arguments can do more than one *setNextState()* call.

A closer look at the *FirstStage* state shows that the *abort()* trigger has cascaded guards that branch on the trigger parameter values. The first level of guards checks the *altitude* parameter, and branches accordingly into *AbortLowActiveLAS* and *AbortHighActiveLAS*. A more careful inspection however unveils that the guard conditions overlap if *altitude=1.2e5*, due to using the wrong set of comparison operators (*<=*, *>=*).

Executing the corresponding model program with JPF automatically finds the defect:

```

===== error #1
NoUncaughtExceptionsProperty
AssertionError:
ambiguous transitions in: ascent.firstStage
processing event: abort(120000,true)

```

```

target-state 1: ascent.abortHighActiveLAS
target-state 2: ascent.abortLowActiveLAS

at ...

```

```

===== choice trace #1
srbIgnition()
abort(120000,true)

```

Even simple defects like this can easily be obfuscated by diagram details and layout, which shows the value of executable models.

### (3) Application Specific JPF Extension Properties

This category includes the most powerful property checks but also requires most effort and knowledge to implement them. Using JPF’s various extension mechanisms, it is possible to create highly sophisticated checks that do not involve model instrumentation, and go beyond standard UML syntax or semantics.

A typical example of checks that fall into this category are temporal properties. We already looked at a specific one in the introductory section – reachability. It can be stated as follows

(P3) “for every state in the diagram, there has to be a sequence of event/parameter combinations that finally cause a transition into this state”

While this might sound trivial from a diagramming perspective, we saw that the presence of actions and guards can deceive the visual perception of reachability.

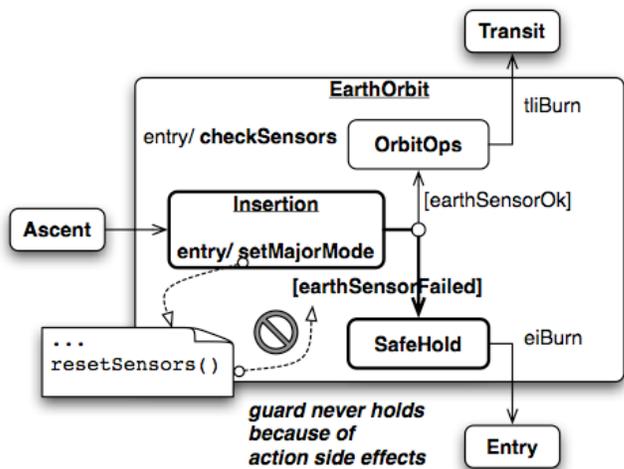


Figure 12 - Reachability Property

To briefly recap, the problem is that the guard for the *Insertion* ⇒ *SafeHold* transition never holds because of the *Insertion* entry/*setMajorMode()* action implementation, which resets sensor errors.

Since we already looked at the nature of this model defect, we now focus on the respective implementation of the property check. Even though the *Coverage* tool is a generic extension that can be used with all models, it is typical for the effort and knowledge that is required to create similar, application specific checks.

It is important to realize that such a property cannot be implemented within the application itself, since variables used to keep track of visit counts would be backtracked by JPF. Here, we want to accumulate information over all paths that are explored by the model checker.

In order to do this, we use a specific JPF extension mechanism called *VMLListeners*, which allows us to create and configure modules that can subscribe to various execution events within the JPF virtual machine. When we run this model through JPF, we specify some additional command line arguments that define and configure the listener to use:

```
>jpf gov.nasa.jpf.StateMachine
  +jpf.listener=.tools.sc.Coverage
  +sc.required=earthOrbit ...
```

The following code fragment is only intended to show the involved level of required JPF knowledge

```
...
public class Coverage {
  ...

  public void executeInstruction(JVM jvm) {
    Instruction insn =
      jvm.getLastInstruction();
    ...
    if (insn instanceof RETURN) {
      MethodInfo mi = insn.getMethodInfo();
      if (mi==visitedMth) {
        ThreadInfo ti
          =jvm.getLastThreadInfo();
        int stateRef = ti.getThis();
        MJIEnv env = ti.getEnv();
        int id = env.getIntField(stateRef,
          "id");

        int nVisits =
          env.getIntField(stateRef,
            "visited");

        int mRef =
          env.getReferenceField(stateRef,
            "machine");

        int mId =
          env.getIntField(mRef, "id");

        allCoverage[mId].addCoverage(id,
          nVisits);
      }
    }
  }
}
```

```
public void searchFinished(Search search){
  // print allCoverage information, check
  // if any required/forbidden state
  // constraints are violated
  ...
}
...
}
```

The *invokeInstruction()* notification gets called by JPF upon every completed bytecode, and allows very fine grained observation of the execution. The *searchFinished()* notification on the other hand is only invoked at the very end of the JPF run, and represents a high level callback.

Looking at the *instructionExecuted()* code fragment, it is obvious that the developer has to be familiar with JPF implementation details, i.e. there is a considerable learning curve. On the other hand, there is almost no limit of what can be verified with such listener extensions, giving us a broad range of potential properties, reflecting the underlying JPF design goals.

## 6. GUIDED MODEL CHECKING WITH SCRIPTS

A discussion of the UML model checking capabilities with JPF would be incomplete without mentioning how to guide the model checker into interesting parts of the model state space.

Recalling our statement that a model checker is supposed to find any defect that manifests itself in the model state space, our last example raises the question of how we ever got past the defect we showed in our second example: since the *Ascent* precedes the *EarthOrbit* phase, we should always run into the ambiguity defect before encountering the reachability problem.

The answer is *guided model checking*. Per default, the JPF UML framework only needs to know the name of our toplevel model class. The so called *scriptless* mode then proceeds by inspecting all state classes to identify trigger methods. In each UML execution step, JPF tries all events for which there are corresponding trigger methods in the set of active UML states, which constitutes an exhaustive search.

This mode is not suitable if our model is not complete yet, we want to ignore certain defects in preceding states we already know about, or we want to check if our model handles a given event sequence correctly. For these cases, the framework provides a guidance script mechanism that lets us control the sequence of events to process. In its most simple form, a guidance script only contains explicit event/parameter combinations:

```
// simple nominal event sequence

srbIgnition
stage1Separation
lasJettison
stage2Separation
lsamRendezvous
tliBurn
...
```

The next higher level is to tell JPF about event alternatives to explore, which is done with the *ANY {..}* expression:

```
srbIgnition()
ANY { abort(100000), abort(120000) }
```

This expands at runtime in two event sequences to execute:

- (1) srbIgnition, abort(100000)
- (2) srbIgnition, abort(120000)

If we specify a wildcard '\*' instead of a list of explicit event names, JPF will determine the set of events to choose from by inspection of the currently active states:

```
srbIgnition()
ANY { * }
```

By using the *REPEAT <n> {..}* expression, we can expand any sequence a given number of times, which means we can approximate scriptless mode by using a sequence like

```
REPEAT 1000 {
  ANY { * }
}
```

This is only an approximation because the repeat count still constitutes a search depth constraint, whereas scriptless mode does not.

Event parameter values can be expanded with a regular expression based syntax:

```
abort(1[024]00)
```

generates a set of three events:

```
{ abort(1000), abort(1200), abort(1400) }
```

which can also be written as

```
abort(1000|1200|1400)
```

Last not least, we can also specify event sequences that are only processed when a certain state becomes active. This uses the *SECTION <state name>* construct, and works hierarchically, i.e. if JPF does not find a SECTION for an

active state, it recursively tries to find one for its super states.

The current policy is to stop an ongoing event sequence as soon as a new state becomes active for which we have a section specified. This also means that it is easy to create loops, for example with self transitions, or transitions inside of a composite state that has a section (which will be re-entered for each of its child states that does not have a section of its own).

With this, we can finally present the script that was used in our last example:

```
SECTION ascent {
  srbIgnition
  stage1Separation
  lasJettison
  stage2Separation
}

SECTION earthOrbit {
  // covers Insertion and SafeHold
  ANY {*}
}

SECTION earthOrbit.orbitOps {
  lsamRendezvous
  tliBurn
}
```

This corresponds to the following informal description:

- (1) Proceed through the *Ascent* flight phase with a nominal event sequence (thus ignoring potential abort defects)
- (2) Once the *EarthOrbit* (composite) state is reached, explore all possible events, except of the *EarthOrbit.OrbitOps* state, for which we also just check the nominal event sequence

Guidance scripts are a convenient way to direct the model checker into interesting parts of the state space. However, to avoid introducing errors on the model environment side, and therefore unintentionally restrict the state space search, it is generally a good idea to keep scripts as simple as possible. When using sections, the user should also be aware of creating loops, especially if there are counters or other accumulated data structures in action code of the model, which would only be terminated by program state matching of the model checker.

It should also be noted that guidance scripts could be helpful to achieve scalability of very large models, by breaking verification down into separate phases.

## 7. CONCLUSIONS AND OUTLOOK

In this paper, we have shown an approach of how to verify UML state charts with embedded code in guards and actions. We first presented a translation scheme from UML to Java, and then applied the *JPF* software model checker to the generated Java program. Based on the structure of these programs, the underlying UML modeling framework, and the capabilities of the JPF model checker, we then gave examples of the property categories that can be verified with our approach. Finally, we showed how this approach can be applied to incomplete or large models by means of guidance scripts.

Due to size limits, we did not describe how the approach can be combined with compositional verification techniques to introduce environment assumptions, for example to constrain the sequence of possible events. We also did not discuss how to explicitly send events from within actions, which is a prerequisite for some executable UML dialects, but can obfuscate the separation between environment (script) and model (Java program).

While the JPF model checker has been developed and used since 1999, the UML verification is work in progress. We have applied the approach to UML statecharts with more than 1000 states, using execution semantics from different UML dialects.

Current work is mostly focused on better separation of model invariants (e.g. state structure), and tools specific execution policy. We also plan to implement more generic checkers, especially for temporal properties, and to extend guidance script semantics. Support for automatic diagram-to-Java translation will be added for selected UML tools. The primary goal of this project remains to provide a UML tool independent way to verify complex models, which is scalable with respect to both model size and level of refinement.

## ACKNOWLEDGEMENTS

We like to thank the members of the Java PathFinder development team at NASA Ames Research Center for their contribution to this project, namely Corina Pasareanu, Dimitra Giannakopoulou, and Masoud Mansouri-Samani. Visiting students Mihaela Gheorghiu Bobaru and Suzette Person provided various suggestions and crucial help for testing and application of the system. The work reported here was performed in the Robust Software Engineering group, headed by Dr. Joseph Coughlan (Technical Area Lead) and Dr. Michael Lowry (Principal Scientist), and funded by the NASA Exploration Systems Mission Directorate's Exploration Technology Development Program.

## REFERENCES

- [1] John D. Poole, Model Driven Architecture: Vision, Standards and Emerging Technologies, Workshop on Meta Modeling and Adaptive Object Models, ECOOP 2001
- [2] The Unified Modeling Language, OMG Web site <http://www.omg.org/technology/documents/formal/uml.htm>
- [3] D. Harel. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3): 231--274, June 2002
- [4] XML Metadata Interchange (XMI) Web site <http://www.omg.org/technology/documents/formal/xmi.htm>
- [5] Unimod Web site <http://unimod.sourceforge.net/>
- [6] iUML Web site <http://www.kc.com/products/iuml.php>
- [7] Chris Raistrick, Paul Francis, John Wright, Colin Carter, Ian Wilkie, Model Driven Architecture with Executable UML, Cambridge University Press 2007
- [8] Beatrice Berard et al, System Software Verification, Springer-Verlag, 2001
- [9] Java™ PathFinder Web site <http://javapathfinder.sourceforge.net/>
- [10] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda, Model Checking Programs, Automated Software Engineering Journal. Volume 10, Number 2, April 2003

## BIOGRAPHY



**Peter Mehlitz** is a senior computer scientist with Perot Systems Government Services (PSGS), formerly QSS, working for the Robust Software Engineering (RSE) group of the Intelligent Systems Division at the NASA Ames Research Center (ARC). As one of the main developers of the Java Pathfinder (JPF) software model checker, he is interested in software model checking, design patterns and design-for-verification (D4V). Mr. Mehlitz has more than 25 years of experience in large scale program development, using a broad spectrum of programming environments and operating systems. Mr. Mehlitz holds an M.S. in aerospace engineering, University of the Federal Armed Forces, Munich, Germany

