Program Model Checking Using Design-for-Verification: NASA Flight Software Case Study[1, 2,3]

Lawrence Z. Markosian, Masoud Mansouri-Samani, and
Peter C. Mehlitz
QSS Group, Inc.
NASA Ames Research Center
Moffett Field, CA 94035
{lzmarkosian, masoud, pcmehlitz}@email.arc.nasa.gov

Tom Pressburger
NASA Ames Research Center
Moffett Field, CA 94035
Tom.Pressburger@nasa.gov

Abstract—Model checking is a verification technique developed in the 1980s that has a history of industrial application in hardware verification and verification of communications protocol specifications. *Program model checking* is a technique for model checking software in which the program itself is the model to be checked. Program model checking has shown potential for detecting software defects that are extremely difficult to detect through traditional testing. The technique has been the subject of research and relatively small-scale applications but faces several barriers to wider deployment. This paper is a report on continuing work applying Java PathFinder (JPF), a program model checker developed at NASA Ames Research Center, to the Shuttle Abort Flight Management system, a situational awareness application originally developed for the Space Shuttle. The paper provides background on the model checking tools that were used and the target application, and then focuses on the application of a "Design for Verification" (D4V) principle and its effect on model checking. The case study helps validate the applicability of program model checking technology to real NASA flight software. A related conclusion is that application of D4V principles can increase the efficiency of model checking in detecting subtle software defects. The paper is oriented toward software engineering technology transfer personnel and software practitioners considering introducing program model checking technology into their organizations.

TABLE OF CONTENTS

1. INTRODUCTION

Model checking is a verification technique developed in the 1980s that has a history of industrial application in hardware verification. Program model checking is a technique for model checking software in which the program itself is the model to be checked. Program model checking has shown potential for detecting software defects that are extremely difficult to detect through traditional testing. The technique has been the subject of research and relatively small-scale applications but faces several barriers to wider deployment. In this paper we report our experience applying Java PathFinder (JPF)[JPF], a program model checker developed at NASA Ames Research Center, to the Shuttle Abort Flight Management system, a situational awareness application originally developed for the Space Shuttle.

Barriers to wider deployment of program model checking for real-world applications include the large, usually unbounded, state space of the application; the lack of tools (model checkers) that accept common programming languages; the complexity of these programming languages; software design process that do not support the requirements for applying verification tools; the absence of usable information about what properties should be checked; and the lack of guidance for software developers on how to take verification requirements into account during development.

This paper provides a brief overview of our program model checking toolset for C++ and the target application. It then discusses our experience in applying "design for verification" principles to increase the effectiveness of model checking. Other aspects of the toolset and the case

[1] 1-4244-0525-4/07/$20.00 ©2007 IEEE
[2] IEEEAC paper #1264, Version 1, Updated December 8, 2006
[3] The research reported in this paper was funded by the NASA Office of Safety and Mission Assurance's Software Assurance Research Program. The development of the Propel toolset, which was used in this research, was funded by the NASA Engineering for Complex Systems Program.

study have been reported elsewhere[O'Malley] or will be reported following completion of the study.

2. TOOLSET FOR MODEL CHECKING C++

The program model checker used in this work is directly applicable to Java. However, NASA flight software is written in C and other languages. While C has been most commonly used, C++ has seen increasing use, for example, in the Shuttle Cockpit Avionics Upgrade (CAU), and it is likely to be used in Constellation, NASA's program for crewed and robotic exploration of the solar system. To apply JPF to C++, we identified the semantic gaps between the two languages and developed a partial C++ to Java translation system, augmented by library and runtime support (for example, to handle pointers to primitive types and arrays of primitive types; and function pointers, which do not exist in Java). The complete set of tools for translating, analyzing, testing and model checking C++ is called Propel. Concurrently with development of Propel, we determined that the CAU coding standards avoided most of the problematic features of C++ (for example, multiple inheritance). *Thus we were able to show the feasibility of our approach for translating real NASA flight software.* Because of broad changes in funding for software engineering research at NASA, we were unable to mature the translator to the level required for automatic translation and execution of a Java implementation of SAFM. Nor were we able to address the issue of whether the translation quality would be adequate for model checking.[4]

Nevertheless, the translation infrastructure supported an adequate representation of parts of SAFM. We were able to (1) use JPF to model check parts of SAFM; (2) identify a design issue in SAFM and measure its effect on model checking; (3) apply a "design for verification" (D4V) technique and measure the improvement on model checking; and (5) through inspection, identify a defect that validated the application of D4V techniques. We are currently assembling a "Practitioner's Guidebook" based on our experience with SAFM and other applications.

3. TARGET APPLICATION

The Shuttle Abort Flight Management system (SAFM) was developed by NASA Johnson Space Center and General Dynamics Decision Systems as part of the Shuttle Cockpit Avionics Upgrade (CAU). SAFM evaluates the potential abort options for the Space Shuttle under various contingencies and provides abort recommendations to the crew. Primarily as a result of the loss of Columbia and the subsequent re-focusing of NASA's manned space flight away from the space shuttle and similar vehicles, both CAU and SAFM were canceled before being deployed, but the developers of SAFM remained interested in the analysis of the software. SAFM is a single-threaded application written in 30KLOC of C++ that follows coding standards appropriate to safety-critical applications.

SAFM was selected for this case study for several reasons.

- High criticality—it is capable of determining a safe landing trajectory for the shuttle autonomously in case of a failure requiring an abort during ascent.
- Commonality of interest with the developers and their willingness and availability to work with us.
- Good conformance of the application with technical criteria for C++ model checking using the Propel toolset.
- Manageable size.

High criticality—Advanced defect detection technologies such as model checking are thought to be capable of detecting subtle defects, defects which could escape detection using standard testing approaches, but they also represent a departure from current standard practice and are relatively immature. Further NASA development and deployment of leading-edge V&V technologies is best justified by demonstrating their success on safety-critical NASA applications. SAFM was rated "Crit 1R", which is high criticality but redundant software. The in-flight SAFM system is redundant because there are ground systems that duplicate and take priority over in-flight SAFM functionality in the presence of effective ground communication. In the absence of ground communication, the on-board SAFM system provides the required situational awareness to the crew. Since human life could depend on flawless operation of SAFM, it needs to be "human rated", which requires extensive V&V.

Commonality of interest—Success in applying a new software engineering technology in the context of a significant application requires a good working relationship among the application developers and the technology developers. The SAFM development team was willing to work with us for several reasons. They recognized a need for significantly greater autonomy (functionality achieved through automation in the absence of ground control) in space flight software than is currently available in applications like SAFM, and they recognized that a major barrier to greater autonomy has been the inability to perform verification and validation of autonomy software. Program model checking has been thought to be particularly well-suited to V&V of complex applications such as autonomy software. In addition, although SAFM had been delivered to NASA by its developers, it was not scheduled for deployment until 2006; this 2-year lead time would

[4] For example, the question of whether automated translation unacceptably increases the state space.

allow remediation of potential issues that our technologies might reveal.

Technical match—Mismatch between technology capabilities and a target application can result in redirection of project focus, resources lost in developing work-arounds, or even a complete failure. SAFM was a good fit to the technical characteristics required by the translation approach taken by Propel. Most of the coding standards identified for successful model checking using Propel were met in SAFM. For example, SAFM did not make extensive use of the C++ Standard Template Library, and had only one use of multiple inheritance. On the other hand, SAFM lacked an important characteristic which may have made it a much better candidate for an early feasibility demonstration: it is single-threaded, whereas the properties that model checkers are best adapted to verifying "out of the box" include freedom from deadlock and other concurrency defects. Another characteristic of SAFM that impeded the effectiveness of model checking is that its input space includes many floating point variables. Program model checking, which focuses on the correctness of state transitions, is not particularly effective at detecting defects in arithmetic computations.

Manageable size—Our program model checking tools are limited by the size of the state space of complex applications. SAFM's size was known from the start to be too large to allow program model checking in its entirety using the approach taken with the Propel toolset. This did not represent a critical problem because the application was modular and would allow scope selection. Our goal was not to model check a complete application but to demonstrate feasibility of Propel. Still, the size of the application needed to be limited because the entire application and its test environment would need to be ported, compiled and run on the platforms used by Propel developers at Ames Research Center; and the case study team would need to obtain a sufficient understanding not only of the application, but also of its low-level technical requirements (the system requirements specification—SRS) to apply Propel effectively.

4. DESIGN FOR VERIFICATION

One of the goals of the case study was to measure the effectiveness of "Design for Verification" (D4V) guidelines on model checking. The D4V approach is a set of principles, artifacts and processes for designing a program so that verification tools can be applied more effectively. It applies a variety of techniques such as design patterns, application programming interfaces, and source annotations, based on the hypothesis that the many of the general architecture and design principles leading to good modularity, extensibility and complexity/functionality ratio can be adapted to overcome some of the limitations of model checking and other software assurance methods.

Below are several D4V design principles relevant to model checking. In the next section we show the application of one of these D4V principles to SAFM.

Avoid redundancy—While not every form of redundancy is as bad from a verification perspective as it is from a maintenance point of view, behavioral redundancy to re-create (local) state can impose problems (the model checker does not distinguish between function local and object state). For example:

```
class A {
  void f (...) {
    X x(...);
    ... // do something with x w/o
changing it
  }

  void g (...) {
    X x(...);
    ... // do something else with
x w/o changing it
  }
};
```

We can change the above example by factoring out the declaration of x and turning it into object state.

```
class A {
  X x;
  ...
  void f(...) { /* use x */ }
  void g(...) { /* use x */ }
};
```

This change is not recommended in the case where x is modified and not used. This will cause an unnecessary increase in the number of object states.

Another harmful example of redundancy can occur with redundant aggregates, for example:

```
class A {
  X x;
  B b;
  ..
  void f (..) {
    x.modify();
    b.update(x);
  }
};
```

```
class B {
  X x;
  A a;
  ..
  void g (..) {
    x.modify();
    a.update(x);
  }
};
```

One can factorize these and turn them into delegation objects as shown below:

```
class A {
  X *x;
  void f (...) {
    x->modify();
    ...
  }
};

class B {
  X *x;
  void g (...) {
    x->modify();
    ...
  }
};

...
X *x = new X(...);
A *a = new A(...);
B *b = new B(...);
a-> setX(x); b-> setX(x);
```

These redundant types are usually a result of program extension: initially attempting to keep changes local (to a function member or class), but later realizing that the changes are required in a larger scope (whole class or set of classes).

Use polymorphism—This guideline is discussed in the next section.

Leverage model-based design—Model-based design provides useful hints of how a large system can be reduced so that its state space becomes searchable. If not inherently visible in the design (e.g. by means of using a 'State' design pattern), the model relevant information should at least be included as comments.

Use finite state machines—A finite state machine (FSM) is one of the most suitable models for formal checks, especially for concurrent systems. However, FSM's can have problems with inheritance (the state model can change in derived classes) if state aspects are not factorized (e.g., with the State design pattern).

Use platform-Independent Libraries and Abstraction Layers—Contemporary libraries can significantly exceed the size and complexity of applications. In practice, the application model or environment model will have to model some of these libraries (especially the ones without source code availability). Use of platform independent libraries, such as POSIX (especially pthreads), is recommended to easily model complexity and increase opportunity to reuse models.

Reduce concurrency—From a model checking perspective, the searched state space consists of all possible thread-state combinations, which implies that the level of concurrency has the biggest impact on state space size. As a consequence, reducing concurrency can be considered as the premier measure to ease model checking.

Reduce the number of threads—Threads can be a useful abstraction and implementation mechanism to partition independent program actions. However, when there is coordination (or interference) between these threads, the required synchronization mechanisms increase the time, increase the state space and introduce potential liveness and safety problems.

Some of these cases can be reduced with multiplexing patterns, e.g. separating asynchronous event emitters from a synchronous event processor by means of an event queue mechanism, i.e., combining sequenced operations inside of one dedicated thread (the event processor) instead of using explicit synchronization between several threads to enforce the sequencing.

Reduce the number of interleavings—Besides the raw number of threads, the state space is affected by the number of potential interleavings of these threads. While there exist automated techniques to reduce these interleavings (partial order reduction), most model checkers include some kind of interface to denote atomic sections (code which does not interfere with other threads). Previous versions of JPF supported two primitives: Verify.beginAtomic() and Verify.endAtomic(). They were used to mark a section of the code that would be executed atomically by the model-checker – i.e., no thread interleavings are allowed.

```
Verify.beginAtomic();

... // code without side effects
// outside this thread

Verify.endAtomic();
```

These calls are deprecated in the current version of JPF which by default supports partial order reduction instead.

5. Application of a D4V Design Principle

As indicated in the previous section, SAFM generally followed coding standards appropriate to high-criticality software. We identified several issues in SAFM that specifically related to D4V and model checking. We investigated these issues in detail in the sequencer component of SAFM. The first issue was the complexity of a critical method in the sequencer component involving a complex conditional statement; this presented us with an opportunity to apply one of the D4V guidelines. The second

was a subtle bug that was introduced when a newly added requirement was not implemented completely by the developers. We initially identified this defect not by applying the model checker but by inspection during the translation. We then used the ability of the model checker to provide better coverage than the existing testing framework that came with SAFM, in order to detect that bug.

In this paper we show the effects of the D4V modification on the use of the Java PathFinder (JPF) model checker and also on increased coverage provided by JPF in order to reveal the bug. We—

- selected relevant metrics,
- ran JPF on the original SAFM code containing the bug and obtained data for these metrics,
- made the D4V modification, and
- obtained the metric data for the modified code.

The metrics show a significant reduction in the resources needed to apply the JPF program model checker to detect the bug.

We confirmed with the SAFM development team that the apparent defect is indeed a real defect. *This defect was not revealed by the SAFM test environment and the developers' code review process*, and would have been difficult to find using traditional testing or by simply applying program model checking to the original code.

In addition to simplifying the code, the D4V modification made the previously-undetected bug quickly identifiable by reducing the length of paths that had to be checked to ensure detection. This confirms that the D4V approach increases the efficiency of program model checking.

In the subsections below we discuss details of the D4V guideline violation, resolution of the problem, and its impact on model checking.

Guideline violation—One of the D4V guidelines relates to the use of polymorphism. Programs, especially those converted from non-OOP languages like C, sometimes use state where they should use inheritance. Figure 1 shows an example of this in C++.

In this example the variable "type" is used to explicitly store the type of an instance of class A or B. It is initialized in the constructors of both classes and is used in the A::foo() method to determine the exact type of the object so that the appropriate operations are carried out. Besides being error prone (type initialization, branch-completeness) and breaking abstraction rules (base classes should not have to know about their concrete derived classes), this produces more code (the branches) and more data (type fields) that need to be handled by the model checker.

```
class A {
  int type;

  A (…) {
    type = TYPE_A;
    …
  }

  void foo(…) {
    …
    if (type == TYPE_A)
      … /* doAStuff */ ;
    else if (type == TYPE_B)
      … /* doBStuff */ ;
    …
  }
};
```

```
class B : public A {
  B (…) {
    type = TYPE_B;
    …
  }
  …
};
```

Figure 1 Use of an explicit state variable to determine object type

One could use the inheritance mechanism in C++ to do this as shown in Figure 2.

```
class A {
  virtual void foo(…) {
    … /* doAStuff */
  }
};
```

```
class B : public A {
  virtual void foo(…) {
    … /* doBStuff */
  }
```

Figure 2 Use of built-in inheritance to determine object type

We identified a similar problem in our case study. In the sequencer component of the original SAFM C++ code, instead of using the existing type hierarchy (shown in Figure 3), the developers used the names of the scenarios in a complex conditional-statement in order to decide the type of the scenario to run next.

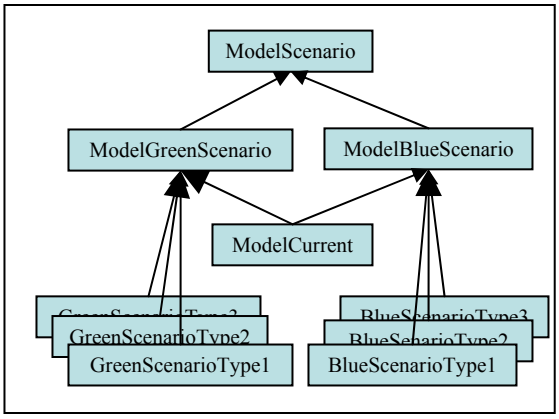


Figure 3 Scenario Type Hierarchy

Figure 4 shows a segment of the Java model which represents the problem as we found it:

```
public class ModelSequencer {
  public void processScenarios() {
    abcFlag=modelInput.getAbcFlag();              // Input #1
    if(modelInput.getModeIndex()==MODE_3){        // Input #2
      abcFlag=true;
    }
    for (short i=0; i<ModelConstants.numberOfScenarios; i++) {
      if (scenarios[i].isApplicable()) {          // Input #3
        if ((ModelConstants.currentScenarioName.equals(scenarios[i].getName()) &&
             abcFlag==false)||
            ModelConstants.blueScenario1Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario2Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario3Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario4Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario5Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario6Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario7Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario8Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario9Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario10Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario11Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario12Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario13Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario14Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario15Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario16Name.equals(scenarios[i].getName())||
            ModelConstants.blueScenario17Name.equals(scenarios[i].getName()))
            {
              ModelBlueScenario bScenarioToProcess; //set inputs to the Blue Scena
            } else {
              ModelGreenScenario gScenarioToProcess; //set inputs to the Green Sce
            }
            scenarios[i].processThis(modelInput);
```

Figure 4: Segment of model showing the D4V issue

The problems with this code are that:

- it is excessively complex (has too many conditions) for the functionality that it performs, and
- its complexity masks a bug.

The large number of name string comparisons done at runtime could easily be avoided with the proposed D4V modification. For example when the for-loop picks scenario `blueScenario17` as the next scenario to initialize and process, in addition to the last name comparison, 17 other name comparisons for other scenarios are performed, before the scenario type is decided. This corresponds to 17 extra nodes in the control flow graph used to determine cyclomatic complexity. The complexity decreases performance (in both space and time).

Comparisons based on string constants can be erroneous (e.g., if the strings are entered incorrectly). Testing this code thoroughly requires—

- checking that all the scenarios have the right names – names that match their types - (21 scenarios – 21 name checks) and
- checking that the names of the scenarios in the conditional statement are correctly entered and match the actual scenario names.

In addition to masking the bug, which we describe later, the complexity of the original implementation also makes it more difficult to maintain and extend the implementation. For example, if a new scenario type is added, all the conditional statements that decide the scenario type based on its name in the original code must be modified to include comparison for the new scenario's name.

Resolution of the problem—The built-in type inheritance and method overriding mechanisms in the C++ language can be used to determine the types of the scenarios instead of using an explicit state variable (the scenario's name string) to make that decision.

To implement this change we declared a new `isBlue()` virtual method in the `ModelScenario` class which is defined in the `ModelGreenScenario`, and `ModelBlueScenario` classes returning false and true respectively. The method is also (re)defined (i.e., overridden) in the `ModelCurrent` scenario returning true if `abcFlag` is false. We then replaced the complex condition in the inner if-statement with a single call to the `isBlue()` method as shown in Figure 5:

```
public class ModelSequencer {
  public void processScenarios() {
    abcFlag=modelInput.getAbcFlag();              // Input #1
    if(modelInput.getModeIndex()==MODE_3){        // Input #2
      abcFlag=true;
    }
    for (short i=0; i<ModelConstants.numberOfScenarios; i++) {
      if (scenarios[i].isApplicable()) {          // Input #3
        if (scenarios[i].isBlue(abcFlag)) {
          ModelBlueScenario bScenarioToProcess;
          //set inputs to the Blue Scenario ...
        } else {
          ModelGreenScenario gScenarioToProcess;
          //set inputs to the Green Scenario ...
        }
        scenarios[i].processThis(modelInput);
      } else {
        scenarios[i].resetParms();
      }
    }
    return;
  }
```

Figure 5 D4V modification simplifying the code

This design change reduced the cyclomatic complexity of the source code, which, in turn, reduced the number of tests required to achieve branch coverage. The cyclomatic complexity of the `processScenarios()` method compared with the original C++ code dropped by 18 – from 32 to 14.

Unlike the original code, the revised code does not perform name comparisons. The virtual method dispatching mechanism chooses the right scenario based on the object type hierarchy and invokes its `isBlue()` method. It does not need to invoke any other scenario's `isBlue()` method. It is invoked only once in the conditional part of the inner if-statement.

Testing the revised source code does not require testing the method dispatching mechanism itself (the compiler and runtime environments are separately certified, as appropriate). Only three definitions of the `isBlue()` method need to be tested. It is not necessary to test for each instance, as is the case for each scenario name in the original version.

In fact, the model could be further simplified by completely removing the name of the scenarios altogether; this reduces the size of the state vector which leads to better savings in time and memory. We did not do that here because we wanted to show the effect of the specific D4V change only.

The change also improves extensibility and maintainability. In the future, scenarios can be added and referenced in this code without having to verify that their names are entered correctly and that they match the corresponding types—resulting in fewer specifications to test. This is an especially significant improvement for our example, because it eliminates the need to test implicit assumptions about the scenario name to type mappings. Such assumptions should be made explicit in the artifacts, for example as comments in the code and as assertions which can be used by a verification tool like a model checker. They also need to be linked upward to the relevant items in the requirements document. Existence of such unstated but critical assumptions accurately reflects our experience with the actual NASA application. All that is needed in the new version of our example is to extend the right class, `ModelBlueScenario` or `ModelGreenScenario`, and then the existing type hierarchy and virtual methods take care of the rest.

Impact on model checking—As shown in Figure 5 the first if-statement sets the `abcFlag` value to true if the modeIndex is set to `MODE_3`. In the third if-statement (the one modified as part of the D4V change) the same variable is used to decide if a current scenario is a blue scenario or a green one in order to set its parameters appropriately before processing it (calling the `processThis()` method). Therefore both values `abcFlag` and `modeIndex` are used to make this decision.

However, we found that the current scenario's `processThis()` method as shown in Figure 6, in order to decide whether to run a blue or a green scenario, checks only the `abcFlag` value and fails to check `modeIndex`. This results in an inconsistent state where the scenario parameters can be set up as one type of scenario but may be run as another type.

```
public class ModelCurrent extends ModelBlueScenario {

  public void processThis(ModelInput modelInput){
    if (!modelInput.getAbcFlag()) {    // Input #1
      // Process as a blue scenario
      super.processThis(modelInput);
    } else {
      // Process as a green scenario
      greenScenario.runGreenScenario();
    }
    return;
  }
```

Figure 6 An erroneous condition in processing a current scenario

We found this bug not through application of the model checker but by visual inspection while trying to create the model in order to show the effects of the proposed D4V

change. To confirm the bug using the model checker, we inserted the following assertion to check the value of modeIndex in the first branch of the if-statement:

```
Assert(modelInput.getModeIndex()
    !=ModelSequencer.MODE_3)
```

We also used the model checker's choice generators to generate test data sets to allow the model checking to cover the state space with respect to the 3 input values used by our model.

Software model checking requires making "good" choices to reach the suspect system states within the resource constraints of the tool and execution environment. The mechanism used by JPF to systematically explore the state space by generating random and non-deterministic choices is called `ChoiceGenerators`.

JPF provides a number of methods in its `gov.nasa.jpf.jvm.Verify` class that are used for generating data choices. Within our model we have used two of these methods: `getBoolean()` and `getInt()`:

```
abcFlag=
 gov.nasa.jpf.jvm.Verify.getBoolean();
seqModeIndex=
 gov.nasa.jpf.jvm.Verify.getInt(-2, 6);
applicable=
 gov.nasa.jpf.jvm.Verify.getBoolean();
```

The `abcFlag` and `applicable` values are set by invoking the `getBoolean()` method which returns random choices of true or false. The `seqModeIndex` value is set using the `getInt()` method which randomly selects integer values between -2 and 6, the range of the values for this variable in the original code. The model checker tries all these values by backtracking to the same invocations multiple times until all the values that could be generated by these methods are actually returned and used. We ran the model checker with this setup.

The metrics collected while running JPF in order to find the bug, for before and after the proposed D4V change, are shown in Table 1 together with the percentage improvements:

Metrics	Before D4V	After D4V	Improvement
Number of choices	36	36	
Total Bytecode Instructions Executed	202993	72168	64%
Relative time [ms]	1658	1242	25%
Search depth	3	3	
New states	47	47	
Revisited states	0	0	
End states	29	29	
Backtracks	43	43	
Processed states	15	15	
Restored states	0	0	
Total memory [kB]	5368	4652	13%
Free memory [kB]	0	0	
State Vector Length	1360	1360	

Table 1 Metrics for before and after D4V modifications

While most of the metrics are the same before and after, this simple modification has a significant impact on the time and memory usage of the model checker required to *find* the bug. Note that the savings may be exponential in practice. In this simple sequential example we are dealing with only three input values – two Booleans and an enumerated type with 9 possible values making a total number of 36 different test cases. Only 2 values (a Boolean and the enumeration) are directly involved in driving the model to the erroneous state. In more realistic cases where we are dealing with hundreds of input values or multithreaded applications with arbitrary thread interleavings, D4V modifications can have a significant impact on model checking performance.

6. FUTURE WORK

The case study, in which the D4V results reported here were obtained, has concluded. The experience with this case study and with others is the basis for a practitioner's guidebook to program model checking to be delivered to the NASA Independent Verification and Validation Facility at the end of March, 2007. R&D on program model checking for C++ continues at NASA Ames Research Center's Robust Software Engineering group, using a strategy that addresses the major issues associated with source-to-source translation.

ACKNOWLEDGEMENTS

The research and development reported here was conducted in the NASA Ames Research Center's Robust Software Engineering group, headed by Dr. Michael Lowry. The authors acknowledge the contributions of Howard C. Hu of NASA Johnson Space Center, who developed the Shuttle Abort Flight Management system used as the target application for the research reported in this paper; Tanya Lippincott, of General Dynamics Decision Support, who collaborated on the specification and verification of critical properties of SAFM and on analysis of test results; John Penix, who was the initial Prinicipal Investigator on the research project reported here as well as on the ECS-funded development of the Propel toolset; Owen O'Malley, who was the lead developer of the Propel toolset; Willem Visser, the developer of Java PathFinder and a contributor to the strategies employed in the research reported here.

REFERENCES

[JPF] http://javapathfinder.sourceforge.net

[SPIN] Holzmann, G. The SPIN Model Checker. Addison Wesley, 2004

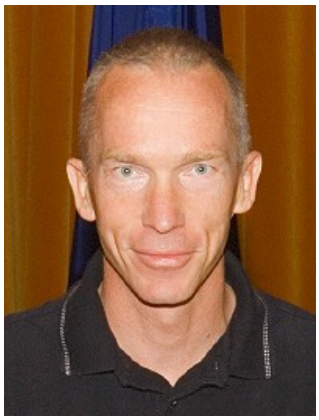
[O'Malley, O. Mansouri-Samani, M., Mehlitz, P. and Penix, J. "Seeing the Invisible: Embedding Tests in Code that Cannot be Modifed", Infotech@Aerospace, 2005.

BIOGRAPHIES

Lawrence Z. Markosian *is a Computer Scientist with QSS Group, Inc. at NASA Ames Research Center, where he led a team developing verification tools for C++. He has been a member of the NASA Software Engineering Initiative's Research Infusion team. Prior to joining NASA, he was a founder of Reasoning Systems., where as VP of Applications Development he managed technology transfer of advanced software engineering tools. Markosian has an undergraduate degree in mathematics from Brown University and has done graduate work at Stanford University in logic and artificial intelligence.*

Masoud Mansouri-Samani *is a senior research scientist with QSS Group, Inc. at NASA Ames Research Center. He has helped develop advanced V&V tools including Propel, a toolset for model checking C++. Mansouri-Samani obtained his B.Eng. in computer science (1991) and Ph.D. in Monitoring of Distributed Systems (1995) at Imperial College. He then joined HP and worked on remote systems diagnosis and management and was involved in the development of a framework for building diagnostic solutions in Java prior to joining QSS Group, Inc.*

Tom Pressburger *is in the Robust Software Engineering group led by Dr. Michael Lowry at NASA Ames. He led a software engineering research infusion subgroup, and has been leading the group developing a model checking practitioner's guidebook. He has worked in the area of program synthesis, participating in the development of systems for Java model checking, state estimation, statistical analysis, solar system geometry, and, when he was at the Kestrel Institute, algorithm design. He also worked at Reasoning, Inc. in software reengineering. He holds a BS in Mathematics from CalTech and an MS in Computer Science from Stanford.*



Peter Mehlitz *is a senior computer scientist with QSS Group, Inc., in the Robust Software Engineering group at the NASA Ames Research Center (ARC). As one of the principal developers of the Java Pathfinder software model checker, he is interested in software model checking, design patterns and design-for-verification. Mr. Mehlitz has more than 25 years of experience in large scale program development, using a broad spectrum of programming environments and operating systems.*