Formal Analysis of the Remote Agent Before and After Flight

Klaus Havelund[1], Mike Lowry, SeungJoon Park[2],
Charles Pecheur[2], John Penix, Willem Visser[2], Jon L. White[3]

The Automated Software Engineering Group
NASA Ames Research Center,
Moffett Field, California, USA.

[1] Recom Technologies, [2] RIACS, [3] Caelum

Abstract

This paper describes two separate efforts that used the SPIN model checker to verify deep space autonomy flight software. The first effort occurred at the beginning of a spiral development process and found five concurrency errors early in the design cycle that the developers acknowledge would not have been found through testing. This effort required a substantial manual modeling effort involving both abstraction and translation from the prototype LISP code to the PROMELA language used by SPIN. This experience and others led to research to address the gap between formal method tools and the development cycle used by software developers. The Java PathFinder tool which directly translates from Java to PROMELA was developed as part of this research, as well as automatic abstraction tools. In 1999 the flight software flew on a space mission, and a deadlock occurred in a sibling subsystem to the one which was the focus of the first verification effort. A second quick-response "cleanroom" verification effort found the concurrency error in a short amount of time. The error was isomorphic to one of the concurrency errors found during the first verification effort. The paper demonstrates that formal methods tools can find concurrency errors that indeed lead to loss of spacecraft functions, even for the complex software required for autonomy. Second, it describes progress in automatic translation and abstraction that eventually will enable formal methods tools to be inserted directly into the aerospace software development cycle.

1 Introduction

Complex concurrent software is difficult to debug and even more difficult to test with adequate coverage. With the increasing power of flight-qualified microprocessors, NASA space enterprises are experimenting with a new generation of non-deterministic flight software that provides enhanced mission capabilities. A prime example is the Remote Agent (RA) autonomous spacecraft controller developed at NASA. In May 1999, the RA was successfully demonstrated in flight on Deep Space 1 (DS-1), the first flight of NASA's experimental New Millennium program. The RA is a complex, concurrent software system employing several automated reasoning engines using artificial intelligence technology. The verification of such complex software is critical to its acceptance by science mission managers.

This paper describes formal methods verification efforts for one of the three subsystems of the RA – specifically, the RA Executive, which provides operating-system level capabilities for goal-directed software. Two different verification activities were conducted, before and after flight, using different technologies and in very different contexts. As such, this paper provides two successive snapshots of progress towards making formal methods verification cost-effective.

In 1997, while the RA was still in the development stage, we modeled and verified a subset of the core services of the RA Executive using the SPIN [10] model checker. That verification unveiled several concurrency bugs that were acknowledged by RA Executive developers [7].

As a result of this effort, it was decided to develop model checking technology for a main stream programming language in order to reduce the amount of time spent on modeling the behavior of programs in SPIN. The result was a translator, called Java PathFinder, from Java to the modeling language PROMELA of SPIN. In addition, a tool was developed for abstracting Java programs to reduce their state space, making model checking tractable.

Then, during the actual RA experiment in 1999, a deadlock occurred within less than 24 hours of operation. Although the problem was promptly identified and circumvented by the DS-1 team, we took the challenge of trying to diagnose the error in a fast-response "clean room" experiment [1]. After isolating a suspicious part of the program by visual inspection, we modeled it in Java, and then used Java PathFinder to exhibit a concurrency error that indeed turned out to be the one that had occurred in flight.

One key observation of these two successive experiments is that the error that caused the deadlock is exactly isomorphic to one of those found using SPIN two years before in another part of the code. It is a concurrency error, whose activation depends on a priori unlikely scheduling conditions between concurrent tasks. In fact, this error did not appear in over 300 hours of system-level testing on JPL's flight system testbed. The conditions under which it occurred in flight were not anticipated during testing. A principal benefit of model checking technologies is to be able to exhaustively cover scheduling alternatives. This paper gives a compelling illustration of how model checking found an error that was a priori unlikely but did actually occur. It also discusses gaps between previous formal method tools and requirements for making them easily accessible to system developers for 'in the loop' verification. Technological advances towards narrowing this gap are described in the context of the RA verification.

Section 2 describes the RA experiment. Section 3 describes the verification effort before flight, while Section 4 describes the verification effort after flight. The section also presents Java PathFinder. Section 5 describes the Java abstraction tool, and finally, Section 6 contains a conclusion.

[1] By "clean room" we are referring to the fact that, while the verification was post-facto, the team had no interaction with the actual debugging team.

2 The Remote Agent Experiment

To prepare for space exploration programs of the next decades within a reduced budget, NASA has set up the New Millennium program: a series of technology validation flights whose objective is to accelerate the qualification for flight of new spacecraft technology. One of the objectives of the New Millennium program is to increase spacecraft autonomy, moving from the low-level control sequences currently in use towards mission-level planning and autonomous health monitoring and recovery.

Deep Space 1 (DS-1), the first New Millennium Mission, was launched from Cape Canaveral on October 24, 1998 and ended its primary mission in September 1999 (it is still operating and is on its way for a comet encounter in 2001). During that mission, it successfully tested 12 cutting-edge technologies such as ion propulsion, on-board optical navigation, and the AI-based Remote Agent, marking the first operational use of artificial intelligence during space flight.

In its initial design, the RA Experiment (RAX) on DS-1 consisted of a short, limited 12-hour scenario designed to gain confidence in the RA, followed by a complete 6-day scenario that was the full RA test. Later, the experiment had to be compressed into a single 2-day scenario, to accommodate external mission constraints. The original scenarios were designed to cover a formal list of validation objectives. To protect the main DS-1 mission from possible misbehaviors of RA, the design included a "safety net" that allowed the RA experiment to be completely disabled with a single command, issued either from the ground or by on-board fault protection.

The RA went through a thorough qualification process before being allowed to run on DS-1. Though some formal verification tasks, such as the one reported here, were performed as feasibility studies, the formal qualification process relied on more conventional testing approaches. However, since the RA was a flight experiment, and not flight software, it was not subjected to the testing standards of the latter.

This section is a short summary of the flight qualification and experience of the RA [2, 13].

2.1 Remote Agent

The RA is an autonomous spacecraft controller developed by NASA Ames conjointly with the Jet Propulsion Laboratory (JPL) [12]. It comprises three components:

- The Planner and Scheduler (PS) [11] generates flexible plans, specifying the basic activities that must take place. Given a mission goal, it produces sequences of tasks for achieving this goal using available system resources.

- The Smart Executive (EXEC) [14] receives the plan from the Planner/Scheduler and then commands spacecraft systems to take the necessary actions to achieve and maintain the specified spacecraft states.

- The Mode Identification and Recovery component (MIR), called Livingstone [16], monitors the state of the spacecraft, detects and diagnoses failures and suggests recovery actions to the Executive.

The Executive subsystem is the focal point of the verification work discussed in this article. It combines features of multi-threaded operating systems with aspects of AI languages based on sub-goaling, such as Prolog. It is conceptually composed of three layers: a set of *core services* that implement a robust operating system for executing concurrent tasks, a set of *engine modules* including a plan runner, and a set of mission-specific *task programs*. The Executive schedules the execution of concurrent tasks. It also monitors a set of *properties* associated with system resources, and takes recovery actions on property violations. The Executive is written in a multi-threaded LISP, using a set of LISP macros called the *Executive Sequencing Language* (ESL) developed at JPL.

2.2 Testing the Remote Agent

Because autonomous systems such as the RA need to respond robustly in a wide range of situations, verifying that they respond correctly in all situations would require a huge number of test cases. Furthermore, these tests ideally have to be run on high-fidelity testbeds that are highly oversubscribed, hard to configure, and, running at real time speeds, take hours or days for a single run.

To address these problems, the RAX team followed a "baseline testing" approach, starting from nominal scenarios and testing a number of nominal and off-nominal variations around these scenarios. A wide range of variations were run on more available and faster low-fidelity testbeds, leading to the identification and resolution of 100-200 bugs during 18 months. An automated testing tool was designed for this purpose. Some of the most likely off-nominal variants were run on medium-fidelity testbeds, while only nominal scenarios and certain performance and timing related tests were performed on high-fidelity testbeds. The final stage was a pair of "dress rehearsal" *operational readiness tests* (ORTs), involving actual communication with the mission control center. The bulk of the problems identified during testing were found with the low-fidelity testbeds. The ORTs only identified minor shortcomings that were resolved prior to flight.

2.3 Remote Agent in Flight

On Monday, May 17th, 1999, 11:04 am PDT, a telemetry packet confirmed that the RA had taken control of DS-1. The scenario went on smoothly, achieving 70% of the objectives, until Tuesday 7:00 am, when it became apparent that a command had not been executed as expected by the RA. The RA Executive was blocked, although the rest of the RA and the spacecraft were otherwise healthy. The Executive's low-level commands were used to gather a maximum of information, and then the experiment was interrupted.

By late Tuesday afternoon, the RAX team had found the source of the problem in the Executive code. They designed a 6-hour scenario that was run on Friday morning and went successfully through the remaining 30% of the objectives. A patch was also generated, but the DS-1 mission decided not to uplink it, considering the insufficient testing of the patch and the very low probability of the problem recurring.

The blocking was due to a missing critical section that had lead to a race condition between two concurrent threads. Under some very precise and unlikely timing circumstances, both threads could end up in a deadlock condition in which each one was waiting for an event that only the other one could provide, which is exactly what happened in flight.

3 Formal Analysis Before Flight

In April-May 1997 we analyzed part of the RA Executive using the SPIN model checker [7]. This effort lead to the discovery of five errors in the LISP code which are described below. As discussed in Section 4.3, one of these errors is isomorphic to the error that actually occurred during flight, causing a deadlock. First we give a short description of SPIN and its modeling language PROMELA.

Then we explain how a PROMELA model was extracted from the LISP code, and how properties were stated and verified in the model, leading to the discovery of the five errors. We conclude with a discussion of the methodology that has been followed.

3.1 The SPIN Model Checker

SPIN [10] is a tool for analyzing the correctness of finite state concurrent systems with respect to formally stated properties. A concurrent system is modeled in the PROMELA modeling language, and properties to be verified are formalized as assertions in the program or as formulae in the temporal logic LTL (*Linear Temporal Logic*). SPIN provides a model checker, which automatically examines all program behaviors in order to decide whether the PROMELA program satisfies the stated properties. In case a property is not satisfied, an error trace is generated, which illustrates the sequence of executed statements from the initial state to the state that violates the property. These error traces can then be executed in a simulator. The set of states reachable from the initial state must be finite in case a property needs to be proven correct for the whole state space.

A PROMELA program consists of a set of sequential processes that communicate via message passing through bounded buffered channels and via shared variables. Processes can be created dynamically. The behavior of an individual process is described using the statement language which provides many standard constructs such as variable assignments, channel communications, loops, conditionals, and sequential composition. Variables are typed, where a type can either be primitive, such as integer, or composite in the form of arrays and records. PROMELA provides inline procedures, which is a limited notion of procedural abstraction that is implemented via macro expansion.

Each process represents a finite automaton, and the global behavior of the system is then obtained by computing on-the-fly an *asynchronous* interleaving product of all these automata, creating the global state space. To perform model checking, SPIN translates (the negation of) any LTL formula into a Büchi automaton, and computes the *synchronous* product of this and the global state space. The result is again a Büchi automaton. If the language of this automaton is empty it means that the formula is satisfied. SPIN searches the state space depth-first, creating the states on-the-fly. A partial-order reduction technique is used to prune the set of transitions to be explored.

3.2 Creating a PROMELA Model

The modeling activity focused on the core services of the plan execution module. The RA Executive core is designed to support execution of software-controlled *tasks* on board the spacecraft. A task often requires specific *properties* to hold during its execution. When a task is started, it first tries to *achieve* the properties on which it depends, after which it starts performing its main function. Several tasks may try to achieve *conflicting* properties; for example, one task may try to turn on a camera while another task tries to turn it off. To prevent such conflicts, a task has to *lock* in a lock table any property it wants to achieve. Once, a property is locked, it can be achieved by the task locking the property.

Properties may, however, be unexpectedly *broken* while tasks depending on them are executing. A property is defined as broken when it is locked in the lock table by some task, has been achieved (an extra boolean field in the lock table), but for some reason fails to hold on board the spacecraft. For the purpose of detecting which properties hold on board, a database is maintained of all properties being true at any time. Hence, an inconsistency can be detected by relating the lock table with the database. Tasks depending on a broken property must be interrupted and informed about the anomaly. For this purpose, a daemon monitors the changes on board the spacecraft, and in particular the consistency between the lock table and the database. The daemon is normally asleep, but is awakened whenever there is a change in the lock table or the database, where upon it checks their consistency.

The PROMELA model focuses on operations on the lock table. Hence, it is an abstraction of the LISP program, omitting details irrelevant for the lock table operations. The LISP program is approximately 3000 lines of code while the PROMELA model is 500 lines of code. Furthermore, the model only deals with a limited number of tasks and properties in order to limit the search space the SPIN model checker has to explore. Most abstractions were made in an informal manner without any formal proofs showing that bugs are maintained. Hence, in the abstraction phase we may have left out errors in the LISP code. However, all the errors we found in the model were also errors in the LISP code.

To give an idea of the modeling, we show how the daemon was translated, since it was the daemon that con-

```
(defun daemon ()
  (loop
    (if (check-locks)
        (do-automatic-recovery))
    (unless
       (changed?
         (+ (event-count *database-event*)
            (event-count *lock-event*)))
       (wait-for-events
         (list *database-event*
               *lock-event*)))))
```

Figure 1: Daemon in LISP

tained the error pattern which also occurred during flight, and which was found using the model checker. The actual LISP code describing the behavior of the daemon is given in Figure 1.

The daemon goes through a loop, where in each iteration it checks the lock table, comparing it to the database, and recovers any inconsistencies that may be detected (if the `check-locks` function returns `true`). After that, it goes to sleep by calling the `wait-for-events` function, which as parameters takes a list of events to wait for. Whenever one of these events is signaled, i.e. the database or the lock table is modified, the daemon will wake up and continue.

In order to catch events that occur while the daemon is executing, each event has an associated event counter that is increased whenever the event is signaled. The daemon only calls `wait-for-events` in case these counters have not changed, hence, there have been no new events since it was last restarted from a call of `wait-for-events`.

The PROMELA model of this LISP code is presented in Figure 2. The if-construct decides whether the daemon should stop and wait for a new database event or lock event to occur (call of `wait_for_events`), or whether it should continue for another iteration. Another iteration is needed if a database event or a lock event has occurred since the daemon was restarted last time; that is, in case the event counter `event_count` differs from the sum of the event counters for the database and lock events. If there is a difference, it means that there has been an event since the last time `event_count` was updated, and the daemon must perform another iteration before calling `wait_for_events`, first updating `event_count` to hold the new event counter sum.

```
proctype daemon(TaskId this) {
  byte event_count = 0;
  do
  :: check_locks_and_recover;
     if
     :: (Ev[DATABASE_EVENT].count +
         Ev[LOCK_EVENT].count
         == event_count )
            ->
         wait_for_events(this,
           DATABASE_EVENT,LOCK_EVENT)
     :: else ->
         event_count =
           Ev[DATABASE_EVENT].count +
           Ev[LOCK_EVENT].count
     fi
  od
};
```

Figure 2: Daemon in PROMELA

3.3 Stating and Verifying Properties

The model was analyzed with respect to the following two properties, here expressed informally. The *release* property reads: *"A task releases all of its locks before it terminates"*. The *abort* property reads: *"If an inconsistency occurs between the database and an entry in the lock table, then all tasks that rely on the lock will be terminated, either by themselves or by the daemon in terms of an abort"*. The release property was formulated by inserting an assertion in the code at the end of each task. This assertion stated that all locks should be released at this point. The second property was stated as a linear temporal logic property of the form:

```
[](property_broken -> <>tasks_informed)
```

This property says: whenever a property is broken, then eventually all tasks depending on this property will be informed about it (in fact terminated). The names `property_broken` and `tasks_informed` are macro names standing for predicates on the state space.

The attempted verification of the two properties led to the direct discovery of five programming errors – one breaking the release property, three breaking the abort property, and one being a non-serious efficiency problem where code was executed twice instead of once. The first four of these errors are classical concurrency errors in the sense that they arise due to processes interleaving in unexpected ways.

The error we want to focus on in this presentation is the one isomorphic to the RAX anomaly. The error caused the abort property to be violated. The error trace generated by SPIN demonstrated the following situation. The daemon is prompted to perform a check of the lock table. It finds everything consistent and checks the event counters to see whether there have been any new events while it has been running. This is not the case, and the daemon therefore decides to call `wait-for-events`. However, at this point an inconsistency is introduced, and a signal is sent by the environment, causing the event counter for the database event to be increased. This is not detected by the daemon since it has already made the decision to wait, which it then does, and the inconsistency now is not discovered by the daemon. Our suggested solution at the time was to enclose the test and the wait within a critical section, which does not allow scheduling interrupts to occur between the test and the wait. Furthermore, two other flawed code fragments violated the abort property.

The release property was violated in the sense that locks did not always get released by a task. The error trace generated by SPIN demonstrated that *during* a task's release of a lock, but before its actual release, the task may get interrupted by the daemon if the property gets broken. This means that the task terminates without releasing the lock. The error is particularly nasty in the sense that all code, *except* the lock releasing itself, had been protected against this situation: in case of an interrupt the lock releasing would be executed.

The model was verified exhaustively using SPIN's partial order reduction algorithm and state compression. Typically between $3,000$ - $200,000$ states were explored in the different models, using between 2 - 7 Mb of memory, and using between 0.5 - 20 seconds.

3.4 Discussion of Methodology

The verification effort has been regarded by all involved parties as a very successful application of model checking, and of SPIN in particular. According to the RA programming team, the effort has had a major impact, locating errors that would probably not have been located otherwise, and identifying a major design flaw.

The modeling effort, i.e. obtaining a PROMELA model from the LISP program, took about 12 man weeks during 6 calendar weeks, while the verification effort took about one week. The modeling effort consisted conceptually of an *abstraction* activity combined with a *translation* activity. Abstraction was needed to cut down the program to one with a reasonably small finite state space, making model checking tractable. Translation, from LISP to PROMELA, was needed to obtain a PROMELA model that the SPIN model checker could analyze.

The abstraction was done without any knowledge about the properties to be verified, since these were stated later. The abstraction maintained important operations on the lock table and ignored most other details of the original LISP program, hence, a kind of program slicing. No formal attempt was made to show that the abstractions preserved errors. It is interesting that such an ad hoc approach still was extremely effective. The translation phase was non-trivial and time consuming due to the relative expressive power of LISP when compared with PROMELA.

Based on these observations, two research efforts were initiated that should make application of model checking within the software development cycle less resource demanding. In one effort a translator from the Java programming language to PROMELA has been developed; see Section 4.2. In another effort, an abstraction tool has been developed, which can perform so-called predicate abstractions on Java programs; see Section 5. Both tools have been applied in the verification of the RA as described in the following.

4 Formal Analysis After Flight

Shortly after the anomaly occurred during the Remote Agent Experiment, on Tuesday May 18, the ASE team at NASA Ames heard that something had broken down in the RA while it was in control of the spacecraft and offered their help to the RAX team. On Friday morning, after a few email exchanges, the RAX team provided access to the source code of the Executive, without identifying where the error was, and offered the ASE group the challenge of seeing "how long it would take for formal methods to come up with it".

On Friday afternoon, we decided to run a "clean room" experiment to determine whether or not the technology currently used and under development in the group *could* have discovered the bug before it actually happened. At that time, we knew that debugging information collected from the spacecraft had enabled the DS-1 team to identify the bug and continue the experiment, and that the failure had something to do with a "handshaking" communication between a Planner process and an Executive process.

Other than these messages we had no further information, and no one in the ASE group had any contact with RAX personnel during that week.

This section first describes how the experiment was conducted. Then the Java PathFinder translator that was used to model check the flawed code is described. This is followed by a description of the error and how it was found using Java PathFinder. We conclude with a discussion of the methodology that has been followed.

4.1 The Clean Room Experiment

To make this clean room experiment credible, we decided that we would need to complete this exercise over the weekend, prior to the return of the RAX team from the DS-1 mission control at JPL the following Monday. This was both to avoid undue influence by people familiar with the details of the bug, and also to meet the "short-turnaround" challenge, mimicking what would be required if we were actually called on to provide "on-line" assistance.

The experiment was set up as follows. A *front-end* group would try to spot the error by human inspection, or at least identify problematic parts of the code. On the basis of that, it would extract a more or less self contained portion of the code containing the problematic code portions, of a tractable size for a model checker. This extracted code would then be handed over to the *back-end* group without any hints as to what could be the error. The back-end group would then try to locate the error using model checking. The situation was comparable to someone doing visual inspection of code, and finding suspect sections which he wanted to explore further.

The front-end team began perusing the code on Friday afternoon, and extracted roughly 700 lines containing questionable code[2]. The full group met again on Saturday afternoon, and the front-end team gave the back-end team the extracted code. In accordance with the design of the experiment, they did not tell where the suspected bug was, but they briefed the back-end team on the control and data structures of the extracted code. The back-end group spent most of the time understanding that code in order to model it, and on Sunday morning came out with a fairly abstract model of the suspicious code. That model was written in Java and verified with the Java model checker Java PathFinder, as described below. It reported a deadlock, which turned out to be the one that had happened in flight five days before.

[2]Though they were not sure that they had indeed captured the concurrency error.

4.2 The JPF Translator

Java PathFinder (JPF) [8, 6] is a translator from a non-trivial subset of Java to PROMELA. Given a Java program, JPF translates this into a PROMELA program, which then can be model checked using SPIN. Java is an object-oriented programming language with a built-in notion of threads. Objects are instantiated dynamically from classes, which can be defined using single class inheritance. Threads, which are special objects with an activity, can communicate by making calls to methods defined in shared objects. Such methods can be defined as synchronized, thereby turning these shared objects into monitors, allowing only one thread to operate in the object at a time.

In the default mode, the SPIN model checker will find any deadlocks present in the Java program. Such deadlocks can occur when several threads compete for access to the monitors. Properties can also be formulated explicitly by the user, either as assertions in the program, or as linear temporal logic formulae. That is, a Java program can be annotated with assertions written as calls to a special `assert` method which takes a boolean argument expression over the variables in the Java program. Any such call is translated into a corresponding PROMELA assertion, which will then be checked during the state space exploration whenever reached. Finally, SPIN's own linear temporal logic can be used to formulate properties over the Java program's static variables (a static variable in Java is defined within a class, but is only allocated once, and hence is shared between all objects of the class).

A significant subset of Java is supported by JPF: dynamic creation of objects with data and methods, static variables and static methods, class inheritance, threads and synchronization primitives for modeling monitors (synchronized statements, and the `wait` and `notify` methods), exceptions, thread interrupts, and most of the standard programming language constructs such as assignment statements, conditional statements and loops.

The translator is written in 6000 lines of LISP, and was developed over a period of 8 months. JPF has been applied to a number of case studies, amongst them a 1500 line game server [9], a NASA file transfer protocol for satellites, and a NASA data transmission protocol for the space shuttle ground control.

A related attempt to provide model checking technology for Java is described by Demartini et. al. [5], which also translates Java programs into PROMELA. However, their approach does not handle exceptions or polymorphism as does Java PathFinder. In another related approach, Corbett [4] describes a theory of translating Java to a transition model, making use of static pointer analysis to aid *virtual coarsening*, which reduces the size of the model.

4.3 The RAX Error

The suspected and eventually confirmed error was a missing critical section around a conditional wait on an event. The relevant piece of code (anonymized for confidentiality purposes) is shown in Figure 3.

```
   (loop
     (when
(*1*)  (or (/= count (esl::event-count event1))
(*2*)      (warp-safe (wait-for-event event1)))
       (setf count (esl::event-count event1))
       ; ...
(*3*)  (signal-event event2)))
```

Figure 3: The RAX Error in LISP

This is the body of one of the concurrent tasks and consists of a loop. The loop starts with a `when` statement whose condition is a sequential-or statement[3] that states: *if the event counter has not been changed (`*1*`), then wait (`*2*`), else proceed*. This behavior is supposed to avoid waiting on the event queue if events were received while the process was active. However, if the event occurs between (`*1*`) and (`*2*`), it is missed and the process goes asleep. Because the other process that produces those events is itself activated by events created by this one in (`*3*`), both end up waiting for each other, a deadlock situation.

This follows a similar pattern to the code shown in Figure 1 that had been identified as a source of error during the verification of the Executive in 1997, as described in Section 3.3. This similarity was spotted by members of both the front-end and back-end teams, and contributed greatly to narrowing down the verification effort to this particular potential problem.

[3] `(or X Y)` is evaluated like `if X then true else Y`.

4.4 Demonstrating the Error with JPF

The modeling focused on the code under suspicion for containing the error. The major two components to be modeled were events and tasks, as illustrated in Figure 4. The figure shows a Java class `Event` from which event objects can be instantiated. The class has a local counter variable and two synchronized methods, one for waiting on the event and one for signaling the event, releasing all threads having called `wait_for_event`. Note how the counter is incremented by `signal_event` in order to allow the tasks to check whether new events have arrived. The increment is modulo 3 in order to reduce the state space to be searched by the model checker. This is an informal abstraction in the sense that it has not been proven to preserve errors. Section 5 explains how an alternative counter abstraction for this program can be made and automatically proved correct.

```
class Event{
  int count = 0;

  public synchronized void wait_for_event(){
    try{wait();}catch(InterruptedException e){};
  }

  public synchronized void signal_event(){
    count = (count + 1) % 3;
    notifyAll();
  }
}


class FirstTask extends Thread{
  Event event1,event2;
  int count = 0;

  public void run(){
    count = event1.count;
    while(true){
      if (count == event1.count)
        event1.wait_for_event();
      count = event1.count;
      event2.signal_event();
    }
  }
}
```

Figure 4: The RAX Error in Java

Figure 4 also shows the definition of one of the tasks. This is an abstraction (in Java) of the LISP code presented in Figure 3. The task's activity is defined in the `run` method of the class `FirstTask`, which itself extends the `Thread` class, a built-in Java class that supports thread primitives. The body of the `run` method contains an infinite loop, where in each iteration a conditional call of `wait_for_event` is executed. The condition is that no new events have arrived, hence the event counter is unchanged. After having applied JPF, the SPIN model checker revealed the deadlock situation described in Section 4.3. In the Java context a new event arrived after the test `(count == event1.count)`, but before the call `event1.wait_for_event()`.

4.5 Discussion of Methodology

The formal analysis of the Executive after the occurrence of the anomaly was preceded by a code inspection, which identified the *possible* source of the error. Some of us spotted the potential error situation because it resembled the similar error we had found using SPIN in 1997, as described in Section 3.3. Due to the focus on the particular code fragment, it was relatively easy to perform the abstraction needed to extract a Java program with a small finite state space. This took about two hours. However, the suspicion was only a suspicion, and a demonstration that the code was flawed was provided using JPF. This showed the usefulness of using a model checker to answer focused queries.

Since the original source code was in LISP, we still had to translate it by hand in Java, which goes against JPF's intended purpose. To avoid that, one would need an abstraction tool and a translator for LISP. Since LISP's future within NASA is questionable we have focused on providing these technologies for Java. Java is a very convenient modeling language, providing most of the high level features of the powerful Common LISP Object System (CLOS), such as dynamically created objects with methods and data. The major experience with all experiments done with JPF are obviously that a non-trivial amount of abstraction is needed in order to reduce the size of a program's state space. This problem is addressed in Section 5.

5 An Abstraction Tool for Java

As a part of the JPF project, we have been developing an automated abstraction tool which converts a Java program to an abstract program with respect to user-specified abstraction criteria. The user can specify abstractions by removing variables in the concrete program and/or adding new variables (currently the tool supports adding boolean types only) to the abstract program. Given a Java program and such abstraction criteria, the tool generates an abstract Java program in terms of the new abstract variables and unremoved concrete variables. To compute the conversion automatically, we use a decision procedure, SVC (Stanford Validity Checker), which checks the validity of logical expressions [1].

The abstraction tool is designed to deal with object-oriented programs. The user can specify abstraction criteria for each class by removing field variables in the class and/or adding new abstract variables to the class. Therefore, it can be used to abstract subcomponents in a program when the whole program is too complicated to apply abstraction globally. In addition, the user can specify new abstract variables which depend on variables from two different classes (inter-class abstraction).

There has been similar work by others [3, 15], all of which require use of only global variables to describe a system in simple languages similar to guarded commands. However, our tool targets a real programming language Java and is able to deal with many problems caused by its object-orientation.

5.1 Application of the Tool to the RA

As we do not have enough space in this paper for a detailed explanation of the abstraction algorithm, let us illustrate the abstraction performed by the abstraction tool on a part of the RA Java code shown in Figure 4. As stated before, state explosion occurs because of the unbounded increase of the count variable in the `Event` class (in the original LISP code) and the assignment of the `count` variable in the `FirstTask` class (as well as in the `SecondTask` class which is not shown). Therefore, we use abstraction to remove those `count` variables by specifying `Abstract.remove(count)` in the classes of `Event` and `FirstTask`. In place of these variables, we add new abstraction predicates which appear in the program with the `count` variables. For instance, we put `Abstract.addBoolean("FcntEqEcnt", count==event1.count)` in the definition of the `FirstTask` class to specify an abstraction predicate: `FirstTask.count` is equal to `Event.count` (For implementation convenience, object names are used to refer to class types.). We also used more inter-class abstractions such as `FcntGeEcnt` (`FirstTask.count` is greater than or equal to `Event.count`), `ScntEqEcnt`

(SecondTask.count is equal to Event.count), etc.

This is an example of an inter-class abstraction. Dealing with such inter-class abstractions is more involved than dealing with the abstractions inside one class. For each inter-class abstraction, the tool generates an additional class definition in the abstract program, which contains new boolean variables corresponding to the specified predicate. The boolean variables in the new class are defined as a two-dimensional array where each index refers to an object in either of the two classes. In Figure 5, the new abstract variable FcntEqEcnt.pred[Fobj][Eobj] corresponds to the user-defined predicate FcntEqEcnt for an object Fobj of FirstTask class and an object Eobj of Event class, i.e., Fobj.count = Eobj.count.

Given the abstraction criteria, we now need to compute the value of the abstract variables in the abstract program so that they are consistent with the values of concrete variables in the program. Figure 5 shows how the abstraction tool converts the assignment statement, count = count + 1 (without the modulo operation) in Figure 4. First, the concrete assignment statement is omitted in the abstract program because the variable to be assigned has been removed. Instead, the tool checks which of the new abstract variables are possibly affected by this assignment and generates corresponding assignments to those abstract variables. For the example statement, a set of boolean variables that refers to 'this' Event object will be affected: FcntEqEcnt.pred[i][this] in Figure 5 (Actually, we use functions that return the corresponding index of a given object). To update those abstract variables, a for-statement is used. For each of the abstract variables, the pre-images that leads the abstract variable to be true (or false) by the assignment are computed. Then the pre-images are mapped into the abstract domain by checking validity of the corresponding logical expressions. Finally, the results are used as a guard condition to set the abstract variables to true (or false). In the example, the variable FcntEqEcnt.pred[i][this] will be set to false if it was true (or if some condition with another abstract variable holds). Otherwise, the variable is set to a non-deterministic boolean value. Because the concrete assignment statement is regarded as atomic, a set of these abstract assignments are declared as atomic for the JPF model checker. The additional statements for updating other abstract variables such as FcntGeEcnt are not shown in the figure.

```
Verify.beginAtomic();
// count = count + 1;
for(int i = 0; i < FcntEqEcnt.numFirstTask; ++i){
  if(FcntEqEcnt.pred[i][FcntEqEcnt.getEvent(this)]
     || FcntGeEcnt.pred[i][FcntGeEcnt.getEvent(this)])
    FcntEqEcnt.pred[i][FcntEqEcnt.getEvent(this)] =
      false;
  else FcntEqEcnt.pred[i][FcntEqEcnt.getEvent(this)]
       = Verify.randomBool();
}

// similar code for updating other inter-class
// abstract variables such as FcntGeEcnt, etc.
Verify.endAtomic();
```

Figure 5: Output of the abstraction tool for the assignment statement

5.2 Discussion of Methodology

Using the tool, we have been able to obtain an abstract Java program of the RA code automatically. In the example, the unbounded integer variables are replaced by a set of boolean variables, hence the abstract program is free from the state explosion. Moreover, use of the tool helps to avoid error-prone abstractions based on human reasoning. The tool generates a sound approximation of the concrete program using an automated validity checker, although it is not necessarily the most accurate one.

However, the user must give reasonable abstraction criteria for the tool to generate a meaningful abstract program in order to check some desired properties. In case the abstraction criteria are not good enough, the result will be a too rough abstract program which can not preserve the properties to be checked.

6 Conclusion

This paper describes two major verification efforts carried out within the Automated Software Engineering Group at NASA Ames Research Center. The first effort consisted of analyzing part of the RA autonomous space craft software using the SPIN model checker. One of the errors found with SPIN, a missing critical section around a conditional wait statement, was in fact reintroduced in a different subsystem that was not verified in this first preflight effort. This error caused a real deadlock in the RA during flight in space.

Such concurrency-related errors only happen as the result of particular scheduling circumstances. Scheduling is totally uncontrolled when tests are run, and is highly sensitive to variations in the operating environment (e.g. operating system, other running tasks). This explains why the anomaly happened in flight, though it had not occurred even once in thousands of previous runs on the various ground testbeds.

Developing the formal model of the program was, however, a time consuming task, requiring a manual translation from the RA LISP code to the PROMELA language of the SPIN model checker. In addition, code details had to be abstracted away in order to obtain a small enough finite state system that could be effectively model checked. The translation difficulty spawned the initiative to automate the translation from high level programming languages to modeling languages for formal verification, such as PROMELA. Java was chosen as the source language because of its modern programming language constructs, such as support for object-oriented programming, and the standardization across implementations of its concurrency constructs. An automatic translator from Java to PROMELA was designed and implemented, called Java PathFinder (JPF). With JPF one can model check smaller Java programs for assertion violations, deadlocks, and general linear temporal logic properties. The translator covers a substantial subset of Java, illustrating the feasibility of the approach.

In the second effort, JPF was used for modeling the RAX deadlock after it occurred. That is, after the front-end team isolated a reduced subset of the code that likely included the error, the back-end team developed a Java program which exposed the error. The translator translated this into a PROMELA model, and the model checking of this model then immediately revealed the error. Java turned out to be an excellent choice as a modeling language, with a high level of abstraction, due to its object oriented features. In later work, a system that automates certain aspects of predicate abstraction was developed and successfully demonstrated on the same example.

This experience gave a clear demonstration that model checking can locate errors that are very hard to find with normal testing and can nevertheless compromise a system's safety. It stands as one of the more successful applications of formal methods to date. In its report of the RAX incident, the RAX team indeed acknowledges that it "provides a strong impetus for research on formal verification of flight critical systems" [13].

A posteriori, given the successful partial results, one can wonder why the first verification effort was not extended to the rest of the Executive, which might have spotted the error before it occurred in flight. There are two reasons for that. First, the purpose of the effort was to evaluate the verification technology, not to validate the RA. The ASE team did not have the mission nor the resources needed for a full-scale modeling and verification effort. Second, the part of the code in which the error was found has been written after the end of the first verification experience.

Regarding software verification, the work presented here demonstrates two main points. First of all, we believe that it is worthwhile to do source code verification since code may contain serious errors that probably will not reveal themselves in a design. Hence, although design verification may have the economical benefit of catching errors early, code verification will always be needed to catch errors that have survived any good practice. Code will always by definition contain more details than the design – any such detail being a potential contributor to failure.

Second, we believe that model checking source code is practical. The translation issue can be fully automated, as we have demonstrated. The remaining technical challenge is scaling the technology to work with larger programs - programs that could have very large state spaces unless suitably abstracted. Abstraction is of course a major obstacle, but our experience has been that this effort can be minimized given a set of supporting tools.

Acknowledgments

We would like to thank Erann Gat, the developer of ESL, for his useful responses to our error reports. We also want to thank Ron Keesing and Barney Pell, of the RA programming team, for explaining parts of the Executive and suggesting properties to be verified. We also appreciate Pandu Nayak, Kanna Rajan, Gregory Dorais, and Nicola Muscettola for their comments on our second verification effort. Finally, but certainly not least, we want to thank SPIN's designer, Gerard Holzmann, for his always reliable support during the work.

References

[1] C. Barrett, D. Dill, and J. Levitt. Validity Checking for Combinations of Theories with Equality. In *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996.

[2] D. Bernard et al. Spacecraft Autonomy Flight Experience: The DS1 Remote Agent Experiment. In *Proceedings of the AIAA 1999, Albuquerque, NM*, 1999.

[3] M. Colón and T. Uribe. Generating Finite-State Abstractions of Reactive Systems using Decision Procedures. In *Proceedings of the 10th Conference on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, July 1998.

[4] J. Corbett. Constructing Compact Models of Concurrent Java Programs. In *Proceedings of the ACM Sigsoft Symposium on Software Testing and Analysis*, March 1998. Clearwater Beach, Florida.

[5] C. Demartini, R. Iosif, and R. Sisto. Modeling and Validation of Java Multithreading Applications using SPIN. In *Proceedings of the 4th SPIN Workshop*, November 1998. Paris, France.

[6] K. Havelund. Java PathFinder, A Translator from Java to Promela. In *Theoretical and Practical Aspects of SPIN Model Checking – 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, July and September 1999. Trento, Italy – Toulouse, France (presented at the 6th Workshop).

[7] K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. In *Proceedings of the 4th SPIN workshop, Paris, France*, November 1998. To appear in IEEE Transactions of Software Engineering.

[8] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. To appear in a special issue of *International Journal on Software Tools for Technology Transfer* (STTT) containing selected submissions to the 4th SPIN workshop, Paris, France, 1998, February 1999.

[9] K. Havelund and J. Skakkebæk. Applying Model Checking in Java Verification. In *Theoretical and Practical Aspects of SPIN Model Checking – 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, July and September 1999. Trento, Italy – Toulouse, France (presented at the 6th Workshop).

[10] G. Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[11] N. Muscettola. *HSTS: Integrating Planning and Scheduling*. Morgan Kaufman, 1994.

[12] N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.

[13] P. Nayak et al. Validating the DS1 Remote Agent Experiment. In *Proceedings of the 5th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS-99)*. ESTEC, Noordwijk, 1999.

[14] B. Pell, D. Bernard, S. Chien, E. Gat, N. Muscettola, P. Nayak, M. Wagner, and B. Williams. An Autonomous Spacecraft Agent Prototype. *Autonomous Robots*, 5(1), March 1998.

[15] H. Saïdi and N. Shankar. Abstract and Model Check While You Prove. In *Proceedings of the 11th Conference on Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer-Verlag, July 1999.

[16] B. Williams and P. Nayak. A Model-Based Approach to Reactive Self-Configuring Systems. In *Proceedings of AAAI-96*, 1996.