Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Program Model Checking as a New Trend

Klaus Havelund[1], **Willem Visser**[2]

[1] Kestrel Technology, NASA Ames Research Center, Moffett Field, CA 94035, USA, e-mail: havelund@email.arc.nasa.gov
[2] RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA, e-mail: wvisser@email.arc.nasa.gov

The date of receipt and acceptance will be inserted by the editor

Abstract. This paper introduces a special section of the STTT journal containing a selection of papers that were presented at the 7th International SPIN workshop, Stanford, August 30 - September 1, 2000. The workshop was named *SPIN Model Checking and Software Verification*, with an emphasis on model checking of programs. The paper outlines the motivation for stressing software verification, rather than only design and model verification, by presenting the work done in the Automated Software Engineering group at NASA Ames Research Center within the last 5 years. This includes work in software model checking, testing like technologies and static analysis.

1 Introduction

This special section contains a selection of five papers that were amongst the 17 papers and six invited talks and tutorials presented at the 7th International SPIN workshop, arranged at Stanford University, California, USA, August 30 - September 1, 2000. The original proceedings were published in Lecture Notes in Computer Science volume 1885, Springer, titled: *SPIN Model Checking and Software Verification*. Model checking is a technique for exploring all possible execution sequences of a system of interacting concurrent components. Such systems may interact in unexpected ways due to unpredictable speeds of the various components, and are hence extremely difficult to test using traditional testing techniques. The many ways components can interact usually leads to a large search space, and model checkers typically incorporate various techniques for conquering this complexity. The SPIN model checker [36], for which Gerard Holzmann recently received the ACM Software System Award, has a large user community, and the SPIN workshop is a forum for this community, and generally for researchers with interest in automata-based, explicit state model checking technologies for the analysis and verification of asynchronous concurrent and distributed systems. The first SPIN workshop was held in October 1995 in Montreal. Subsequent workshops were held in New Brunswick (August 1996), Enschede (April 1997), Paris (November 1998), Trento (July 1999), and Toulouse (September 1999).

Traditionally, the SPIN workshops present papers on extensions and uses of SPIN. As an experiment, SPIN 2000 was broadened to have a slightly wider focus than previous workshops in that papers on *software verification* were encouraged, as reflected in the name of the workshop: *SPIN Model Checking and Software Verification*. In this paper we shall try to explain the background for emphasizing software verification. We will do this by outlining in the following sections some of the research that has taken place in our own verification research group at NASA Ames Research Center throughout the last years since its start in 1997, together with some thoughts on the future. The verification group is part of the Automated Software Engineering (ASE) group, the purpose of which is to develop software technology for supporting software development within NASA. The selected papers will be introduced and related to this work in special subsections throughout the presentation.

By software verification we mean model checking of source code (or the corresponding object code it is compiled into). This is in contrast to analysis of designs or models of software, which are usually much more abstract. That is, we suggest to focus attention on the real beast in all its complexity. Although this view at the time of writing seems to have caught on as a popular research topic, at the time leading up to the workshop, this subject was only investigated by few research groups, including our own. Amongst the other work in this domain at the time was [4] and [11], and in fact only [4] was

known to us when we started. Now SPIN has a C interface [37] and can hence model check C programs, and other tools exist as will be elaborated in later sections.

Although targeting source code may appear as just worsening the problem of state space explosion usually associated with model checking, we believe that there are some benefits from such an approach, as we shall outline here. Note, that we do not suggest that design or model verification is uninteresting - far from. However, our experience throughout several experiments during 1996 and 1997 at NASA as well as with a Danish audio video company lead to the (folklore) conclusion that programmers often write code without first writing a detailed design. We concluded that if formal methods were to be adopted at NASA within a shorter time frame, we would have to provide a technology that could analyze real programs.

One can argue that programmers should be urged to write formalized designs that can be analyzed. However, a point of view may be that in order for a design to contain enough information to be useful for formal analysis, the design may approach the final system in complexity, in which case programmers will avoid the extra work and just write the code directly. This fact may be the reason why software developers do not create detailed designs as do engineers in other disciplines. The distance for example between a design of a bridge, and the bridge itself, is enormous, and therefore the design is well motivated. In case the code is generated from the design, the design becomes the code, and we are left with code analysis anyway. Even a mainly graphical design language such as UML raises the issue of program verification since UML designs can contain code fragments, and can evolve into fully fledged programs.

This new trend brings new challenges into focus, such as dealing with object oriented dynamic memory allocation and garbage collection, program libraries, and, last but not least, an increased state space to explore. These problems require new approaches, amongst them perhaps the most challenging being how to deal with really big state spaces. Techniques to deal with this include for example static analysis, abstraction, guided search, and intelligent testing techniques in between complete state space exploration such as model checking on the one hand, and partial search, such as simulation on the other. We believe that this is an interesting research direction for the formal methods community for the following reasons. First of all, if tools can handle real programs, the user community will increase dramatically. Second, programming languages often offer quite convenient notations for expressing solutions to problems, compared to modelling languages. Third, by trying to handle real programs, the scalability issue becomes much more pressing, and will therefore spawn new research to develop more scalable solutions that can even help in design verification. Fourth, and finally, researchers from different groups that develop model checkers for the same programming language will be able to exchange examples and compare technologies very easily.

The following sections proceed as follows. In Section 2 we describe a case study where SPIN was applied to the analysis of a space craft controller, successfully identifying several errors. This and other previous case studies lead to the development of the Java PathFinder 1 system: a translator from Java to the PROMELA language of SPIN, described in Section 3. This system allows to model check programs written in a non-trivial subset of Java. Section 4 describes another case study where SPIN was applied to analyze an avionic real-time operating system. Java PathFinder 1 was limited in the sense that it could not handle the Java libraries well. Translating the libraries would give too large PROMELA models and writing stubs for them would require an enormous amount of work. Hence it was decided to model check Java byte code instead, based on a homegrown Java Virtual Machine. This effort is described in Section 5. Section 6 identifies some technologies that are regarded as essential to make model checking of software work. This includes topics such as abstraction and search heuristics. One of our more recent research topics is runtime verification, as described in Section 7, where scalability is achieved by just examining single execution traces. Lastly, some final thoughts are given in Section 8.

2 The Remote Agent Example

2.1 Description of the Remote Agent

The first verification case study that was performed in the newly created Automated Software Engineering group at NASA Ames was the application of the SPIN model checker to analyze part of the *Remote Agent* space craft controller [29,28]. The Remote Agent is a software system based on artificial intelligence techniques such as planning and scheduling. It is meant to execute on board the space craft and it's purpose is to take over part of the operations normally carried out on ground during the operation of a space craft, thereby relieving ground personal from micro-managing the space craft, and instead focus on higher level goal management. The Remote Agent was tested on board the *Deep-Space 1* space craft during May 1999. The space craft itself was launched on October 24, 1998. It was the first demonstration of a complete take over of a space craft by an artificial intelligence based software system in NASA's history.

The Remote Agent consists essentially of three modules: a *Planner*, an *Executive*, and a *Diagnosis* module. The standard operation of the space craft using this system may proceed as follows: a *goal* is created by ground personal, for example "move towards the comet and take a picture", and up-linked to the space craft. The planner on board will then from this goal generate a plan using a set of sophisticated search algorithms, based on

a static predefined model of what the possible transitions are relative to a current state. The result is a plan specifying a sequence of tasks for each relevant component on board the spacecraft, that must be performed in succession in order to achieve the goal. Tasks from different components may run in parallel according to certain time constraints generated as part of the plan. The plan is then sent to the Executive, which executes the plan, thereby operating the space craft. The diagnosis module constantly monitors the behavior of the craft and compares the observed behavior to the expected, signaling the executive, or in worst case the planner, if something goes wrong, whereafter proper action can be taken to repair the situation.

The Executive was selected for the verification case study, and in particular the language named Esl (Executive Support Language), implemented as an extension to multi-threaded Common Lisp for supporting the execution of tasks. Esl is essentially an API for multi-task programming similar to POSIX threads, but with extra domain specific functionality.

2.2 Model Checking

The Esl module consisted of approximately 3000 lines of code. Initially we had a choice between various possible verification tools, mainly theorem provers and model checkers. We quickly decided that theorem proving would be too time consuming for an experiment limited to a couple of months of duration, and our goal was to find errors, and not to prove complete correctness. We decided to use SPIN since it already had a programming language like syntax and since it allowed dynamic process creation, one of the features of the system.

From the 3000 lines of LISP code we extracted approximately 500 lines of PROMELA code, representing an abstraction of the original system. The abstraction was made based on informal reasoning, focusing attention on a lock table that all threads were accessing. By asking engineers, two properties were formulated in SPIN's Linear Temporal Logic (LTL) and verified against the model. Neither of the two properties turned out to be satisfied by the model, and a total of 4 classical concurrency errors were revealed, each of which had a counterpart in the original code, as confirmed by the programmer. They were classical concurrency errors in the sense that they could occur due to totally unexpected interleavings of tasks, interleavings that had not been detected by traditional testing. As an example, one of these violations was caused by a missing critical section around a piece of code of the form:

```
if(no_new_events())
  goto_sleep()
```

This would cause the executing thread to execute the condition `no_new_events()`, and in case it evaluated to true, decide to go to sleep. However, if a new event occurred in between the condition and the actual call of `goto_sleep`, the thread would miss the new event and just go to sleep.

The programmer of the system was very impressed by these results, as documented in [29]. As an interesting aftermath, when the Remote Agent was activated on May 18, 1999, an anomaly occurred: thrusting did not turn off as requested. The experiment was immediately terminated from ground and put in stand-by mode for 5 hours until the reason for the error had been detected. It turned out to be a missing critical section around a piece of code similar the one above, but in a different part of the system that had not been analyzed with SPIN. One thread would then block, missing an event, and the whole system would eventually deadlock. We had hence demonstrated to NASA that model checking successfully can find errors that can damage a mission.

2.3 Lessons Learned

The experiment was regarded as successful by all involved parties. Not only had 4 errors been found that were very hard to find with normal testing, one of these actually also demonstrated a major design flaw in the system. Furthermore, one of the errors was later reintroduced in another sibling module, causing deadlock during flight that put the space-craft in stand-by mode for several hours.

However, observing the verification process, the result was not so encouraging. Twelve man-weeks (two researchers during 6 weeks) were spent on creating the 500 line PROMELA model from the 3000 lines of LISP code. The LISP code was undocumented and used many layers of macros, which made it difficult to read. Just understanding the code in order to make a proper translation was definitely one of the problems. A second problem was to define the mapping from the very powerful LISP language to the less powerful PROMELA language. A third problem was to decide what parts to translate and for those parts, whether the translation should be one-to-one, or some abstraction. It was clear to us that the first two problems (understanding and translation) were the hardest, while the abstraction problem strangely enough was less of a problem. This gave us the hope that if the translation could be automated, and the verification was performed by the programmer himself, using some kind of semi-automated abstraction support tool, then the experiment could potentially have been done within a single day.

Another important source of experience supporting the construction of a software model checker was the application of the UPPAAL real-time model checker [39] to analyze two audio video systems developed by the Danish audio video company Bang & Olufsen [26,25].

The verification effort was very successful in one occasion ([26]) in that a 10 year old known, but unexplained, bug was found and explained. However, as was the case with the Remote Agent study, some time was spent on manually creating a model from the program, in one case 2500 lines of assembler code. As a result of these experiences we decided to create a translator from a programming language to PROMELA, as described in the next section. The first idea of developing a Java model checker was in fact conceived during the work with UPPAAL in 1997.

3 Java PathFinder 1

3.1 Rationale

As outlined in the previous section, a series of experiments with applying existing model checkers in the modelling and analysis of software systems had lead to the observation that it would be extremely useful if model checkers could analyze programs written in traditional programming languages. We therefore decided to develop a model checker for some well chosen programming language, and the choice fell on Java.

There are several objective reasons for choosing Java. First, it was viewed as important that the chosen language was object oriented since that was the current trend in programming language design. Second, the language should also be popular in order to gain a broader user community. These criteria ruled out C (not object oriented) and LISP (not popular). C++ was regarded as too complicated for formal analysis due to its rich syntax and capabilities for operating with pointers. Java was hence the obvious choice for the above reasons. However, NASA currently operates mostly in C, and in some cases in C++. LISP was only used for the Remote Agent experiment and has been abandoned for future missions. This gave us the burden of arguing going for Java. Our response would be that Java would be good for prototyping the ideas, and potentially Java could become the language of the future. As it turns out, experiments are currently undertaken within NASA to evaluate Java as a possible replacement of C and C++. The occurrence of Real-Time Java may have an important role to play in this decision.

The development of a model checker for Java could again take a number of avenues. One could either write a model checker from scratch for Java, or write a translator from Java to the modelling language of some existing model checker. The SPIN model checker was early on regarded as either the target for a translation, or at least an example of how one could write a new model checker for a programming language. The PROMELA language has a high resemblance to a programming language. One of the salient features of PROMELA is the capability of dynamic process creation. We early on imagined that this could be used to model dynamic thread creation as existing in Java.

It was finally decided to write a translator from Java to PROMELA, the modelling language for SPIN, since it would potentially require less work than writing a model checker from scratch. The project was named Java PathFinder (JPF) [30], later to be named Java PathFinder 1 (JPF1), after the Mars PathFinder rover that explored Mars in 1997. The goal was to produce a prototype relatively fast in order to evaluate the potential of model checking real programs. At the time, only a source-to-source code translation was considered. Java source code is compiled into byte code by the compiler, and hence an alternative approach would have been to translate byte code to PROMELA. This latter approach was, however, hardly considered at the time, possibly reflecting a belief that byte code verification would be too inefficient with too many detailed interleavings between single byte code instructions. As it turned out, as described in Section 5, when we later concluded lessons learned from the JPF1 project, byte code verification actually turned out to be a very viable solution.

3.2 Design and Implementation

JPF1 translates a Java program into a PROMELA model. The Java program can contain assertions as calls to an `assert` method, which will be translated into calls of PROMELA's assert statement. The resulting PROMELA model can then be checked for assertion violations and deadlocks. There is also a possibility, of course, to check general LTL formulae on the resulting PROMELA model, although this requires some minimal knowledge about the generated PROMELA code. Error traces produced by SPIN are visualized using SPIN's message sequence charts, assuming that special print statements have been inserted into the code. JPF1 does not apply any analysis to reduce the state space of the generated model. Hence, the Java program must have a finite and tractable state space.

The translator is developed in LISP, and comprises 6000 lines of code. We have used an already existing parser front-end written in Moscow-ML by Peter Sestoft (the Royal Veterinary and Agricultural University in Denmark), ported from a Standard-ML version written by Olivier Brunet and Gordon Woodhull (University of California, Berkeley, USA). The parser handles Java 1.0, an early version of Java. As a result, the translator translates a subset of Java 1.0. However, a significant subset of Java 1.0 is supported by JPF1. This includes: class definitions with class variables, fields and methods; simple data types such as integers, booleans, object references and arrays of all these types; class inheritance; dynamic object creation; threads and synchronization primitives such as `synchronized` statements and the `wait` and `notify` methods; exceptions and thread interrupts; and

finally most of the standard programming language constructs such as assignment statements, conditional statements and loops. Amongst the features not translated are: packages (the parser could only read one package), overloading, method overriding, recursion (since method calls are translated by inlining), strings, floating point numbers, some thread operations like `suspend` and `resume`, and some control constructs, such as the `continue` statement. Furthermore, arrays are not objects as in Java since they are modelled using PROMELA's own arrays to obtain an efficient verification. Finally, but perhaps most importantly, the translator can not translate the pre-defined class library, including for example numerous container classes. In spite of these omissions, JPF1 at the time translated more of Java than any to us known other similar tool.

A key design issue was how to translate dynamic object creation. Dynamic object creation is handled by for each class to define an array of fixed size, each entry of which corresponds to the data area of the class. Hence, for example if a class has two variables `x` and `y`, then an array of records containing these two variables is generated. The size of the array sets a limit on how many objects of the class can be generated, and must be re-defined by the user if the default value is not satisfactory. An index variable always points to the next free object. An object reference is a pair consisting of the name of the class and an index variable pointing into the corresponding array. Another key issue was the translation of dynamic thread creation and the various thread synchronization constructs. Threads are naturally mapped to PROMELA processes. The key synchronization constructs, such as the synchronized methods, the `synchronized` statement, and `wait` and `notify`, are handled by introducing extra variables in the data area for each object (in the array corresponding to the class). For example, locking an object is modelled by introducing a `LOCK` variable, which by default contains `null`, and which is assigned the thread id of any thread locking the object. Another thread cannot access the object in case this variable differs from `null`. Similarly, a PROMELA zero-capacity (synchronous) channel variable is introduced to model the `wait` and `notify` operations: waiting corresponds to executing a "?" operation on the channel and a notification corresponds to executing a "!". A major feature of the translator is that it can handle exceptions and the `finally` construct. Exceptions are translated by using the `unless` construct of PROMELA[1] A special variable `EXN` is introduced in each thread object, holding the default `null` value. An exception (which is an object in Java) is thrown by storing the exception object into this variable, which again triggers the surrounding `unless`-constructs, which are of the form `P unless EXN != null`.

[1] Gerard Holzmann introduced a special `-J` (for Java) option in SPIN to interpret `unless` from inside-out rather than from outside-in, in order to make it useful for this translation.

3.3 Lessons Learned

JPF1 was considered a successful tool, achieving attention from various research groups. The tool was applied to a game server consisting of 1400 lines of Java code in 16 classes [34]. Although the example was not very big, it was non-trivial, and not written with formal verification in mind. A suspicion about a deadlock in the system was confirmed using JPF1. The tool was also applied to analyze the Remote Agent after it deadlocked in space, as described in [28]. In this case the space craft engineers at JPL in Los Angeles informed us that a deadlock had occurred and challenged us if we could find the error using model checking. We did find the error, however discovering it though code review since we had seen it before as described in Section 2. However, JPF1 was used to confirm that it was an error.

It was clearly felt that smaller Java programs of up to 2000 lines of code could be handled with this kind of technology[2]. This could either mean that the technology was well suited for unit testing, or perhaps for testing even larger systems using abstraction before the application of the tool. However, the tool itself had some drawbacks concerning its applicability. As described earlier, although a considerable subset of Java was translated, not all was translated, and in particular not the pre-defined Java library. It was regarded as impractical to translate the library using JPF1 (even if we had the sources). Hence, a program would have to be modified in order to fit the well-formedness criteria of the translator if it used the library, and most Java programs do. Also, there were other translation omissions, such as recursion, that seemed hard to capture considering the then existing translation framework. In general, it was the perception that the closer one got to cover 100% of Java, the harder it became to extend the translator. As it turned out, working at the byte code level would solve all these problems, without costing a big loss of efficiency.

4 The DEOS Case Study

In 1998 Honeywell Technology Center approached the ASE group with a request to investigate techniques that would be able to uncover errors that testing is not well suited to catch [43]. The next generation of avionics platforms will shift from federated system architectures to *Integrated Modular Avionics* (IMA) where all the software runs on a single computer with an operating system ensuring time and space partitioning between the different processes. For certification of critical flight software the FAA requires that software testing achieves 100% coverage with a structural coverage measure called

[2] Evidently, the complexity of a program cannot be purely measured in terms of lines of code, but rather one has to consider the amount of interleaving possible between threads

Modified Condition/Decision Coverage (MC/DC). Honeywell was concerned that 100% structural coverage would not be able to ensure that behavioral properties like time-partitioning will be satisfied. In particular, they developed a real-time operating system, called DEOS, where an error in the time partitioning of the O/S was not uncovered by testing. As an experiment the ASE group undertook the challenge of finding this error with a model checker without knowing what it was, where it was, or even, how to check for it. In a kick-off meeting Honeywell visited the ASE group and discussed the basic functionality of DEOS, and subsequently produced a slice of the O/S that contained all the code required to show the error. The code that was analyzed was 1500 lines of C++ code (full DEOS is 10000 lines of code).

Since we didn't have a model checker that could take C++ as input we were forced to again translate the code to a suitable model checker's input notation. However, unlike with the Remote Agent we decided to do a methodical 1-to-1 mapping between the code and the model checker input, so that we could avoid first understanding all of the program. We again chose the SPIN model checker since the PROMELA language was the closest model checker input to a real programming language, like C++. The translation scheme we used was based on the Java PathFinder 1 approach for dealing with object oriented programs (see Section 3).

The error was found in 3 man-months work: divided between 1 man-month translating the C++ code to PROMELA and 2 man-months finding the error. In the Remote Agent case it took 3 man-months to translate the code, and two man-weeks to do the analysis. This difference can easily be explained by the differences in the two systems: one system was nearing the end of its design cycle, written to be certified, tested thoroughly and contained only one very subtle error (DEOS), the other was in the middle of its development cycle, written in a semi-research environment, tested by the developers and contained a number of errors (Remote Agent).

The analysis of the DEOS system was very well received by Honeywell and subsequently the DEOS system became the focus of a number of research efforts [43, 50, 17]. Honeywell proceeded in creating their own model checking team to analyze future DEOS enhancements as well as the applications to run on top of DEOS [7]. Honeywell is continuing to extend the DEOS PROMELA model to support verification of more complex versions of DEOS.

4.1 Lessons Learned

From a research perspective the work on DEOS validated our hypothesis that real programs can be analyzed directly, however DEOS also showed us some other problems:

- Model checking programs directly shifts the burden of work from the translation of the code to the model checker's input to the analysis of the code.
  - Typically the translation from code to model checker involves some ad-hoc abstraction and slicing of the code, that makes the model checking more efficient.
  - When this translation is done 1-to-1 it means much of this clever encoding that was previously done by the human translator now needs to be done by clever tools with minimal human input.
- Creating an environment for the program to execute in during model checking can be very challenging
  - Model checkers can only analyze closed systems hence any system to be analyzed must be supplied with an environment to drive it. This is analogous to creating a test-driver and selecting test cases to support testing.
  - Creating an environment for DEOS to show the error occurring took the most time in the DEOS model checking (2 man-months).

4.2 Related Paper in this Special Section

Traditionally the SPIN workshop has had a strong focus on the use of SPIN in real-world case studies - similar to the DEOS case study described here. In keeping with this tradition the paper by Brinksma, Mader and Fahnkar, entitled *Verification and Optimization of a PLC Control Schedule* describes the use of SPIN (as well as UPPAAL [39]) for the analysis of a programmable logic controller system. What makes this contribution novel is that firstly the PLC controller is a real-time system and SPIN doesn't support real-time directly (the paper also describes a comparison study with UPPAAL that does support real-time), and secondly, that not only correctness properties of the controller are considered but also optimization issues in the use of the controller. One of the contributions of this paper is the use of *variable time advance* for handling real-time in SPIN, we adopted this approach also in the analysis of DEOS.

5 Java PathFinder 2

5.1 Rationale

As pointed out in Section 3 the Java Pathfinder 1 (JPF1) model checker was highly successful, but had a number of drawbacks that limited its effectiveness. Essentially the main reason for this was the translation based approach adopted: although SPIN is a very powerful model checker and the PROMELA language very expressive, the mapping between Java and PROMELA is not straight forward. Java PathFinder 2 (henceforth JPF2) was developed to address the shortcomings of JPF1 (see Section 3):

1. Handle all the language features of Java
2. Handle Java libraries
3. Allow more flexible approaches to model checking Java programs

The major design decision for JPF2 was to base it on a custom-made Java Virtual Machine (JVM) that could execute all Java bytecodes. This addressed issues 1 and 2 from above, since all of Java could now be model checked and also all Java libraries. We addressed the third issue by designing JPF2 in a modular fashion in order to allow many different search strategies to be easily integrated into the model checker.

A number of different research groups have worked on Java model checkers, but most of these have been based on the translation approach as used for JPF1 [11,9]. To date, JPF2 is still the only model checker that can handle all the language features of Java. The only other custom-made model checkers that address real programming languages are, dSPIN [12] an extension of SPIN that can handle dynamic memory creation and functions, the new version of SPIN that can handle a subset of C, and the SLAM model checker [1] that checks reachability properties of sequential C programs.

5.2 Design and Implementation

JPF2 is written in Java which made the development of a custom-made JVM quite easy - one could exploit the fact that we were doing "Java-in-Java" by allowing the underlying JVM to handle the implementation of some of the tricky bytecodes such as floating point division (FDIV). We believe that since we wrote JPF2 in Java, it contributed to the fact that a prototype system that had similar functionality as JPF1, was completed in only 3 man-months.

JPF2 is an explicit-state model checker which means it enumerates each reachable system state from the initial state and in order to not redo work (and therefore terminate) it is required to store each reached state. When analyzing a Java program each state can be very large and thus require much memory to store, hence reducing the size of systems that can be handled during model checking. This was the fundamental problem that had to be solved for JPF2 to work. Others considered this problem too hard and developed so-called state-less model checkers (i.e. they don't store states and therefore do a partial state-space search) [20]. In JPF2 this problem is solved by using novel state-compression techniques [41] that reduce the memory requirements of the model checker by an order of magnitude. Another novel feature of JPF2 is the use of symmetry reduction techniques to allow states that are the same modulo where an object is stored in memory to be considered equal [41]. Since, object-oriented programs typically make use of many objects, this symmetry reduction often allows an order of magnitude less states to be analyzed in a typical program.

JPF2 uses the BANDERA [9] toolset for specifying the properties to be analyzed, the display of the error-path if one exists, as well as for certain forms of abstractions and slicing. BANDERA supports the specification of predicates within Javadoc comments that can be used to check linear temporal logic (LTL) behavioral properties as well as pre- and postconditions for methods. To handle LTL properties JPF2 has a front-end translator from LTL to Büchi-automata [19] that is highly optimized to produce succinct automata. The JPF2 model checking algorithm then checks whether all program behaviors comply with the behaviors described by the automata, using a highly optimized algorithm based on the work presented in [51].

JPF2 also supports distributed memory model checking, where the memory required for model checking is distributed over a number of workstations [41]. Although this technique requires an additional time-overhead due to the sending of messages over a network, it allows examples to be analyzed that previously would not fit in the memory of one workstation. The crucial factor for the success of distributed model checking in this fashion is how to partition the memory across the different workstations — in [41] we investigated a number of partitioning schemes and found that dynamic partitioning (partitions evolve during model checking rather than being statically fixed at initialization) worked best.

5.3 Lessons Learned

JPF2 has been successfully used in a number of projects, most notably the DEOS error (from Section 4) was rediscovered in a Java translation of the original code. More recently, 7000 lines of code from a Mars rover was successfully analyzed. The JPF2 system was made available to the user community via a web download in February 2001 and since then more than 100 organizations have registered to use the tool. More importantly, JPF2 has had the desired effect of becoming a vehicle for research on analyzing programs with model checking: we have close collaborations with the BANDERA group at Kansas State University as well as other groups at CMU, Stony Brook, Minnesota, Freiburg and Liverpool Universities.

The development of JPF2 was the culmination of 3 years of research in the application of model checking to software within the ASE group. In many ways however it is the stepping stone for the future: instead of worrying about how to encode a program in some model checking notation, one can rather think of the behavioral properties one would like to check, which parts of the program to abstract to make the model checking more tractable, and how to improve model checking for specific classes of programs. These issues will all be discussed in the following section.

5.4 Related Papers in this Special Section

As mentioned in Section 5.2 JPF2 supports LTL model checking through the use of the BANDERA toolset to describe the properties to be checked on the Java programs. In this special section the language for describing these properties, namely the BANDERA Specification Language (BSL), is outlined in detail in the paper by Corbett, Dwyer, Hatcliff and Robby, entitled *Expressing Checkable Properties of Dynamic Systems: the BANDERA Specification Language*. The BSL language has recently been fully integrated with JPF2.

An important component of explicit-state model checking is how to check temporal properties efficiently. JPF2, as well as SPIN, uses the so-called automata-theoretical approach where each LTL (linear time temporal logic) formula is translated to a Büchi automaton before model checking commences. This translation from LTL to Büchi automata has been the focus of much research, and a number of tools for doing such a translation exist (including the one used in JPF2 [19]). However, doing this translation efficiently is non-trivial and therefore also error-prone. The second paper, by Heikki Tauriainen and Keijo Heljanko, entitled *Testing LTL Formula Translation into Büchi Automata*, deals with this, somewhat overlooked, area of the correctness of LTL to Büchi translators. We will soon by relying on their technique also to the test the LTL to Büchi translator used within JPF2.

6 Technologies for Software Model Checking

For model checking to make an impact on the quality of programs produced the amount of human effort in operating the tools should be kept to a minimum. With JPF2 we have reduced the amount of effort considerably since a translation phase is no longer required. However, because the automated translation preserves all the details of the software implementation, the model checking itself is more difficult. The reason is that manual translation, typically involves significant optimization and abstraction of the system. Therefore, to truly reduce the amount of manual effort and place model checking into the development loop, we need tools to support the typical optimizations and abstractions previously done during translation. In general, the goal is to reduce the state-space of the system that the model checker needs to analyze, providing both scalability and responsiveness.

6.1 Abstraction

When using abstraction techniques to reduce the number of states of a system one can either remove some behaviors present in the original system (under-approximations) or introduce new behaviors not present in the original (over-approximation).

6.1.1 Under-approximations

Under-approximation of the behaviors is by far the most common form of manual abstraction before model checking. Under-approximation doesn't preserve correctness, i.e. if the abstract system satisfies a behavioral property then it doesn't follow that the original system does as well. Under-approximation are however good for finding errors, since an error in the abstract system implies the same error in the original [50]. JPF2 was built with the view that it should cover the spectrum of analysis techniques from testing, where only one execution of a program is analyzed, to model checking where all the paths are analyzed, hence JPF2 supports a number of techniques for doing under-approximations during model checking (we highlight two below).

Race-Guided - where a race-analysis is done on the program first and if a race violation is found the model checker focuses on the threads involved to see what the race violation might lead to. We used this technique to find the error in the Java translation of the Remote Agent error that occurred during flight [52].

Heuristic Search - using techniques from AI we can apply either general or program specific heuristics to guide the search towards likely errors. For example to find deadlocks we use a heuristic that tries to maximize blocked threads - this heuristic found the Remote Agent deadlock within seconds whereas in exhaustive mode the model checker will fail due to memory limitations. We also developed a heuristic for finding problems that are due to thread-interleaving and lastly, one based on trying to increase structural testing coverage [22]

6.1.2 Over-approximations

With this technique one represents a group of states in the concrete (original) program by a small finite set of states in the abstract program — and can therefore lead to huge state-space reductions. This form of abstraction is inspired by *abstract interpretation* as first used in static program analysis [10], where the data domain (type) of a variable is replaced by an abstract type over which all concrete operations are then interpreted. Note that this type of abstraction causes more behaviors to be present in the abstract program than in the original program. The fact that more behaviors are possible in the abstract program means that if a behavioral property expressed in LTL holds in the abstract it also holds in the concrete, but if an LTL property fails in the abstract then it might not fail in the concrete (since it might fail due to a behavior found in the abstract that is not present in the concrete). Another very popular form of over-approximations is called *predicate abstraction* [21, 1]: here one replaces a predicate used in the program by a boolean variable and all updates to the variables

in the predicate are changed to updates of the boolean variable.

JPF2 supports predicate [50] and BANDERA supports type-based abstraction [17]. In order to handle over-approximations of the program behaviors we have extended Java with two special method calls that signals nondeterministic choice (random(n) that return values between 0 and n inclusive and randomBool() that return true or false) — whenever the model checker encounters these methods it will nondeterministically try all possible results for each call.

Predicate Selection The first problem one encounters with the application of over-approximations in practice is how to select the parts of the program to abstract — typically this requires human intervention. In BANDERA type-based abstraction can be done automatically though, by doing a backward dependency analysis of the program from all points that directly influence the temporal property to be checked, to determine a set of variables that *most* influence program behavior (with respect to the property to be checked) and these variables then become candidates for abstraction [17]. Although predicate abstraction can be applied automatically too, by selecting all predicates in the program and in the property, we have found that in practice this leads to too many spurious counter-examples (i.e. too many behaviors not present in the original that then lead property violations).

Abstract Program Creation Both predicate and type-based abstraction can be applied either during or before model checking. However in practice, the calculations required to determine the abstract state of a program is too slow to be done during model checking, and therefore we only use abstract program creation before model checking in JPF2 and BANDERA. In order to calculate an abstract operation, given an abstraction mapping (type or predicate) and a concrete operation, one requires an automated theorem prover (i.e. a set of decision procedures for the domain). For predicate abstraction we use the Stanford Validity Checker (SVC) [2] to calculate abstract statements and for type-based abstraction BANDERA uses PVS [42]. Although most of the abstraction calculations are done before model checking, object-oriented programs are particularly challenging for predicate abstraction, since predicates may relate variables from different classes that during execution can have a number of instantiations. Predicate abstraction is typically done in a *static* setting, whereas with object-oriented programs predicates can be created dynamically during execution when new objects are instantiated. JPF2 therefore supports mechanism to allow predicates to be created on-the-fly during model checking when predicates are specified across different classes [50].

Result Interpretation The biggest drawback of over-approximation based abstractions are that errors found can be spurious (i.e. not present in the original). The more *aggressive* the abstraction, i.e. the bigger state-space reduction one achieves, the more likely it will be that a spurious error will occur. It is a well known fact that users of systems where spurious errors can be reported are more likely to complain about the spurious errors than if it reported no errors (supported by data presented by Microsoft after using their static analysis tool PREfix for discovering run-time errors [49]). For a program model checker using abstraction to be of practical use it is therefore vitally important that spurious errors be eliminated. JPF2 supports a novel technique for achieving this goal: as a first-pass after abstraction it only searches the parts of the abstract program's state-space that it knows contains no behaviors that are not also part of the concrete program [47]. One can view this as doing a on-the-fly under-approximation of the state-space generated from doing an over-approximation of the original program. This technique has been remarkably successful: both the Remote Agent and DEOS examples' bugs can be found using abstraction and this search technique.

Abstraction Refinement An abstraction can be too coarse in certain situations, i.e. a spurious error cannot be removed unless the abstraction is refined. JPF2 supports a very practical approach to determining where a refinement is necessary: the path reported by JPF2 as a counter-example over the abstract program is executed over the concrete and where the path diverges (if it doesn't diverge then of course the abstract path is not spurious) the predicates at that point are likely candidates to refine the abstraction. Refinement then proceeds by adding these predicates to the predicate abstraction and repeating the abstract program creation. This approach was first demonstrated in the Invest tool [3].

6.2 Slicing

Slicing is a technique that yields a precise abstraction (neither over nor under approximation) of the program behavior with respect to the property being analyzed [14]. A sliced program yields a smaller state space than the original un-sliced program, and hence, slicing allows the model checker to handle larger programs. There are two important aspects to selecting statements that will be eliminated. First, these statements should not appear on the dependence graphs of the statements containing variables that are terms of the property being checked. Second, the sliced program should still be executable (since JPF2 is an explicit-state model checker). Slicing in JPF2 is provided through the slicing capability of the BANDERA toolset [9]. In BANDERA, slicing is performed based on six types of dependencies [24]: three intra-thread dependencies which are usually found in sequential programs, namely *data*, *control* and *divergence* dependencies and three types of dependencies (*interference*, *synchronization*, and *ready* dependencies) that capture concurrency issues.

6.3 Partial-Order Reduction

The goal of partial order reduction is to exploit the commutativity of concurrent transitions to reduce the state space that needs to be explored by a model checker. This technique, which is well described in [5], relies on the concept of *independent* transitions. Two transitions are independent if the execution of one does not disable the other (and vice versa) (*enabledness* condition) and they result in the same state regardless of their execution order (*commutativity* condition). JPF2 relies on a stronger concept based on *safe* transitions [38]. In essence, a transition is *safe* if it is independent on any transition of any other thread. A partial order reduction scheme that selects only safe transitions, when some exist, for exploration is guaranteed to yield correct results.

From a static analysis point of view, identifying safe statements can be reduced to the problem of identifying objects that can escape the thread where they have been created. Indeed, if we can identify such objects we can identify objects that can be shared by different threads. Then, unsafe statements are those that access shared objects, as well as those that correspond to entering a monitor in Java (these ones are easily identifiable syntactically). Our "safe statement" analysis is essentially an aliasing analysis. In a first phase, we build the program call graphs associated with each thread. As we build these graphs, we identify some escaping objects (they are passed as arguments to the class constructor of the thread). It is easy to realize that all other escaping objects are aliased to the escaping objects identified in the first phase. Therefore, the second phase consists of an aliasing analysis. Note that we do not have to compute aliases created by considering all interleavings (which is quite costly). Indeed, all escaping objects are identified by computing "intra-thread" aliases. This means that the complexity of our analysis is similar to the complexity of an aliasing analysis for sequential programs.

6.4 Environment Generation

An explicit-state model checker, such as JPF2, requires a *closed* system to analyze, i.e. a system and the environment it needs to operate in must be provided before model checking [16]. Often, however, the environment is not available and it needs to be created — during testing an analogous problem exists when a test-harness must be created, however a few subtle but important differences exist. For model checking it is important that all *relevant* environment behavior be present, whereas in testing a subset of all possible test-cases will be tested. Knowing which environment actions are relevant is however only possible with domain knowledge, something not always possible if the domain experts are not involved in the model checking (as is almost always the case in a research environment).

A common approach favored during model checking of systems without a known environment is to create the most aggressive environment, i.e. one that can perform any legal action at any possible time — often referred to as the *universal* environment [15]. If a property holds for a system composed with its most aggressive environment then the system will be correct when used in any environment. This is similar to the case where an over-approximation is done during abstraction. Unfortunately it also has the same problem as over-approximation in abstraction: spurious errors may result since the universal environment allows behaviors for which the system was not designed. A novel approach to remove such spurious behaviors is by filtering unwanted behaviors from the environment using LTL properties augmented with filter properties [15]. This technique was successfully used to create the DEOS system's environment [46] in only a few days rather than the 2 months used creating the environment manually.

6.5 Related Papers in this Special Section

Two of the papers in this special section are related to JPF2 as well as the state-space reduction techniques described in this section.

Firstly, the paper by Scott Stoller entitled *Model-Checking Multi-Threaded Distributed Java Programs* exploits the specific thread synchronization facilities in Java to optimize model checking by improving partial-order reductions (see Section 6.3). This work is illustrated within the context of doing state-less model checking (see Section 5.2) and is also implemented within JPF2.

Curbing the omnipresent state-explosion problem has been a fruitful line of research within the SPIN community as well as the model checking field in general. One popular technique for combating the state-explosion problem, not highlighted in this section, is to exploit symmetry reductions within the system that is being analyzed. The paper by Bosnacki, Dams and Holenderski, entitled *Symmetric SPIN*, introduces a symmetry reduction package for SPIN. The significance of this work lies not only in the theoretical contributions, but also in the fact that the research ideas were implemented within SPIN and is supported by empirical data. As mentioned in Section 5.2 JPF2 also supports symmetry reductions, but only for the objects instantiated within the Java program, whereas this work also handles symmetries in the process structure.

7 Java PathExplorer

7.1 Rationale

Since the Java PathFinder attempts to explore the entire state space of a Java program, storing the states

explicitly, it naturally suffers from the classical state space explosion problem. For very large applications one may therefore want to apply complementary techniques more closely related to traditional testing. Testing can be characterized as: "execute the program with different *test-cases* and *observe* each execution, comparing it to the expected behavior". Although we believe that the area of automated test-case generation has great potential, we think that its maturity is still at least 5 years out in the future. Also, providing a general, application independent, framework for automated test-case generation is not obvious. Engineers at JPL, in addition, expressed scepticism that such automation could be done, suggesting that it always at the end requires some engineer to sit down and think out what the test case should be. Our goal was to develop a technology that had a chance of being adopted by space craft designers within a relatively short time horizon (a couple of years). Our interest hence was turned on the observation part of the equation. The question was:

> How much information can be extracted about a program from observing a single execution trace?

It was our intention to develop a technology that could be applied automatically and to large full-size applications, with minimal modification to the code. The SPIN 2000 workshop hosted two invited talks on two commercial tools in this category: Temporal Rover [13] and Visual Threads [23]. Temporal Rover monitors the execution of a program, and checks its behavior against a collection of temporal logic formulae written in a temporal logic. Formulae are written in the code as comments, and then translated into formula checking code, which is executed as assertions. Visual threads performs various concurrency error analysis, such as deadlock and data race analysis. In particular, it implements the Eraser algorithm [48] for detecting data races. It was decided to build a tool, Java PathExplorer (JPaX) [31–33], which combined the functionality of these two tools, and in addition added new functionality. The Java PathExplorer analyzes (explores) single executions traces.

Visual Threads is tightly coupled to Compaq's Alpha microprocessors, and in addition does not work properly on Java programs. Hence, one goal was to port some of the technology to work for Java. The Temporal Rover required manual instrumentation of the code. We decided that automated instrumentation is desired, and hence focused on providing that capability. In Temporal Rover one can for example state a property over a set of program variables. One then has to insert the property at each update of these variables manually. With automated instrumentation capability, the property checks will be inserted automatically at all updates. This work was also inspired by the MAC tool [40], which performs automated instrumentation.

7.2 Design and Implementation

Two kinds of event analysis are currently implemented.

Logic based monitoring consists of runtime checking formal requirement specifications written in high level logics by users of the system. Logics are currently implemented in Maude [6], a high-performance system supporting both rewriting logic and membership equational logic. One can naturally and easily define new logics in Maude, such as for example temporal logics [44], together with their finite trace operational semantics. Currently, JPaX supports two built-in logics, future time and past time linear temporal logic.

Error pattern analysis consists of analyzing one execution trace using various error detection algorithms that can identify error-prone programming practices that may potentially lead to errors in some executions. Two such algorithms focusing on concurrency errors have been implemented in JPaX, one for deadlocks and the other for data races: the Eraser algorithm [48]. It is important to note, that a deadlock or data race potential does not need to actually occur in order for its potential to be detected with these algorithms. This is what makes them very useful in practice. As an example, the deadlock algorithm works by building a graph of locks acquired during the execution, building a edge from a lock *L1* to a lock *L2* if some thread *T* holds *L1* while acquiring *L2*. The lock graph accumulates all such updates and a warning is issued if it eventually becomes cyclic.

An instrumentation module performs a script-driven automated instrumentation of the program to be observed. The instrumented program, when run, will emit relevant events to an observer, potentially running on a different computer, in which case the events are transmitted over a socket. The Java byte code instrumentation is performed using the powerful Jtrek Java byte code engineering tool [8] from Compaq. Jtrek makes it possible to easily read Java class files (byte code files), and traverse them as abstract syntax trees while examining their contents, and insert new code.

7.3 Lessons Learned

At the time of writing, PathExplorer is being applied to a couple of case studies at NASA Ames, with so far promising results. Deadlocks and data races have for example been located. Deadlock and data race analysis, however, is limited, evidently, to only that kind of errors. So although the technology is powerful, it only covers a smaller fraction of the errors usually contained in software. Temporal logic monitoring can be used to check a broader class of errors, although in this case an error has to actually occur in order to be detected. Runtime monitoring can potentially be combined with model checking, for example as described in our paper in the SPIN 2000 proceedings [27]. Here deadlock and data race analysis

has been integrated into the Java PathFinder tool in such a way that one can first run the tool in simulation mode where deadlock and data race potentials are detected in a very scalable manner, whereafter the model checker is started to focus in on the threads involved in the warnings. A major issue that current case studies demonstrate is that it is difficult for software engineers to generate requirements that a software system should satisfy, even in English. Hence, it is interesting that formalizing the properties, once provided informally, is not the main problem.

8 Summary

In the previous sections we tried to give a flavor of the research within the Automated Software Engineering group at NASA Ames, that led to the decision to focus the 7th SPIN Workshop on model checking software. The sections related to the different research activities described were given in a roughly chronological order of when the work started. The concept of the workshop was formulated in late 1999, which would place it somewhere in the early stages of the JPF2 (Section 5) and Java PathExplorer (Section 7) development. These two projects as well as the work on supplementary technologies for model checking (Section 6) are very much still ongoing.

A number of other projects (in the ASE group) in the general field of software verification and validation have started since the SPIN 2000 workshop, but since these are not directly related to the workshop we only briefly mention them here:

- We use the PolySpace Verifier [45] to check for runtime errors in Space Flight software, and have found errors in Mars PathFinder code as well as in code to run biological experiments on the International Space Station. PolySpace is a commercially available tool that uses static analysis techniques to discover errors.
- In a joint project with the University of Minnesota we are using JPF2 for test-case generation [35]. Within the context of this work we are currently extending JPF2 with the capability to do symbolic execution.

We would like to emphasize that we regard program analysis as a complementary technique to design analysis, and hopefully the two approaches eventually can coexist within a unified framework.

References

1. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proceedings of TACAS01: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, Genova, Italy, April 2001.
2. 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 *LNCS*, pages 187–201, November 1996.
3. Saddek Bensalem, Yassine Laknech, and Sam Owre. Invest: A Tool for the Verification of Invariants. In Alan Hu and Moshe Vardi, editors, *CAV'98: 7th International Conference on Computer Aided Verification*, volume 1427 of *LNCS*, pages 505–510, 1998.
4. Tierry Cattel. Modeling and Verification of sC++ Applications. In *Proceedings of TACAS'98: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 232–248, Lisbon, Portugal, April 1998. Springer.
5. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.
6. Manuel Clavel, Francisco J. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. The Maude system. In Paliath Narendran and Michaël Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 240–243, Trento, Italy, July 1999. Springer-Verlag. System Description.
7. Darren Cofer, Eric Engstrom, Nicholas Weininger, John Penix, and Willem Visser. Using model checking for verification of partitioning properties in integrated modular avoinics. In *Proceedings of the Digital Avionics Systems Conference*, 2000.
8. Seth Cohen. Jtrek. Compaq, `http://www.compaq.com/java/download/jtrek`.
9. James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limeric, Ireland., June 2000. ACM Press.
10. P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 4(2):511–547, August 1992.
11. C. Demartini, R. Iosif, and R. Sist. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, July 1999.
12. C. Demartini, R. Iosif, and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proceedings of the 6th SPIN Workshop*, volume 1680 of *LNCS*, 1999.
13. Doron Drusinsky. The Temporal Rover and the ATG Rover. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
14. M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In Olivier Danvy, editor, *Proceedings of the 1999 ACM Workshop on Partial Evaluation and Program Manipulation (PEPM'99)*, January 1999. BRICS Notes Series NS-99-1.
15. M. B. Dwyer and C. S. Păsăreanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 1998.
16. M. B. Dwyer and C. S. Păsăreanu. Model checking generic container implementations. In *LNCS 1766*.

Generic Programming—Proceedings of a Dagstuhl Seminar, 1998.

17. Matthew Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina Pasareanu, Robby, Willem Visser, and Hongjun Zheng. Tool-supported Program Abstraction for Finite-state Verification. In *Proceedings of the 23rd International Conference on Software Engineering (to appear)*, Toronto, Cananda., May 2001. ACM Press.
18. Dimitra Giannakopoulou and Klaus Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
19. Dimitra Giannakopoulouand and Flavio Lerda. From States to Transitions: Improving translation of LTL formulae to Büchi automata. In *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2002)*, Lecture Notes in Computer Science, Houston, Texas, 2002. Springer.
20. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.
21. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *CAV '97: 6th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, 1997.
22. Alex Groce and Willem 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.
23. Jerry Harrow. Runtime Checking of Multithreaded Applications with Visual Threads. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 331–342. Springer, 2000.
24. J. Hatcliff, J.C. Corbett, M.B. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Proc. of the 1999 Int. Symposium on Static Analysis*, 1999.
25. K. Havelund, K. G. Larsen, and A. Skou. Formal Verification of an Audio/Video Power Controller using the Real-Time Model Checker UPPAAL. In *5th Int. AMAST Workshop on Real-Time and Probabilistic Systems*, number 1601 in Lecture Notes in Computer Science. Springer-Verlag, May 1999. Bamberg, Germany.
26. K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *Proceedings of the 18*th *IEEE Real-Time Systems Symposium*, pages 2–13, Dec 1997. San Francisco, California, USA.
27. Klaus Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 245–264. Springer, 2000.
28. Klaus Havelund, Michael R. Lowry, SeungJoon Park, Charles Pecheur, John Penix, Willem Visser, and John L. White. Formal Analysis of the Remote Agent Before and After Flight. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, June 2000.
29. Klaus Havelund, Michael R. Lowry, and John Penix. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, August 2001. An earlier version occurred in the Proceedings of the 4th SPIN workshop, 1998, Paris, France.
30. Klaus Havelund and Thomas Pressburger. Model Checking Java Programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000. Special issue of STTT containing selected submissions to the 4th SPIN workshop, Paris, France, 1998.
31. Klaus Havelund and Grigore Roşu. Monitoring Java Programs with Java PathExplorer. In Klaus Havelund and Grigore Roşu, editors, *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science.
32. Klaus Havelund and Grigore Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
33. Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. EASST best paper award at ETAPS'02.
34. Klaus Havelund and Jens Skakkebæk. Applying Model Checking in Java Verification. In *Proceedings of the 6th SPIN Workshop*, 1999. In connection with FM99, Toulouse.
35. M. Heimdahl, S. Rayadurgam, and W. Visser. Specification Centered Testing. In *Proceedings of the Second International Workshop on Automated Program Analysis, Testing and Verification.*, Toronto, Canada, May 2001.
36. Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
37. Gerard J. Holzmann and Margaret H. Smith. A Practical Method for Verifying Event-Driven Software. In *Proceedings of ICSE'99, International Conference on Software Engineering*, Los Angeles, California, USA, May 1999. IEEE/ACM.
38. G.J. Holzmann and D. Peled. An Improvement in Formal Verification. In *Proc. FORTE94*, Berne, Switzerland, October 1994.
39. Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
40. Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
41. Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *Proc. of the 8th International SPIN Workshop*, volume 2057 of *LNCS 2057*. Springer-Verlag, May 2001.

42. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 1th International Conference on Automated Deduction (LNCS 607)*, 1992.
43. J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, Limeric, Ireland., June 2000. ACM Press.
44. Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
45. PolySpace. `http://www.polyspace.com`.
46. C. S. Păsăreanu. DEOS kernel: Environment modeling using LTL assumptions. Technical Report NASA-ARC-IC-2000-196, NASA Ames, july 2000.
47. C.S. Păsăreanu, M.B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted java programs. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, 2001.
48. Stefan Savage, Michael Burrows, Greg Nelson, Patrik Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
49. Microsoft Spec and Check Workshop, 2001. `http://research.microsoft.com/specncheck/`.
50. W. Visser, S. Park, and J. Penix. Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking. In *Proceedings of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, August 2000.
51. W.C. Visser. *Efficient CTL* Model Checking using Games and Automata*. PhD thesis, Manchester University, June 1998.
52. Willem Visser, Klaus Havelund, Guillaume Brat, and Seung-Joon Park. Model checking programs. In *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, Grenoble, France, September 2000.