Experimental Evaluation of Verification and Validation Tools on Martian Rover Software

Guillaume Brat, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, Rich Washington

NASA Ames Research Center

Abstract. We report on a study to determine the maturity of different verification and validation technologies (V&V) on a representative example of NASA flight software. The study consisted of a controlled experiment where three technologies (static analysis, runtime analysis and model checking) were compared to traditional testing with respect to their ability to find seeded errors in a prototype Mars Rover. What makes this study unique is that it is the first (to the best of our knowledge) to do a controlled experiment to compare formal methods based tools to testing on a realistic industrial-size example where the emphasis was on collecting as much data on the performance of the tools and the participants as possible. The paper includes a description of the Rover code that was analyzed, the tools used as well as a detailed description of the experimental setup and the results. Due to the complexity of setting up the experiment, our results can not be generalized, but we believe it can still serve as a valuable point of reference for future studies of this kind. It did confirm the belief we had that advanced tools can outperform testing when trying to locate concurrency errors. Furthermore the results of the experiment inspired a novel framework for testing the next generation of the Rover.

1 Introduction

To achieve its science objectives in deep space exploration, NASA has a need for science platform vehicles to autonomously make control decisions in a time frame that excludes intervention from Earth-based controllers. Round-trip light-time is one significant factor motivating autonomy capabilities, another factor is the need to reduce ground support operations cost. An unsolved problem potentially impeding the adoption of autonomy capabilities is the verification and validation of such software systems, which exhibit far more behaviors (and hence distinct execution paths in the software) than is typical in current deep-space platforms. Hence the need for a study to benchmark advanced Verification and Validation (V&V) tools on representative autonomy software.

The objective of the study was to assess the maturity of different technologies, to provide data indicative of potential synergies between them, and to identify gaps in the technologies with respect to the challenge of autonomy V&V.

The study consists of two parts: first, a set of relatively independent case studies of different tools on the same autonomy code, second a carefully controlled experiment with human participants on a subset of these technologies. This paper describes the second part of the study. Overall, nearly four hundred hours of data on human use of three different advanced V&V tools were accumulated, with a control group that used conventional testing methods. The experiment simulated four independent V&V teams debugging three successive versions of an executive controller for a Martian Rover. Defects were seeded into the three versions based on a profile of defects from CVS logs that occurred in the actual development of the executive controller.

The experiment evaluates tools representing three technologies: static analysis, model checking, and runtime analysis; and compared them to conventional testing methods. The static analysis tool is the commercial PolySpace C-verifier [13]. This tool analyzes a C program without executing it; it focuses on finding errors that lead to run-time faults such as underflow/overflow, non-initialized variables, null pointer de-referencing, and array bound checking. The model checking tool is Java PathFinder (JPF) [15], which is an explicit-state model checker that works directly on Java code. JPF specializes in finding deadlocks, verifying assertions, and checking temporal logic specifications. JPF explores all possible interleavings in multi-threaded Java programs. The runtime analysis tools are Java Path Explorer (JPaX) [12] and DBRover [9]. JPaX can infer potential concurrency errors in a multi-threaded program by examination of a single execution trace. Amongst the errors detectable are deadlocks and data races. DBRover supports conformance checking of an execution trace against a specification written in metric temporal logic.

The rest of the document is structured as follows. We discuss related work in section 2. In section 3 we discuss the Rover Executive that we analyzed in the experiment as well as the correctness requirements this software should satisfy. The methodology used for setting up the experiment is given in section 4, followed by a description of the V&V tools used in section 5. The results obtained from the study are divided between qualitative (section 6) and quantitative (section 7) results. Concluding remarks are given in section 8.

2 Related Work

There is relatively little work on comparative studies of different formal methods tools, and their relationship to traditional methods. The reason for this is likely the fact, that this kind of comparative work is hard, and for many researchers perhaps not as interesting as developing theory and tools. Early work seemed to be focused on analyzing and comparing formal specification languages and refinement environments (which refine code from specifications). With the arrival of model checkers, work has been done on comparing these with the purpose of evaluating how well bugs are located. Only recently has work appeared, which relates different bug-locating tools. We will outline a few references in each of these categories.

Amongst early work on comparing a formal method based on a formal specification language to standard practice is the much celebrated CICS experiment [7], where the Z specification language was used to specify IBM's CICS transaction system (Customer Information Control System). The project was so successful that Oxford University Computing Laboratory and IBM received the Queens's Award for Technological Achievement. The use of Z reduced development costs significantly and improved reliability and quality. It was estimated that IBM was able to reduce their costs for the development by almost five and a half million dollars. In addition, the quality was claimed higher based on feedback from users. As an example of a study comparing specification languages, an international Dagstuhl seminar was organized in 1995 as a competition between different researchers, each representing a formal method/specification language. The participants were all asked to specify a steam-boiler problem based on an informal requirement description. The result is presented in [1].

In [18] a more recent experiment is described comparing the state-of-the-art formal specification and code generation method Specware (combined with the functional programming language Haskel) with a state-of-the-practice waterfall method rated at Capability Maturity Model (CMM) level 4. Two teams were established, which concurrently applied their method, in which they were specialist, to the given problem. The goal was to develop a real-time Personal Access Control System (PACS) (badge reader) system. Both parties were given the same requirement specification, in which two errors had been seeded. An independent third party evaluated the results at the end, using extensive testing. The formal methods project seemed to perform better. They found the two errors, in contrast to the CMM team, and produced code with a reliability of 0.77 (23 failures within 100 test cases), whereas the CMM team produced code with reliability 0.56. Had the formal methods team corrected the two errors they found, which they did not, they would have had a reliability of 0.98. None of the teams satisfied the original requirements completely.

With the increased focus on formal methods tools, some work has been done on comparing such tools. A comparison of four different finite-state verification tools is for example described in [4]. The tools were the model checkers SPIN and SMV, INCA, that performs necessary conditions analysis, and FLAVERS, that performs data flow analysis of Ada programs. The verification tools were applied to the Chiron user interface system, a real Ada program of substantial size. A tool automatically translated the Ada program into the input languages of the respective verification tools. After the translation, different researchers applied the different tools, and comparisons were made with respect to time and memory. It was concluded that the results were very dependent on the translation schemes from Ada to the input notations. Other authors have also compared model checking tools [10, 6, 8].

The work in [2] describes an analysis of five different formal methods tools: the model checker Rivet, the temporal logic runtime monitoring tool MaC, the concurrency runtime analysis tool VisualThreads, the static analysis and theorem proving tool ESC/Java and finally the static analyzer Jlint. The goal was to

identify the most practical tool for finding defects in software developed by the company in which the author was an intern. Each tool was examined on fifteen test-cases, which were created for this particular experiment, and which represented small well-known errors that can occur in multi-threaded programs. The 15 test-cases were created based on a statistical analysis of a large body of code. Not surprisingly, the different tools had different weaknesses and strengths. The conclusion for static checkers was that the simple checkers worked as well as the complex checkers. The VisualThreads tool worked very nicely but did not work for Java programs. The MaC tool did not deal with multi-threading.

3 Target Software and Verification Requirements

3.1 Rover Executive

The NASA Ames K9 rover is an experimental platform for autonomous wheeled vehicles called rovers, targeted for the exploration of a planetary surface such as Mars. K9 is specifically used to test out new autonomy software, such as the Rover Executive [17]. Previous to the development of autonomy software, planetary rovers were controlled through sequences of detailed, low-level commands uploaded from Earth. The Rover Executive provides a more flexible means of commanding a rover through the use of high-level plans. Plans are programs written in a language that specify actions and constraints on the movement, experimental apparatus, and other resources of the Rover. The operational semantics of this language take into account the possibility of failure of atomic-level command actions. The Rover Executive is a software prototype written in C++ by researchers at NASA Ames.

The Rover executive is approximately 35K lines (Kloc) of C++, of which 9.6 Kloc are related to core functionality and the rest is for data structure manipulation, modules for specific rovers and science instruments, and research-related extensions. Here the main focus is on the core functionality. Because the V&V tools benchmarked in this experiment analyze programs written in Java and C, the C++ code was manually translated into these languages. Due to resource constraints, the translation was selective; some components of the executive were stubbed. The translated Java version is approximately 7.3 Kloc. The C version consists of approximately 4.5 Kloc.

The Rover Executive is essentially an interpreter for the plan language. The executive also monitors execution of primitives, and performs appropriate responses and cleanup when the execution of a primitive fails. A plan consists of nodes representing high-level control structures or primitive plan elements called *tasks*. The high-level control structures of the plan are sequential composition (*block*) and conditional execution (*branch*). Primitive tasks command the vehicle to perform an action that is monitored by the Rover Executive. Associated with nodes are conditions. A node succeeds or fails based on evaluation of these conditions. Four types of conditions may be specified: *wait-for*, *start*, *maintain*, and *end* conditions. A *wait-for* condition is monitored prior to the execution of a node; the node cannot begin execution until its *wait-for* condition is satisfied. A

```
(block
  :id plan
  :node-list (
      (task
         :id drive1
         :end-conditions   (time +0 +20)
         :action action1
         :continue-on-failure

      )
      (task
         :id drive2
         :action action2
         :start-conditions (time +0 +10)
         :end-conditions   (time +1 +30)
      )
   )
)
```

Fig. 1. Example Plan

start condition is a pre-condition on the execution of node. If the *start* condition evaluates to false then execution of the node fails. The *maintain* condition is monitored during execution of the node; should the *maintain* condition at any time evaluate to false, the node fails. The *end* condition is evaluated at the end of execution of the node; if the *end* condition evaluates to false, the node fails.

Conditions are conjuncts of atomic conditions. There are two types of atomic conditions. One type of atomic condition is a predicate over values in a main-memory database. The database is an environment mapping names to values. Typically the database is updated with values provided by the controlled system, e.g. the vehicle's current state. In addition, the Executive uses the database to record and manage its own state. The second type of condition is a temporal condition that specifies a time interval, using relative time (with respect to a plan element) or absolute time (relative to the start of execution of the plan). For example, the start of a task can be constrained to a relative or absolute time interval.

Execution of a node either succeeds or fails and the outcome affects further execution of the plan. Associated with each node is a Boolean attribute *continue-on-failure*. If the *continue-on-failure* attribute is false then failure of a node will propagate up and cause the containing node (or the plan itself) to fail. When a node fails its execution terminates. Thus if a node n is a sequential composition of nodes, $n_1; \ldots; n_k$ and node n_i fails then n immediately fails and nodes $n_{i+1}; \ldots; n_k$ are not executed. Similarly if the execution of an alternative of a conditional node fails, the conditional node itself fails. If the flag is true, then failure of a sub-node does not cause the node to fail. Hence in the case of sequential composition, failure of node n_i with *continue-on-failure* set to true,

the executive will initiate execution of node n_{i+1}. Figure 1 is an example of a plan.

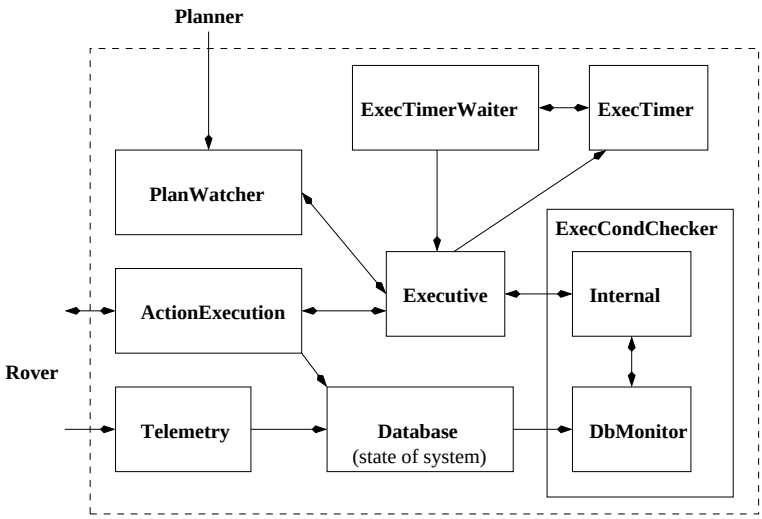

Fig. 2. Rover Executive Architecture.

The architecture of the Rover executive is illustrated in Figure 2. Generally each component in Figure 2 executes as a separate thread. The *Executive* is the central component interpreting the plan and controlling execution of plans (received from a Planner via the *PlanWatcher*). The *ExecTimer* and *ExecTimer-Waiter* provide a timer capability used to support temporal conditions. These conditions are processed by the Executive, which then posts timer requests to the *ExecTimer* thread. The *ExecTimerWaiter* signals the Executive when posted time points are reached. The Database is an environment holding name-value pairs. The *ActionExecution* thread sends commands to the rover and monitors the status of action execution on the rover, updating the Database to reflect changes to that status. The *Telemetry* thread monitors the state variables of the external system and updates the Database. The *ExecCondChecker* (composed of two separate threads) monitors changes in the database (*DbMonitor*) and prioritizes the changes and signals the Executive (*Internal*). Changes in the database or the passing of posted time points, events that may lead to the change of valuation of a condition, are signaled to Executive. Should these changes cause a node to succeed or fail, the Executive advances execution thorough the plan according to the semantics described above. Execution of a primitive task causes the Executive, via the *ActionExecution* thread, to command the vehicle to perform the task's action.

The Java version (7.3 Kloc) used for this experiment contains all the components of Figure 2, except the *Telemetry* thread. The *Database* and *PlanWatcher* were simplified versions of the original C++ code: *Database* only contained a small subset of variables in order to preserve the communication behavior and similarly *PlanWatcher* only contained the mechanism to communicate the fact that a new plan was available. The C version (4.5 Kloc) used for static analysis was very similar to the Java version in terms of its general structure (i.e. used the same components), but used further simplified code, e.g. the exception mechanisms were not treated. The difference in size between the two versions is mostly accounted for in C being somewhat more succinct than Java.

3.2 Autonomy Requirements

Due to speed-of-light, bandwidth and energy considerations, low-level commanding of a Rover on the surface of Mars severely limits the amount of tasks a Rover can perform in a day and decreases the science return of the mission. Thus it is NASA's goal to increase the autonomy of spacecraft. For example an objective of one mission profile for the Mars Smart Lander mission, scheduled for launch in 2009, is to command the Rover to travel distances as long as one kilometer. This requires complex energy planning, terrain mapping, obstacle avoidance, localization, etc.

This in turn may require use of heuristic algorithms and other Artificial Intelligence programming techniques. At the same time flight software has extreme reliability requirements. Thus verification and validation of autonomy flight software is of prime importance. The methodology used previously involved extensive testing with simulations that represent nominal and slightly-off-nominal flight scenarios. However for highly autonomous software, the number of scenarios is much larger, limiting the effectiveness of traditional testing. In this context, mathematical methods, which have the potential of proving at once system properties for large classes of input scenarios, become attractive.

Concurrency is a necessary attribute of highly autonomous software, for two reasons. The first is that robust autonomy software responds to many environmental variables that change asynchronously, including status of mechanical systems, features of the terrain, obstacles, unplanned science opportunities, etc. Thus V&V of autonomy software entails analyzing the parallel composition of a software system and an environment running concurrently and changing asynchronously. Second, because of the logical complexity of autonomy software, such as the need to initiate and monitor concurrent streams of control, autonomy software is typically written as a set of interacting threads or processes. The number of threads in deep-space flight software has grown very substantially over the last decade, and is expected to continue to grow in the future. The next mission to Mars, the Mars Exploration Rover, has over a hundred threads. The Rover executive studied in this paper demonstrates many of the characteristics of autonomy software, especially the substantial use of concurrency.

Properties for Verification of Rover Software A simple definition of correctness of the rover implementation is that plans are executed correctly. That is, given a plan, the rover shall execute that plan according to its intended semantics, which is given informally in various documents describing the plan language. As an example, consider the plan provided in Figure 1. It consists of a top-node (a block) named "plan", which is decomposed into two sub-nodes (tasks), "drive1" and "drive2". The corresponding actions to be executed are called "action1" and "action2". The task named drive1 is supposed to terminate between 0 and 20 seconds after it has started. The task named drive2 is supposed to start between 0 and 10 seconds after drive1 terminates and drive2 is expected to terminate itself between 1 and 30 seconds after it starts. Since the continue-on-failure flag is set, in case drive1 fails, drive2 should be executed anyway. The just presented informal description is part of the semantics of this plan, and any execution of the plan should satisfy this.

The (partial) semantics can be formalized by expressing it in temporal logic. The following formulae express some of the informal semantics stated above:

```
P1 : [](end("drive1") -> <0,10>start("drive2"))

P2 : [](start("drive2") -> (<1,30>success("drive2") or <>fail("drive2")))
```

The formula P1 states that it is always the case (`[]`) that when drive1 ends (successfully or by failure) then (`->`) eventually (`<>`) between 0 and 10 seconds drive2 should start. Formula P2 states that after drive2 starts, it should either eventually terminate successfully within 1 to 30 seconds, or eventually fail. These two formulae only constitutes a subset of the formulae that can be written for this particular plan.

The rover can violate the plan semantics in numerous ways, which we of course cannot exhaustively cover here. However, some of the tools examine a program for particular kinds of bugs, or rather coding errors, that can cause a program to fail its goal. Deadlocks can for example cause the rover to halt execution in the middle of a plan. Data races can cause unexpected concurrent accesses to variables that may alter the outcome. Null pointer references, uninitialized variables and indexing out of array bounds may cause the program to terminate abruptly during execution.

4 Methodology

This section describes the design of the methodology for conducting the controlled experiment. The main reference for the experimental design with human subjects is [1].

The experiment had the following goals:

1. Evaluate relative strengths and weaknesses of traditional and advanced verification and validation approaches and tools on autonomy software; potentially leading to a methodology for their combined use in the software life-cycle;

2. Determine whether the current state-of-the-art gives evidence that advanced verification techniques have a potential for significant improvement over traditional ones.

In this experiment, the four different methods for V&V of representative autonomous software, were:

1. traditional testing (TT)
2. static analysis (SA);
3. model checking (MC)
4. runtime analysis (RA).

4.1 Code Preparation - Defects and Defect Seeding

In order to be able to guarantee the existence of at least a few bugs of different categories in the code for the experiment, e seeded bugs in the current version of the code. This facilitated the control and understanding of the results of the experiment. We considered two options: first, to seed bugs randomly in the code, and second, to re-seed old bugs retrieved from the developer's CVS logs. We opted for the second choice because, as pointed out by Barry Boehm and Daniel Port in [5], it has the advantage of being unbiased since the bugs seeded have been real defects. The shortfall is that such bugs are likely to be the most detectable defects using traditional techniques. However, we expected that the code would still contain live and harder to detect bugs, on which the tools could also demonstrate their relative strengths.

We modified the code so that the specific actions of the Executive to be observable during execution. For example a task starting, failing, etc. were made observable by adding print statements to the code. This allowed the observers and participants in the experiment to have a common framework for understanding the program execution and the errors being reported.

Bug classification We classified the bugs into three categories: deadlocks, data races, and plan execution defects. Deadlocks and data races are bugs that can appear or be detected irrespective of the specific input plan. Moreover, no specific properties or assertions need to be provided for these kinds of defects; they are generic errors that runtime analysis and model checking tools can often detect. Plan bugs are bugs that manifest themselves only when specific plans are used as an input, and when the semantics of the plan language is violated with the program execution.

Number of bugs seeded A total of 12 bugs were extracted from the CVS logs, of which 5 were deadlocks, 2 were data races, and 5 were plan-related. One of the deadlock bugs was given as an example during training on the tools, and, one of the data races was unreachable in the code that was eventually analyzed - thus leaving only 10 seeded errors.

Versions In our experiment, we simulated a normal development cycle where testing teams receive updated versions of the code and performed regression testing. Moreover, we also wanted to evaluate to which extent the set-up effort that advanced verification techniques sometimes require gets leveled-off during regression testing. For these reasons, we produced 3 versions of the code, where versions contained a number of common and a number of different errors. The code was also reorganized and changed at places between versions. Version 1 contained a total of 7 known bugs, out of which 2 were deadlocks, 1 data race, and 4 were plan-related. Version 2 contained a total of 7 known bugs, out of which 2 were deadlocks, 1 data race, and 4 were plan-related. Finally, version 3 contained a total of 7 known bugs, out of which 2 were deadlocks, 1 data race, and 4 were plan-related.

New bugs During the experiment, 18 new previously unknown bugs were detected, 3 of which were deadlocks, and 15 of which were plan-related. Although we did our best to understand these bugs and their causes, it is possible that some of them are different manifestations of the same actual bug.

4.2 Experiment Design

We used the traditional approach to experimentation where we assigned one group per treatment (V&V method) and the observers analyzed the results obtained by each group.

The questions that our study targets are difference questions that are based on comparing the performance of the various groups. To populate groups we need to determine the frame of our experiment, which, for our purposes, is system developers/engineers with a good understanding and training in computer science. However, due to lack of human resources, we could not construct a sample of subjects that would be large or diverse enough to allow for proper generalization of the results of our study.

Participants The participants to the study consist of 4 graduate students in Computer Science on summer internships at NASA Ames, and 4 NASA Ames computer scientists. The summer interns are working towards PhDs in the domain of formal methods. Some are very proficient with the tools used in the study, while others have only a superficial knowledge of these tools. However, even the latter are very knowledgeable in formal methods and have worked with similar tools. The permanent Ames employees hold Masters or PhDs in relevant fields of computer science, and have extensive background in software development.

Validity For reasons stated earlier in this section, our study does not claim external validity (possibility to generalize), but focuses on establishing internal validity. To achieve the latter, we tried to ensure the following characteristics:

1. Groups must be **equivalent**.

   Each of the four groups for our study was formed by two participants. We assigned to each approach/treatment the participants that were most proficient in the particular approach, based on their work experience and the focus of their PhDs. More specifically:

   Testing (TT): two NASA Ames software engineers with extensive development and testing background.

   Static Analysis (SA): one NASA Ames software engineer with extensive background on static analysis techniques and compilers, and one summer intern whose PhD is related to static analysis techniques for module verification.

   Runtime Analysis (RA): one NASA Ames software engineer with a PhD in monitoring in distributed systems, and one intern with extensive experience in building tools for model checking for Java programs and knowledge of runtime analysis algorithms.

   Model Checking (MC): two summer interns working on model checking, one of them involved in the development of some features of the model checking tool to be used.

   To bring all participants to a similar level of competence within the time frame of the experiment, we gave them 3 days of training on the tools they were to use and the code they were to analyze, as described in the training section (see below).

2. Groups must be **independent**.

3. All factors, except for the treatment, must be **constant** among the groups. This is to ensure that the differences could only be influenced by the treatments.

   Except for the SA group, all participants have been provided with exactly the same code, as well as a set of requirements that this code must satisfy. Specifically, in our context, these requirements consisted of the semantics of the plan language that the Executive interprets. The experiment time was exactly the same for all groups. As already mentioned, since the SA team worked on different code and concentrated on different types of bugs, their experiment results were not directly used for comparison with other tools. However, the SA experiment has still been valuable for obtaining some information about the benefits that the PolySpace tool can offer vs. the effort that its use requires in practice.
   Each participant had a powerful PC on their desk, with the exact same configuration. Finally, we made sure that during the experiment, groups would not receive hints or input from the observers/designers of the experiment accidentally. For this reason, personal communication had to be avoided. A mailing list was created through which all communication between the participants and the observers was performed. The mailing-list included all the

observers. This gave time to the observers to read the queries carefully, think before sending replies, and check the replies sent by others. Four experts in each technology were responsible for replying to queries of their respective groups.

Training The training was performed in four stages:

1. An introduction to the Executive functionality and architecture. That was performed during an interactive session by the developer of the software to all the participants in the experiment.
2. Code walk-through. One group consisted of the TT, RA and MC participants, since they would all work on the Java code, and the other one consisted of the SA participants that would work on C code. The walk-through was performed by the software engineers that translated the code.
3. Use of the tools and methodologies associated with the approaches/treatments. The participants got hands-on experience on running the tools on examples that were different from the experiment code. The four groups received separate training during this session.
4. Running the Rover and writing plans. The use of the tools was also demonstrated on one of the seeded bugs of the Rover code.

4.3 Experimental Set-Up

Working hours Each version of the code to be verified was sent to all participants at the same time through email, at the beginning of the session for the particular version. The participants had 16 hours to work on the version. The start and the end times of each sub-experiment were fixed. However, since each sub-experiment was two working days, and to accommodate the constraints on the daily schedule of the participants, they had the flexibility to schedule their 16 hours as fit. Experiments started at 1pm and ended 1pm, to allow for overnight runs of the PolySpace Verifier.

Reports Participants had to send a report to the mailing list after every hour of work. This report accounted for the way they used their time during this hour. The aim was for us to collect data about the relative set-up times of tools, tool running times, time required to understand/analyze the results obtained, etc.

Moreover, participants were asked to send two types of bug reports. The first, named preliminary bug report, was a report that had to be sent immediately after a potential bug was encountered, together with the information (output) provided by the tool, or the reason why this was considered a potential bug. The second, named final bug report, had to be sent after the participants had performed sufficient analysis of the bug, to be able to say with some confidence whether they considered the bug in the preliminary report spurious, or real. All teams had to provide at least the following information associated with each bug considered real: the input that resulted in the manifestation of the bug (a plan,

or a property, or both), whether the bug was reproducible, and the reason why the behavior of the system on the specific input was considered erroneous.

The intention for the bug reports was to collect data about types and percentages of bugs that were found by different tools, rates of finding defects, spurious errors reported, etc. If code fixes for a bug were available, they were provided as soon as the bug was confirmed real.

4.4 Debriefing

After the end of the experiment, all participants were asked about their experiences from the use of the tools. They also prepared written reports including such issues as the approach/methodology they followed, difficulties that they faced, and potential improvements for the tools.

5 Overview of Technologies Benchmarked

5.1 Static Analysis

The PolySpace Verifier is the first commercial tool that uses abstract interpretation to find runtime errors. Abstract interpretation is a static analysis technique that explores the source code of the program without executing it, and therefore no test cases are required. The tool focuses on identifying the following errors:

- Attempt to read a non-initialized variable
- Access conflicts for unprotected shared data in multi threaded applications
- Referencing through null, or out-of-bound pointers
- Out-of-bounds array access
- Illegal type conversion (long to short, float to integer)
- Invalid arithmetic operations (e.g. division by zero)
- Overflow / underflow of arithmetic operations
- Unreachable code

In the context of the V&V benchmarking study the static analyzer, PolySpace Verifier, has been solely used as an abstract debugger for finding runtime errors. The output of the tool consists of a color-coded version of the program source code. Each potentially dangerous operation can be flagged by four colors depending on how the associated safety conditions have been handled by the tool:

- Red: the operation causes a runtime error for all execution paths that lead to that operation.
- Green: the operation is safe, a runtime error can never occur at that point.
- Orange: this color covers two cases:
  - An execution path to this operation causes a runtime error, whereas another execution path does not.
  - the tool has not been able to draw any conclusion on this operation.
- Grey: the operation is unreachable (dead code).

The verification process using the static analyzer starts with a piece of software and no a priori insight on the structure of the code. This is the most common situation: it amounts to using the static analyzer during the validation process of a fully developed software. There are three main stages in putting the static analyzer to work:

- Isolate the part of the code that will be analyzed and make it accepted by the front-end. This is the compilation stage.
- Run the analyzer and correct all certain runtime errors that are detected by the tool. This is the debugging stage.
- Run the analyzer and modify the code in order to improve the precision and remove false alarms. This is the refinement stage.

Each of these three stages is highly iterative: the user runs the analyzer and modifies the command-line parameters and/or the code until no more error messages are issued. If this process had to be compared to software development, the first stage would correspond to compiling the program and eliminating all syntax and type errors, the second stage to debugging the program and the third stage to optimizing the algorithms for achieving better efficiency.

The training of the participants consisted of verifying the cipher algorithm RC4 from the ssh source distribution with PolySpace Verifier. Several runtime errors have been artificially inserted into this algorithm.

Static Analysis Experiment Setup At the time of the experiment, PolySpace Verifier worked only on Ada and C programs. Therefore, the participants were given a C version of the rover executive code. The translation from C++ to C was done automatically by a commercial compiler frontend (from Edison Design Group). However, this led to a code explosion (due to the exhaustive nature of such a translation) and we therefore modified the resulting code to stub out some low level calls before giving it to the participants. We also gave the stubs (all grouped in one file for easy access) to the participants so that they could see how to write stubs within the PolySpace analysis context. They were free to refine those stubs and write additional ones.

The participants used the PolySpace Verifier on two machines with the same configuration (DELL 530 PC running Linux). Unfortunately, in the Linux configuration we had we could not use the full power of PolySpace Verifier (i.e., the tool ran out of memory in its highest precision level). Therefore, the participants were restricted to using the quick analysis mode and the low precision level of the tool. We made sure that the seeded bugs could be found with these restrictions.

Finally, each version was delivered to the participants with a short description of what changes were made to the code. The intent is to give the same type of information a CVS repository would give in a typical NASA development environment.

5.2 Model Checking

The Java PathFinder (JPF) model checker [15, 16] is an explicit-state model checker that analyzes Java bytecode classfiles directly for deadlocks, assertions violations and general linear-time temporal logic (LTL) properties. JPF is built around a special-purpose Java Virtual Machine (JVM) that allows all Java programs to be analyzed. Furthermore there are no limitations on the number of threads, objects or classes in the program to be analyzed - in theory it can therefore find errors in infinite-state programs, but it cannot determine whether such programs are correct. The level of atomicity can be set to either one bytecode instruction or one line of code, with the latter being the default setting.

JPF supports depth-first, breadth-first as well as heuristic search strategies such as best-first and A*. The heuristic search strategies can be guided by built-in heuristics for finding concurrency errors, achieving structural coverage of the code, minimize nondeterministic choice or the user can define their own heuristics [11]. A novel feature of the heuristic search is that it also allows the user to specify the size of the queue of states to be analyzed next - this allows a lossy search that focuses in on errors (somewhat like a beam-search) to deal with cases where the state-explosion is too severe.

In order for the user to interact with the model checker a special class called `Verify` has been introduced: the user can call special methods of this class that gets intercepted during model checking. The following methods are most commonly used:

random(n) - Returns a nondeterministic value between 0 and n - used when modeling a nondeterministic environment.

randomBool() - Returns true or false nondeterministically - often used for data abstraction.

assertTrue(condition) - Allows local assertions to be checked - if the condition fails an exception is thrown. Note, since the introduction of assertion checking within Java (since Java 1.4), JPF also traps these methods.

beginAtomic(), endATomic() - Respectively indicates the start and end of a block of code that the model checker should treat as one atomic statement.

ignoreIf(condition) - Allows the user to truncate the analysis of a specific behavior if a condition becomes true.

boring(condition), interesting(condition) - These directives are used only during heuristic search and respectively decreases and increases the heuristic values for a specific state if the condition holds.

JPF cannot deal with *native* code - that is code that is not written in Java, but is called from within the Java program. In these cases the user need to specify the behavior of the native code and JPF will treat it as an atomic statement, which means errors in this code might be missed. In this study no native code was used, since we were not analyzing the actual communications between the executive and the rover hardware.

JPF Experiment Setup Abstraction is typically a very important technique required for the success of model checking, since it combats the state-explosion problem inherent to the analysis of complex software systems. We therefore provided a framework in which the participants could add abstractions in a non-intrusive fashion to the code. Besides the infrastructure for adding abstractions we also gave the participants the so-called "point" abstraction for all time-related data - i.e. all operations on time become nondeterministic choices. We anticipated that they would refine this abstraction to suit their needs. We also provided the Rover code with `beginAtomic()` and `endTomic()` calls to show where interleavings are unnecessary.

Lastly, we also provided the participants with a number of simple Rover plans as examples, as well as a so-called "universal" planner that can generate all possible plan structures up to a specific size limit in a nondeterministic fashion. The idea behind the universal planner is common in model checking, where a system is often analyzed within the most general environment it can operate in.

Since the participants were both familiar with model checking and more specifically had used JPF before, the instructions on how to use the tool were minimal. Essentially, we requested that the participants "only" run the model checker and not run the code directly to determine where possible errors might exist - we wanted them to use the model checker to discover errors rather than just localize errors first observed during testing/simulation. We also requested them to determine whether errors reported were spurious or real - this meant that they needed to understand the code quite well.

5.3 Runtime Monitoring and Analysis

Two tools were used: DBRover, which checks execution traces against temporal logic requirements, and JPaX, which checks execution traces for deadlock and data race potentials.

DBRover Run-time monitoring is a method whereby the correctness of a single execution is validated against a set of formally stated requirements. DBRover [9] uses Metric Temporal Logic (MTL), which essentially is Linear Temporal Logic (LTL) extended with real-time constraints. LTL extends propositional logic with four future time operators: Until, Eventually, Always, and Next, and four dual past-time operators. LTL has the property of being capable of describing many interesting properties of reactive systems. MTL extends LTL in that every temporal operator can be augmented with a real-time constraint. Hence, for example, '$x > 0$ Until$_{<5}$ $y > 0$' means '$x > 0$' must be true until a future time, at most 5 real-time units in the future, where '$y > 0$' must hold.

The DBRover consists of a GUI for editing temporal formulae, a simulator for testing the semantics of the formulae, and a remote execution/validation engine. The DBRover builds and executes temporal rules for a target program or application. In run-time, the DBRover listens for messages from the target application and evaluates corresponding temporal formulae. Hence, in the example above,

the DBRover will listen for Boolean messages pertaining to the run-time value of the '$x > 0$' and '$y > 0$' propositions, and then evaluates the corresponding MTL formula. Monitoring is performed on-line, namely, the DBRover operates in tandem with the target program, and re-evaluates formulae every cycle. The DBRover uses an underlying algorithm that does not store a history trace of the data it receives; it can therefore monitor very long and potentially never ending target applications with no performance degradation over time.

In order to drive the DBRover temporal logic monitoring engine, the program to be verified must be instrumented to emit events to it. When a set of formulae have been created in DBRover, a code snippet can automatically be generated, which the tester then shall insert in the program to be monitored at places where the formula should be evaluated.

JPaX The Java PathExplorer (JPaX) [12] can detect deadlock and data race potentials in Java programs by analyzing a single execution trace. A characteristic of the tool is that it can detect such flaws even though these do not occur in the actual execution trace examined. The program to be analyzed is automatically instrumented to emit events representing acquisitions and releases of locks as well as read and write accesses to variables. The lock acquisition/release events are used by the deadlock analysis as well as by the data race analysis. The variable read/write events are used by the data race analysis. For each kind of analysis, the tool builds an internal data structure representing an abstraction of the execution trace. The abstraction will violate a certain well-formedness predicate if there is an allowed permutation of the original execution trace that causes an error (a deadlock or a data race).

The deadlock algorithm detects cycle deadlocks (where several threads acquire locks in a cyclic manner, such as for example when a thread T_1 acquires a lock L_1 and then a lock L_2, and another thread T_2 acquires these locks in the opposite order. The data race detection uses the Eraser algorithm [14], adapted to Java. A data race occurs if several threads access a variable at the same time, and at least one of the threads writes to the variable. Normally variables must be protected with locks to avoid this. JPaX checks that for each variable that is shared between several threads, there is a common lock that all the threads acquire before they access the variable. If not, a violation is reported.

Runtime Experiment Setup Instructions for the participants were separated into general instructions on how to use DBRover and JPaX on any application, and specific instructions on how to use the tools on the Executive. The general instructions for using JPaX were minimal since the tool requires no specifications to be written and since instrumentation is performed automatically by the tool. Since DBRover requires specification writing, and manual code-snippet generation and insertion, the general instructions for this tool were more elaborate. However, the learning curve did not seem to cause any problems and could be done in a few hours.

The participants furthermore were instructed on how to write temporal formulae in DBRover for plans, representing the plan semantics. In general terms, for each action node in the plan the testers were recommended to write a set of formulae that would test that it was executed correctly. Note that although the other groups were also given instructions on the plan semantics and its temporal properties, the RA group received specific instructions.

The RA group was asked to use the DBRover as much as possible, and not to only rely on the printed information from the program executions.

6 Qualitative Results

We will first describe the information we got from debriefing the participants after the experiment - augmented with our own comments in *italics* - followed by some final thoughts on how each technology was used.

6.1 Testing

Debriefing The initial focus was on finding concurrency related errors, since the assumption was made that this will form a good basis of comparison with model checking and runtime analysis. *Note that even with this focus, testing found less seeded concurrency errors than the other teams.* Due to the time-limitations of the experiment and the tight-coupling of the components in the code, both unit testing and integration testing phases were skipped in favor of system level testing. Furthermore, due to the input-output nature of the system (plans as input and rover executive actions being observable) the focus was on black-box testing. The system design indicated the likelihood of race conditions and inconsistent system states due to the use of exceptions for control flow. *Although this was a very accurate observation, the seeded errors only contained one race error. On the other-hand the use of exceptions contributed, in one way or the other, to one seeded error and most of the unseeded errors in the code.*

Since, the system consists of multiple threads, two different operating systems (Linux and Windows) and two different Java Virtual Machines (1.4 and 1.3.1) were used during the testing to see if the native scheduling will show different behaviors that will point to concurrency errors. In addition, the only other tool support was the Java debugger environment to allow a dump of the thread's call stacks when a deadlock was perceived - this was used to eliminate possibly spurious deadlocks. *Note, the testing team did find spurious deadlocks, but this was due to the original rover accepting multiple plans whereas the one in the experiment only receiving one plan - hence it could be waiting for the next plan, which was not considered to be a deadlock, although, from a black-box point of view, it looked like one.* The code was instrumented to allow rover actions to be better observable, but care was taken to do this without modifying the system timing. The plan parser was extended to allow the expected output to be printed when the plan was parsed - this expected output could then be compared to the actual output to find discrepancies.

The test strategy was to use the grammar of the plans to create test-cases that may cause problems, e.g. having an *end-time* that is before a *start-time* in a plan, missing parts of a plan, etc. *Note, that although this was the strategy the testing team didn't find the only two seeded plan errors that were due to malformed plans. The explanation for this is that the testing was not systematic and therefore not all the combinations of "bad" plans ere tried.* During the last phase of the testing, active scheduler variants (as apposed to passive variations from the different execution environments) were used to try and find concurrency errors, these included, changing thread priorities, adding yield and sleep statements, etc. *These changes allowed two unknown deadlocks to be uncovered during the testing of the last version of the code.*

The plans that produced errors in the code were kept in a regression suite for use in later versions.

Observations The testing group took a black box approach to finding bugs. However they did modify the code to provide more elaborate diagnostic information via print statements. The participants recognized from the tutorial that concurrency was extensively used and may be a major source of bugs. Thus they focused their efforts on formulating test plans that might demonstrate concurrency bugs. They ran their tests on multiple platforms to identify whether behavior depended upon the scheduling algorithm. For some tests they modified task priorities. They also wrote plans in which the time point in temporal constraints were temporally close, to analyze boundary cases.

The testing group maintained a regression suite that they ran on each new version, and so quickly identified bugs that were seeded in multiple versions. Very limited tool support was used.

6.2 Runtime Analysis

Debriefing The JPAX tool was particularly attractive since it didn't require any user input and therefore it was always tried first to find deadlocks and data races. More complex plans were then created to flush out other errors and JPAX was ran periodically on these new plans. Although it is possible in general, JPAX didn't produce any spurious errors during the experiment. *This was a particularly interesting result, and points to the usefulness of the approach to finding data races and deadlocks. JPAX did miss one of the seeded deadlocks, but this was because it cannot find so-called "wait-notify" deadlocks, only deadlocks due to cycles in the lock graph.* No time was spent coming up with ways to make the errors JPAX reported appear during execution; code inspection was deemed sufficient to determine if the errors were real. Near the end of the experiment they also constructed plans to maximize the execution of locking instructions to find more concurrency errors. *This attempt seemed to have been unsuccessful since no new concurrency errors were found doing this.*

DBRover had the overhead of first requiring one to determine a temporal logic formula that the input plan will satisfy and then to input this formula to

the tool. For simple plans it turned out that it was easier to just look at the output of the run to determine if a formula was satisfied. *This approach will not work for complex plans.* The runtime team concluded that if they had a tool that could produce the formulas for each plan automatically they would have used DBRover more often. *We considered creating such a tool before the experiment, but didn't do it - we have since created it and it is now in use for analyzing the current version of the Rover.*

Once an anomalous behavior was detected they analyzed it in more detail, by trying to understand what the program is doing, to determine if it was a real error or maybe a misinterpretation of the specification. Also, they considered whether the current error might be due to a previous error that was not corrected (yet). *The runtime team spent much more time analyzing the code than the testing team did and thus produced error reports that could more easily be classified by the experiment observers. Although we tried, it was often impossible for the observers to produce a fix for an unknown error - we had fixes for all the known errors.* Time was also spent trying to fix errors that were found, for which no fix was provided. *Although this was a possible waste of time, new errors were discovered this way that was masked by the original.*

The collaboration within the team consisted of one person coming up with plans and the other mostly trying to locate an error after observing an anomalous behavior. *The person looking at the code was particularly good at finding errors through inspection and acquired an understanding of the code that rivaled that of the developer.* The team members also spent a large amount of time discussing the different errors they have found.

Observations The JPaX tool appeared to be used in the way it was intended. When receiving a new version of the Executive, they usually started out applying JPaX to a few simple test cases, and thereby easily discovered deadlocks and data races this way. No advance preparation was required in addition to writing the test cases. From time to time, in between applications of DBRover, they further applied JPaX on some of the more complex plans, to see if it could detect in them other possible deadlock and race conditions. Close to the end of the time available they also tried to construct plans to maximize the coverage of the locking instructions reported by the deadlock analysis, or determined by examining the code that such a locking instructions was not reachable. The use of JPaX required to check that a deadlock or race condition potential reported was actually able to lead to a real error. For these kinds of errors the code was analyzed to understand if the error could effectively occur or not. They did not try to make the errors happen in the real code. The testers stated that the tool allowed them to "easily detect deadlock and race Conditions". Stating that "We could run it and it would come back with suspected deadlock and race condition problems which would have been very hard and time-consuming to find with the traditional testing".

The DBRover was not fully used as requested. For each plan they had to write temporal formulae, generate instrumentation and insert the instrumentation in

the Rover code. It took some time to get familiar with the notation and the tool. Furthermore, they perceived the writing of temporal formulae as "consistent overhead", while for most of the simple plans just looking at the output was effective to determine if the plan behaved correctly or not. Hence, their general working mode was to write test cases, and in some cases just run the program while examining the output, and in some cases also writing temporal formulae using DBRover. In some cases DBRover detected errors that were not caught by analyzing printed results.

One of the testers was specifically responsible for applying JPaX, while both RA participants applied DBRover. The tester that applied JPaX also wrote most of the test cases. The other tester would mostly try to locate and often fix the bugs in the code, and hence was the one that eventually became very familiar with the code and found bugs by simple code inspection.

JPaX found all the cyclic deadlocks and a data race condition that were seeded in the program quite easily. Since these are the only kinds of bugs the tool is designed to catch, one may say that its application was a success. Furthermore, it gave no false positives. The participants were quite happy with the tool. Of course, the ease of use has been achieved by restricting the kind of errors that can be found to deadlocks and data races. DBRover, on the other hand, allows much more general properties to be analyzed, and hence also is more complicated to use. In general, however, DBRover seemed to be less useful for the testers (their own formulation), and the main reason for this was that it took time to write the temporal formulae. Although this can be said to be a negative result, three observations can be made for a positive evaluation of DBRover for testing real systems in a realistic development environment:

- the plans used were quite small and one could manually check the outputs, but in general plans will be very large and an automated tool to check the properties will be required
- if an automated translation from plans to temporal formulae were available the tool would have been used much more
- the tool is specifically useful in regression testing where large numbers of tests are run, which were not the case here

6.3 Model Checking

Debriefing The team started running the model checker on the supplied code straight-away looking for deadlocks, i.e using the supplied sample plans. They modified the search parameters rather than modify the input plans at first. *This is in contrast to the more traditional testing approach, for example used by the testing and runtime teams.* When errors no longer appeared using this approach they started using the *universal* planner - that generated all plans up to a specific size. The universal planner was used only sparingly. *We assume this is due to the fact that they wanted control over the exact plans that was used in a model checking run.*

They found an error immediately, but then spent hours to determine that the error was spurious due to the abstraction of time. *It turned out that this*

error was in fact a real error, but the specific version found was spurious. One of the team members then decided to create a more realistic version of a time abstraction that would not allow over-approximations of time behavior. The team didn't want to spent all their time figuring out if an error was spurious, since the code was to complicated to understand completely in the time allowed. *This new abstraction of time was an under-approximation of timing behavior and could lead to missing some errors, although it did allow all the seeded plan errors that could be found by model checking to be found.*

Both team members cited the difficulty in understanding what the program was doing as their major difficulty in model checking the code. *This mostly factored into the abstraction, and specifically determining whether errors after abstraction were spurious or not.* A large amount of time was also spent tuning the model checker's search options to cover parts of the state-space that wasn't covered before - e.g. to cover "deep" paths without cutting too many behaviors during the early parts of the search. *The fact that the team members knew the model checker's search characteristics very well might have hampered them more than it helped, since it made them spend more time on tuning the model checker than what was needed.* Lastly, they didn't use any temporal properties to check program behavior, rather encoding plan related properties as assertions. Writing these assertions was cited as the third most time consuming activity.

Two problems in the provided code hampered the model checking team: firstly, there were some mistakes in the atomic regions provided that would mask some errors, and secondly, a mistake in the universal planner meant it was not complete and certain plans were not generated. *The latter problem made the model checking team miss a plan related error until the last version when the problem was discovered and fixed. The atomic problem was discovered early on, but the participants spent a fair amount of the first version's time to fix it.*

Observations After finding the first spurious error the participants abandoned the point abstraction of time in favor of a more realistic (under-approximation) of time. The main reason for this was the complexity involved in understanding the Rover code to determine if a bug is spurious or not. This new abstraction of time, had time starting at 0 and on each lookup of time, time either remained the same or incremented a random amount up to a certain maximum time at which point time stood still.

The participants only used the universal planner sparingly: first they would try and uncover bugs with the concrete plans provided (and small modifications thereof) and only after exhausting most of their ideas would they turn to the universal planner to discover errors.

During the running of the JPF tool most time was spent in trying different search strategies to reach different parts of the state-space. These mostly included setting different parameters for the heuristic facilities provided by the tool.

The participants used the point abstraction of time and the universal plan less than we anticipated. We believe this to be due to the fact that we knew

the code much better than they did and could find all the known bugs with the abstracted code, whereas they found spurious behaviors early on and decided rather to abandon the abstracted code for something more realistic (although at the risk of missing some errors). This was a good decision on their behalf since they clearly spent less time on determining whether bugs were spurious than what we anticipated, but still found almost all the known errors.

6.4 Static Analysis

Debriefing The participants had problems optimizing the analysis given that they had two computers at their disposal. Long analysis runs generally took eight hours, and up to, in at least one case, 23 hours. This prevented them from running concurrent analyzes on the same artifact because the concurrent analyzes destroy each other's result as they go along. *Actually, there is a simple solution to this problem: duplicate the artifact on two different physical locations so that the analyzes can be run in parallel.*

Their main strategy was as follows:

1. Run PolySpace
2. Analyze red check (definite errors)
3. Analyze gray checks (dead code, possibly due to errors)
4. Analyze orange checks (possible errors)

This already reveals a problem; dead code is indicative of errors only if it follows a definite error. Therefore, manually analyzing dead code will not help in finding more errors. It will just delay the analysis of the possible errors. This aspect somehow eluded the participants.

Not surprisingly, the participants encountered difficulties analyzing possible errors. There was very little feedback from the tool, and no way to let the tool know which errors were spurious. Many possible errors were caused by the imprecision of the analysis and the imprecision of stubs. They therefore use the following strategy:

1. Read the code to look for errors.
2. Iterate through the possible errors item by item.
3. Simplify the code and stubs so that PolySpace could do a better analysis.

This points actually illustrates one of the biggest difficulty when using static analysis tools. In dealing with possible errors, it helps a great deal if one knows the algorithms used by the analysis. It allows the user to make intelligent decisions about improving precision (other than raising the precision level, which was not an option in this case). For example, knowing the source of approximations in the analysis algorithms may help you avoid wasting time on trying to lift uncertainty on some possible errors.

The participants felt that reading the code violated the spirit of the experiment. They however did some of it *and actually found errors that could not have been found by the tool, e.g., correctness issues* Overall, they found that reading code was difficult because of the mature of the code that they analyzed.

Iterating over the errors was also difficult, because it was not supported by PolySpace. They would have liked support for:

1. Recording which warnings were spurious, therefore oranges were reanalyzed.
2. Annotations that record variables that are defined/properly aligned.
3. The types of the variables and sub-expressions involved in the problem. In more than one case, the bug involved knowing that a variable or subexpression was signed versus unsigned. This led to false dismissals of real problems.
4. Example paths/data values to demonstrate the problem.

Overall, they favor the option of simplifying the code (*which is a really bad idea since you are not analyzing the real code anymore*), and working on the stubs (*this also led them to get confused and start reporting stub errors instead of code errors*).

Observations The users of the static analyzer spent a lot of time trying to remove orange alerts by adding assertions into the code (refinement). Their efforts were mostly vain because the assertions they inserted could not be taken into account precisely by the tool. Performing refinement efficiently requires a very good knowledge of the abstract interpretation used by PolySpace Verifier. This can be interpreted as a limitation of the tool.

During the reviewing process of orange alerts, some obvious runtime errors were completely overlooked whereas numerous spurious bugs were reported. There are two interpretations:

- The domain of applicability of each operation flagged as orange should be carefully checked in every possible execution context. The overwhelming amount of orange operations makes this process very difficult to apply rigorously.
- Most importantly, the notion of runtime error was loosely interpreted: a runtime error is the violation of a safety condition which is clearly defined by the programming language (in this case C) specification for each operation. Several (spurious) bugs that were not related to the violation of a safety condition have been reported. Similarly, the users of the tool spent a substantial amount of their time trying to increase the number of execution paths that could be analyzed by the tool. This resulted into unduly modifying the original program or trying to make dead code reachable.
  On the positive side, while sorting the orange alerts, the participants read the code with careful attention. It resulted in finding four correctness errors (e.g., erroneous conditions in conditional statements and loss of precision problems), which were unseeded. Correctness errors are typically not caught by a tool like PolySpace Verifier. So, as it has been observed with other formal methods, using the tool brought a new, sharpest, sense of attention to reading the code, and it resulted in finding errors. In all cases, these errors were found around the 9th or 10th hour of running the experiment. This reflects the fact that it takes a long time to sort through the orange alerts, which corresponds to the period of increased attention.

Similarly, as expected, we noticed that errors that are easy to understand such as attempts to access non-initialized variables were caught much faster than others such as overflow problems. In the case of overflows, it takes a while to be convinced that it is a real problem. Most of the time, these errors reflect type clashing that results in theoretical overflow rather than realistic ones (in the sense that the program might never execute long enough to reach the overflow).

7 Quantitative Results

In this section we will give some qualitative results from the study. Due to the complexity of setting up an experiment like this it is impossible to draw statistically valid conclusions. We'll nonetheless give our interpretation of the results in the hope that it'll serve as a source of reference for similar studies.

The static analysis portion of the results were obtained on a different version of the code than the other techniques, and furthermore, static analysis looked for a different set of errors since the errors from the CVS log were not suitable for static analysis. For these reasons we only concentrate on the results from model checking, runtime analysis and testing in this section.

Note that a vast amount of data was collected during this study, but here we only highlight some selected pieces.In the future we hope to make all the data available for the research community to interpret.

7.1 Finding Seeded Bugs

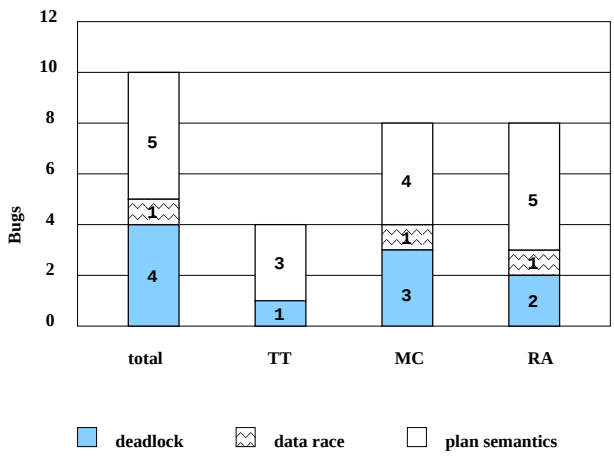

Fig. 3. Seeded Bugs found by each technique.

Figure 3 shows how the different techniques performed when looking for the seeded errors. Although we started with 12 errors, one was used for illustration during the training phase and one of the race errors was unreachable in the version of the Rover the participants worked on. The advanced tools - model checking and runtime analysis - performed better on the concurrency related errors. This was a result we anticipated beforehand, and is also common folklore in the verification community. The fact that testing performed worse than model checking in all categories was somewhat of a surprise. Runtime analysis outperforming testing is not surprising, since runtime analysis can be seen as an advanced form of testing.

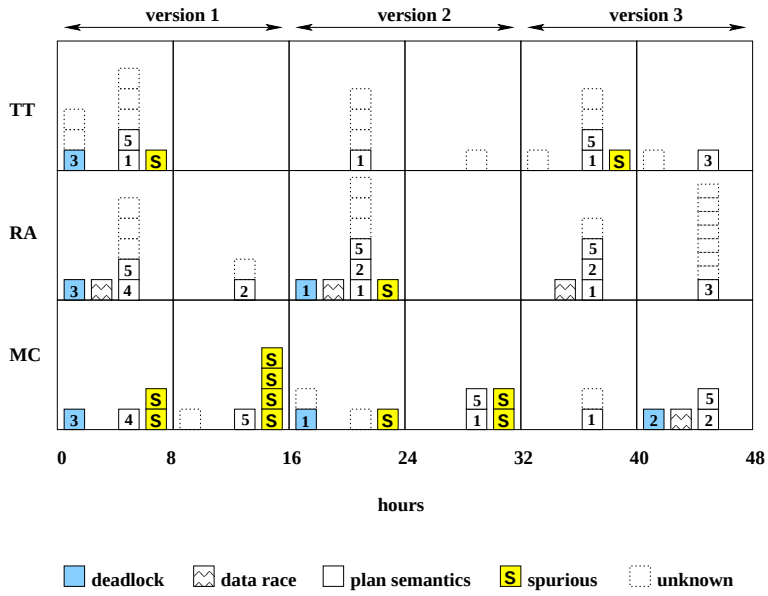

Fig. 4. Time-line for seeded and unknown (dotted boxes) bug discovery across Versions.

In Figure 4 we can see the breakdown per day of the bugs found on a 48 hour (6 day - 3 2-day sessions) time-line. For the known bugs one can also see when each team found the bug, since the bugs are numbered. The spurious errors as well as the unknown bugs are also shown. As pointed out before, we focus on the known bugs, since it was often impossible to tell if a unknown bug is not a (different) manifestation of another unknown bug (or maybe even a known bug). Note that Figure 4 doesn't show the exact times when a bug is found, rather all the bugs discovered are grouped per the day it was found.

The first observation is that for model checking spurious errors decrease across the three versions. This is due to the participants becoming more familiar with the code and also developing better abstractions as the experiment contin-

ued. This seems to indicate that the start-up cost to establish a framework for efficient model checking can be amortized effectively for verification of long-term systems/projects. For runtime analysis spurious errors were almost nonexistent - the one reported was due to a misunderstanding of the Executive behavior - the two spurious errors for testing also fell into this category.

For runtime analysis that the deadlock and race checking were ran near the start of each experiment - within the first day of each version there is a race and a deadlock found (except in the 3rd version where the seeded deadlock was not of the kind that the runtime analysis could detect).

Deadlock 3 is very simple and was found early on by all three methods. Deadlock 1 on the other hand is more subtle, but both model checking and runtime analysis found it early in version 2 (the only version in which that error was seeded), end testing never found it. Deadlock 2 was the most complex of the deadlocks seeded and it was only present in the last version. Deadlock 2 was only found by model checking; note that the runtime analysis tool could not detect this kind of deadlock. There was only one race violation bug in the code and it was seeded in all three versions. Runtime analysis is geared to finding these efficiently and found it in each version. The model checking team on the other-hand only attempted to find race errors later on in the experiment and therefore only found the bug in the last version. A race violation is almost impossible to find with black-box testing and hence testing didn't find this error.

Testing did well on the seeded plan errors, but although the focus was on trying to create potentially "problematic" plans (see section 6.1 for the debriefing of the testing team) they didn't find plan bugs 2 and 4 that were caused by malformed plans. Runtime analysis performed best on plan errors. Interestingly though, many of the plan errors that they discovered were by using very similar techniques to the testing team - i.e. printing observable Rover executions and comparing it with expected results. The DBRover tool that could have automated this process for them was not used as extensively as expected due to the constant overhead of adding temporal formulae for the tool to check when it was easier to just look at the printing - see section 6.2 for the debriefing of the testing team. Model checking didn't find many plan errors early in the experiment since they focused on finding errors by tuning the model checker's search strategy while keeping the plan constant - with the hope of finding concurrency errors.

Testing and runtime analysis found a large number of previously unknown errors, but interestingly many of these errors were *only* found by the respective tools, i.e. the ones testing found was not found by runtime analysis and vice versa. We conjecture that this is symptomatic of the fact that the observers could not adequately determine the cause of each anomaly reported by these two teams. The reason of course being that for both runtime analysis and testing the diagnosis of each error is hard since, unlike with model checking, there is no trace that shows the error happening. Interestingly, the only two unknown errors found by model checking was also found by the other tools (testing found one and runtime analysis found one).

7.2 Tool Usage

Figures 5, 6 and 7 shows how the participants spent their time during the analysis of the three different versions of the code. The information for these figures were constructed from the hourly reports that the participants provided during the experiment. The different categories were defined roughly as follows:

Delays: This includes waiting for hardware or tool failures to be resolved, as well as time spent waiting for feedback on questions to the "developers".

Setup/Preparation: This category had a different meaning to each group. For example for model checking this included time to create abstractions, writing properties and test-cases, whereas for runtime analysis this mostly included creating properties and test-cases, and for testing, the activity of creating test cases and instrumenting the code (for example, adding print statements).

Running the rover: Not really relevant for model checking, but essential for runtime analysis and testing.

Running the tool: Testing didn't really use any tools, but model checking and runtime analysis were tool-based.

Result Analysis: This is the time spent determining what the cause of an error is. For model checking and to a lesser extent runtime analysis this involves determining whether an error is spurious or not.

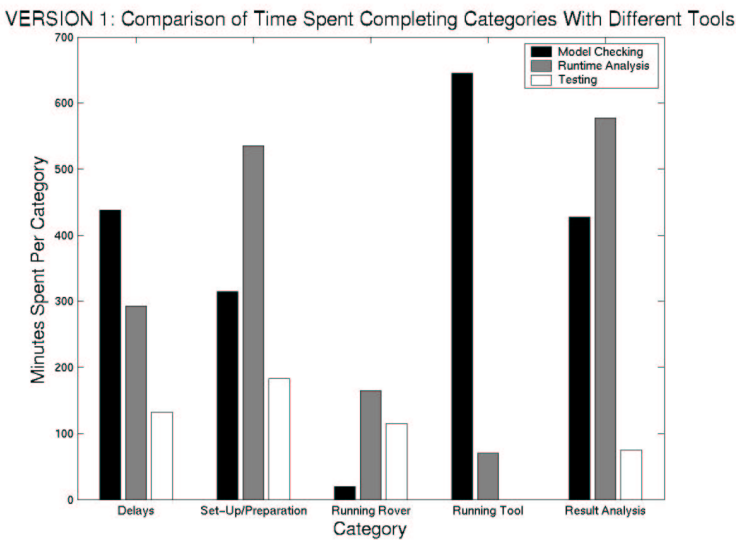

Fig. 5. Time usage during version 1.

The most notable observation is that the model checking participants used the tool a great deal - essentially running the tool at all times while doing other activities. There is less time spent running the model checker in the first version, obviously due to getting to grips with the problem, and then in the other versions almost 100% tool usage is reported. This is due to the fact that the model checker is quite slow on such a large example.

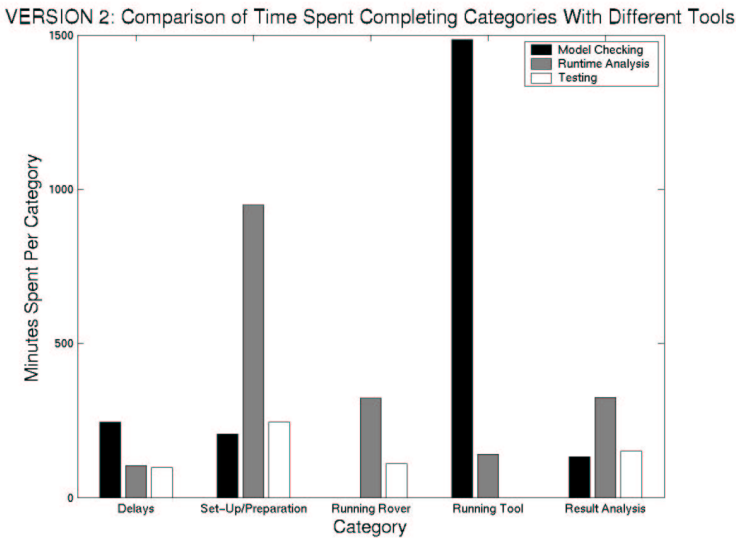

Fig. 6. Time usage during version 2.

Another observation for the model checking group is that they spent more time on setup in the first version than the next two. This also confirms the suspicion that when using a complex tool such as a model checker, it takes some time to setup the ideal working environment before any meaningful work can be done. A similar pattern is seen for runtime analysis - again as to be expected.

The testing group didn't spend much time analyzing the errors they found. This put much more of a burden on the controllers of the experiment to determine which of the known bugs, if any, they found. The other two groups spent quite large amounts of time on determining the cause of errors in the first version, but far less on the later two versions. We assume this is due to having to understand the code in the beginning, whereas later they could go through error-traces more quickly.

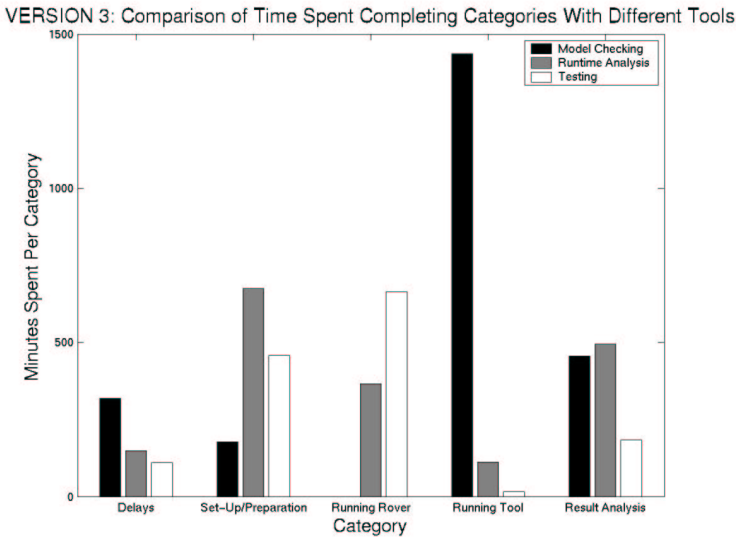

Fig. 7. Time usage during version 3.

8 Conclusions

We described an experimental study to determine the relative strengths of three advanced verification and validation technologies, namely, static analysis, model checking and runtime analysis, compared to traditional testing for finding errors in a representative example of Mars flight software. A conservative estimate on the amount of effort in setting up and running the experiment, as well as interpreting the results, is one man-year. An anecdotal observation is that this experiment inspired a rewrite of the Rover Executive.

The most important conclusion we drew from this study is that one should conduct such a complex study in the most controlled fashion possible. Although for the most part we achieved this goal, one area where we didn't, was in understanding the code sufficiently. This meant that in the end there was a large number of previously unknown defects reported that we could not adequately classify - although we knew they were not one of the seeded errors we couldn't determine if they were all unique etc. We didn't expect any unknown errors, and hence we have missed the opportunity to also conclude information about these errors. We decided to exclude them completely from making any judgements about the relevant merits of the techniques.

A consequence of the static analyzer not working on the same notation as the other tools was that we could not make any comparisons between it and the other techniques. Although we believe the information obtained from the study of the PolySpace Verifier turned out to be very useful and has guided our

own research in a significant fashion. The most notable conclusion we drew here was that for complex static analysis, such as done by PolySpace, the tool users require knowledge about the underlying algorithms to adequately use the tool.

With respect to the analysis of the Java version of the Rover the following general observations were made:

- The advanced tools performed very well on concurrency related errors, where traditional testing often performs worst - as it did here.
- Runtime analysis is a light-weight technique that produces very few spurious errors, and should always be one of the first techniques to be used on new code.
- Model checking requires some initial start-up costs, for example in coming up with suitable abstractions, but once this is done it can be reused (hence amortizing this cost over the rest of the verification phase).
- Using abstractions during model checking forces a very good understanding of the code to determine whether errors are spurious or not.
- A major weakness of black-box testing is that it doesn't typically provide enough information to diagnose the cause of an error. Runtime monitoring also suffers from this problem, but to a lesser extend, since the monitoring of the events allows for a partial trace to the error to be observable. Model checking gives a precise trace, but it might be spurious if abstractions were used.

One of the goals of the experiment was to determine synergies between the different techniques for finding errors in flight software. To this end we have developed a framework for analyzing the rover executive inspired by our observations of the experiment. The framework combines model checking and runtime analysis techniques in a novel way [3].

It was clear that runtime analysis and monitoring were very successful in uncovering the seeded errors, but since both these techniques require the code to be executed, they are only as effective as the test-inputs they are being executed with. In other words one requires good coverage of the input space for this type of analysis to uncover errors - note that in the experiment testing missed two errors due to malformed plans although that was their focus, simply because they were not exhaustive in creating such bad plans. Model checking on the other hand is good at systematically analysis, for example, covering all input plans up to a specific size for the rover - as was done with the "Universal" planner in the experiment. Lastly, the runtime team didn't use the monitoring facility of DBRover as much as we thought they would, due to the effort involved in writing the temporal formulae to be monitored for each plan. Our framework therefore combined the model checker's ability to create all plan structures (up to a specific size) and temporal formulas that each such plan should satisfy with a temporal monitoring facility to check that each plan is executed correctly.

Acknowledgement We would like to express our gratitude to the participants of the experiment, without whom this experiment would never have been possible: Jamie Cobleigh, Alex Groce, Oksana Tkachuk, Owen O'Malley, Flavio Lerda, Masoud Mansouri-Samani, Peter Mehlitz and Phil Oh.

References

1. J.-R. Abrial, E. Börger, and H. Langmaack. Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. In *LNCS*, volume 1165. Springer-Verlag, October 1996.
2. C. Artho. Finding Faults in Multi-Threaded Programs. Master Thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, October 2001.
3. C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, and W. Visser. Experiments with Test Case Generation and Runtime Analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines (ASM'03)*, Lecture Notes in Computer Science, pages 87–107. Springer, March 2003.
4. G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Pasareanu, and S. F. Siegel. Comparing Finite-State Verification Techniques for Concurrent Software. Technical Report UM-CS-1999-069, Department of Computer Science, University of Massachusetts at Amherst, USA, 1999.
5. B. Boehm and D. Port. Defect and fault seeding in dependability benchmarking. In *Proc. of the DSN Workshop on Dependability Benchmarking*, June 2002.
6. A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. An Empirical Comparison of Static Concurrency Analysis Techniques. TR 96-84, Department of Computer Science, University of Massachusetts, 1997.
7. B. P. Collins and C. J. Nix. The Use of Software Engineering, Including the Z Notation, in the Development of CICS. *Quality Assurance*, 14(2):103–110, Sept. 1988.
8. J. C. Corbett. Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Trans. Softw. Eng.*, 22(3):161–179, Mar. 1996.
9. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
10. S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Application and Experimental Evaluation of State Space Reduction Methods for Deadlock Analysis in Ada. *ACM Trans. Softw. Eng. Meth.*, 3(4):340–380, Oct. 1994.
11. A. Groce and W. Visser. Model Checking Java Programs using Structural Heuristics. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, July 2002.
12. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
13. PolySpace. `http://www.polyspace.com`.
14. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.
15. W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. of the 15th IEEEInternational Conference on Automated Software Engineering*, Grenoble, France, September 2000.
16. W. Visser, K. Havelund, G. Brat, S.-J. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), Arpil 2003.
17. R. Washington, K. Golden, and J. Bresina. Plan execution, monitoring, and adaptation for planetary rovers. *Electronic Transactions on Artificial Intelligence*, 4(A):3–21, 2000. http://www.ep.liu.se/ej/etai/2000/004/.

18. J. C. Widmaier, C. Smidts, and X. Huang. Producing More Reliable Software: Mature Software Engineering Process vs. State-of-the-Art Technology. In *Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland.*, pages 87–94. ACM Press, June 2000.