Model Checking of Concurrent Java Code

Nastaran Shafiei

January 14, 2011

Abstract

Since software is playing a vital role in our lives, the quality and reliability of software is of great concern. In general, the cost dedicated to software verification is significantly higher than its development cost. In the last few decades, there has been a significant amount of work to develop different software verification techniques. This paper gives a general overview of some of these techniques that can be applied to concurrent code. These techniques are compared with one another. Among these techniques, we are most interested in model checking. We specifically focus on model checking of concurrent code written in the Java programming language. This paper also explains the structure and features of a widely used model checker for Java code, Java PathFinder (JPF).

1 Why do we Need to Verify Software?

In the last few decades, the involvement of software systems in human life has increased significantly. Nowadays, many aspects of our lives are affected by these systems and we use them on a daily basis. For example, every day we are confronted with devices like the telephone, television, automobile, elevator, automated teller machine, microwave oven, or services like online banking and online shopping.

However, in spite of the important role that these systems play in our lives, they are not reliable. Software systems are developed by humans and due to the limited capability of humans to handle complexity, errors in software cannot be avoided. Unlike hardware, software has a tendency to grow in size very fast which can make software more vulnerable to errors. Basically, it is nearly impossible to create error-free software. According to Paul Strassmann, former CIO of Xerox, “software can easily rate among the most poorly constructed, unreliable and least maintainable technological artifacts ever invented by man”.¹

One of the most notorious software errors of all the time is the error of Therac-25, which is a computerized radiation therapy machine. Due to this error, known as a race condition, between June 1985 and January 1987, at least six patients received massive radiation overdoses which resulted in deaths and serious injuries. Another notorious software error is the

¹http://www.strassmann.com/pubs/cw/bad-software.shtml
one that caused the explosion of the Ariane 5 rocket. According to the report by the Ariane inquiry board, on June 4 1996, only 40 seconds after the launch, the rocket exploded.\textsuperscript{2} They reported that the failure was due to a software exception. The exception occurred in a data conversion from a 64-bit floating point value to a 16-bit signed integer value in the Ada code. Since the converted result was too large to fit in a 16-bit signed integer, the data conversion instructions led to the exception that was not handled. There were some other data conversions in the code that were handled. The development of Ariane 5 cost the European Space Agency about $7 billion. The Northeast Blackout of 2003 which was a massive loss of electric power in parts of the northeastern United States and Ontario, Canada, also resulted from a software error known as a race condition. The cost of this outage is estimated to be between $7 and $10 billion.\textsuperscript{3}

As our lives are becoming more engaged with software systems, their correctness is becoming a more serious issue. However, detecting and fixing a software error can be very hard and time consuming. According to the U.S. Defense Department and the Software Engineering Institute at Carnegie Mellon University, there are about 5 to 15 errors in every 1,000 lines of code. According to a five-year study by the Pentagon, tracing each error takes about 75 minutes and fixing them takes two to nine hours each. It takes about 150 hours to verify 1,000 lines which costs roughly about $30,000.\textsuperscript{4}

As can be seen in Figure 1, the earlier the error is detected, the better [35]. The cost of detecting and repairing a software error during maintenance and operation is considerably higher compared to the early stages of the development. In later stages, the malfunctioning of the system can severely damage the reputation of the company and it can even be a threat to its survival. According to a study by the National Institute of Standards and Technology (NIST) in 2002, software errors cost the U.S. economy about $59.5 billion annually which is about 0.6 percent of the gross domestic product.\textsuperscript{5}

As pointed out earlier, software errors cannot be avoided and detecting them is a very hard task. In general, because concurrent systems are much more complex than sequential ones, detecting errors in these systems is notoriously difficult. The way that concurrent systems behave depends on the relative speed of the executions of different components in the system which cannot be predicted. Hence, the behavior of a concurrent system is non-deterministic. It is very hard for programmers to consider all possible interferences of concurrently running components.

It is becoming increasingly difficult to follow the traditional path to increase the performance of processors, by switching transistors at ever greater speeds. This is mainly the case because the amount of power used and the amount of cooling technology needed also increases when the speed at which transistors switch does [44]. Nowadays, there are many sequential systems that are being replaced with concurrent versions. That is done by putting multiple processors, or cores, on a single chip.

As pointed out earlier, software errors cannot be avoided and detecting them is a very hard task.

\begin{itemize}
\item \textsuperscript{2}http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf
\item \textsuperscript{3}https://reports.energy.gov/
\item \textsuperscript{4}http://www.businessweek.com/1999/99_49/b3658015.htm
\item \textsuperscript{5}http://www.nist.gov/director/planning/upload/report02-3.pdf
\end{itemize}
difficult and tedious process. But, since errors in software can have extremely undesirable consequences, there is an essential need for techniques that can help to detect errors in software systems. These techniques are invaluable to developers and can cut the cost significantly, especially if the error is detected in the early stages of the software development.

2 Different Software Verification Techniques

As we have seen in the previous section, it is very useful to develop techniques that can be used to detect errors in software systems. In the last few decades, there has been a considerable amount of work on developing techniques to verify code. The goal of software verification techniques is to check that software behaves in a way that it is supposed to. The behavior of the software is outlined in the system’s specification which is a document that includes all the properties that the software should satisfy. Basically, the software is said to be correct if it is performing in a way such that all of its properties are satisfied.

It should be noted that, in practice, code verification is undecidable. According to Rice’s theorem no algorithm exists that can always decide whether code satisfies some non-trivial property [49]. By detecting and fixing errors, software systems can only get better. Considering this fact, verification techniques are, in general, not able to prove the correctness of code. However, applying them can still have a significant impact. The main software verification techniques include testing, theorem proving, runtime verification, abstract interpretation, type system, and model checking. In the following sections, we give a brief overview of each of these techniques.
2.1 Testing

Testing is one of the methods used to verify software systems [41]. The testing process includes providing the compiled code of the system under consideration with inputs and observing the outputs to find errors. The software should produce the outputs as expected. In other words, the outputs should be compatible with the system’s specification. Software testing is a widely used method and usually about 30% to 50% of software engineering projects costs go to the testing process [3, page 4].

One of the main advantages of testing is that it can be used to verify all kinds of software. However, to achieve full code coverage, as the size of the code increases the test cases in general quickly become very complex. Furthermore, using testing, it is nearly impossible to capture all potential executions of the code. One reason is that it is very hard to test the code against all possible input sequences. Since testers usually do not have control over the scheduling of the concurrently running components, it is very hard to use them to detect concurrency errors. Even if a bug is detected, because the execution of the code is non-deterministic, the bug is very difficult to reproduce. According to Edsger W. Dijkstra “Program testing can be used to show the presence of errors, but never to show their absence!” [15].

2.2 Theorem Proving

Another technique to verify software systems is theorem proving. This method uses axioms and inference rules to prove properties of the code. One of the main advantages of this technique is that it can be applied to code with infinite state spaces. Theorem proving requires a high level of mathematical complexity and can only be used by an expert in the formulation of formal arguments and proof techniques.

Until the 1960s, this technique was completely performed by humans. Today, some interactive software tools, called theorem provers, have been developed which can be used to develop formal proofs. Some examples of such provers can be found in [23, 46]. These tools ensure the correct applications of axioms and proof rules. At each stage of the proof, they can also suggest possible ways to make progress. It should be noted that there exist some theorem provers which are fully automatic [53, 14]. Compared with these fully automated theorem provers, in general the underlying logic of interactive theorem provers is more expressive. In general, theorem proving is a very time-consuming process and no bound can be set on the time and memory that is needed to verify code. That limits the use of this technique.

2.3 Runtime Verification

Runtime verification is also a software verification technique. This technique can be defined as a combination of testing and formal methods since it uses the functionality of both approaches. Formal methods is the application of methods from mathematics and formal logic to reason about computing systems in order to detect errors. Runtime verification tools analyze the behavior of the code at runtime. They go through a current execution of the code
to check if it is running correctly with respect to a system’s specification. Some examples of tools performing runtime verification are Monitoring and Checking (MaC) [34] and Java PathExplorer (JPAX) [27].

In the MaC framework, during the execution of the code under consideration, information about the execution is extracted and it is checked against the system’s specification. A prototype implementation of the MaC architecture has been developed for Java code, called Java-MaC [33]. The JPAX tool is very close to the MaC architecture. It extracts information about an execution. It checks the execution against the system’s specification and it also applies error detection algorithms to detect errors such as data races and deadlocks. Moreover, JPAX allows the user to define new logics.

In general, this method scales very well, i.e., using this technique, as the size of the code increases the time and memory needed to execute the code increases linearly. Moreover, runtime verification can be completely automated. However, similar to testing, using the runtime verification technique, it is nearly impossible to capture all possible execution paths.

### 2.4 Abstract Interpretation

Another technique that can be used to verify software systems is abstract interpretation. The semantics of code can be defined as a mathematical formalization of all possible behaviors of the code. Abstract interpretation requires the precise definition of a concrete semantics which represents the actual executions of the code. Since, in general, the problem of specifying whether code satisfies some non-trivial property is undecidable, this technique approximates the concrete semantics of the code to obtain an abstract semantics. More specifically, abstract interpretation can be defined as a general theory to compute the abstract semantics of code which can be further used to answer questions about properties of the code [10, 11]. The abstraction function is defined to assign each value in the concrete semantic domain to a value in the abstract semantic domain.

The goal of abstract interpretation is to obtain an abstract semantics as an approximation that gives reliable answers to questions about properties of the code, i.e., answers which are neither false positives nor false negatives. Otherwise, the analysis is not reliable. For example, consider an analyzer that uses approximations to check whether an index of an array is used out of its bounds. The analyzer computes the approximation of the set including all the indices used to access the array. One possible case is that the analyzer approximates this set by computing a subset of the indices used to access the array. If the analyzer does not find any errors and indicates that no violation is found, the answer could be a false positive, because the analyzer has not checked all the indices used in accessing the array. Another possible case is that the analyzer computes a superset of all the indices used to access the array. If the analyzer does not find any errors and gives an answer indicating that no violations are found, the answer is reliable, because the analyzer has checked all the indices used in accessing the array.

The tools based on abstract interpretation are considered as static analyzers since they determine run-time properties of the code statically, without executing it. There are many automated tools based on abstract interpretation that are used to verify code. Cibai [36] is
an example of an abstract interpretation-based tool for the verification of code written in Java. The errors that can be detected by Cibai are division by zero, array indices out of bounds, and null dereferences. Cibai can also check for user-defined assertions.

Static analyzers implementing abstract interpretation are designed based on the properties of interest. These tools accept only code as input and not the properties to be checked. They check the code against the properties which have been already hard coded into the tool [16]. Another drawback from these techniques is that creating counterexamples is difficult, i.e. precision loss resulted from approximating the system makes it hard to map the error back to the actual code.

2.5 Type Checking

Another method to verify software systems is type checking. This approach is based on formal type systems. Basically, using the type system, properties can be expressed as types and the code verification is reduced to type checking. A type system consists of a set of types and a set of rules that associate types with (parts of) code. Type checking amounts to applying the rules with the objective of showing that the code can be typed. A type system checking a particular property is designed in such a way that code satisfies the property if it can be typed.

One example of a tool including a formal type system is the race detector discussed in [21] (see Section 4 for a detailed discussion of race detection). This tool uses a type system to perform a static race detection analysis for code written in Java. The type system guarantees the absence of races in well-typed code. This race detector relies on code annotations. It is based on the lock-based synchronization discipline, i.e., each field is protected by a lock which is held while accessing the field. The race detector keeps track of the locks for each shared field and makes sure that the locks are held during every access of the field.

Another race detector based on a formal type system is presented in [8]. This race detector also relies on user annotations. The type system of this race detector is more expressive than the one in [21]. This type system allows the user to assign a different protection mechanism to different objects of the same class. This is useful since it is not always necessary to protect an object by a lock, e.g. when the object is immutable. However, in [21], the protection mechanism has to be specified at declaration time and it is applied to all instances of the class. The type system of [8] is also extended to detect deadlocks in Java code as explained in [7]. Well-typed code in this type system is free of races and deadlocks.

The design of formal type systems is specific to certain properties. This implies that to check any other properties, a new formal type system has to be designed. But type checking is considered as an efficient technique for software verification, in terms of both time and memory consumption.

2.6 Model Checking

Finally, model checking is an automated verification technique that examines all possible system states in a systematic way to check if desired properties are satisfied. In the cases
that a property is not satisfied, model checkers provide a counterexample (i.e., an execution path that leads to the erroneous state) that can be used to correct the error.

Model checking considers all possible execution paths of the code. Therefore, it can detect some errors that cannot be found using the testing and the runtime verification techniques. Moreover, since model checking is mostly automated, it is generally easier to use than theorem proving. Using model checking techniques does not require a high level of expertise and user interaction. Furthermore, model checkers, especially the ones that are directly applied to the actual system, can easily provide counterexamples which make the process of detecting errors much easier. Finally, model checkers are not specific to certain properties, i.e., the input to model checkers is the code (or the system model) and the properties to be verified.

However, model checking is subjected to the state space explosion problem. This is considered as one of the most challenging issues in model checking. The state space explosion problem occurs when the state space of the code becomes too large and available memory resources are not enough to store it (see Section 3.2). In this case the model checking process cannot be completed and terminates by running out of memory. In general, as the size of the code increases, the time and memory needed to model check the code increases exponentially.

Although model checking is subject to the state explosion problem, considering its advantages, in many cases, it can be preferable to the other methods. It should be noted that the comparison mentioned earlier between model checking and other techniques is based on their basic definitions. In practice, these techniques can be closely related. For example, in some cases, model checking can be viewed as theorem proving [24]. Moreover, software verification tools may not fall exactly within one of these techniques. For example the systematic testers such as ExitBlock [32] and CHESS [39] are designed to test concurrent code. The technique implemented by these tools is very close to model checking. They have control over the scheduling of the concurrently running components. Given a test scenario, these tools are able to repeatedly execute the code so that each execution has a distinct scheduling.

The increasing interest in using model checking tools in industry demonstrates the preference of many companies to use model checking techniques. Many leading companies have started research on model checking and have developed their own model checkers, e.g., SLAM and ZING developed at Microsoft [4, 2], Spin developed at Bell Labs [30], and RuleBase developed at IBM [5]. There are also many examples where model checking has been successfully applied in industry and it has detected previously unknown errors [26].

3 Overview of Model Checking

Based on the way that states are represented, model checking algorithms can be classified into two main categories: explicit-state model checking versus symbolic model checking. In explicit-state model checking, graph algorithms are used to create and explore the state space. While exploring the graph, the states are checked against the desired properties. To avoid regenerating states, the algorithm should keep a record of visited states which are usually stored in a hash table. Spin [30] and Java PathFinder (JPF) [55] are examples of
explicit-state model checkers.

Symbolic model checking deals with sets of states instead of directly dealing with states. In symbolic model checking, states are usually represented using binary decision diagrams (BDDs) [1]. A BDD is a data structure for boolean expressions. It is a rooted, directed, acyclic graph that can be retrieved from a decision tree by identifying all the identical nodes. Due to BDD’s features, there are efficient algorithms that can perform logical operations on them. One issue with using BDDs is choosing a suitable ordering of variables, which is tedious and affects the size of the BDD. Another way for symbolic representation of states is propositional logic formulas [6]. Microsoft’s SLAM project [4] and SMV [38] are two examples of symbolic model checkers.

Symbolic model checking is mostly applied on hardware rather than software. Because symbolic model checking deals well with static transition relations and it is not suitable for systems including dynamic creation of threads and objects. Compared to explicit-state model checking, it can verify larger systems. Unlike explicit-state model checking, symbolic model checking can handle systems with infinite state spaces. Symbolic methods are more suitable for systems subjected to data non-determinism. Explicit-state model checking works better in finding concurrency errors [17]. It should be noted that the remainder of this paper is restricted to explicit-state model checking. From here on, we use model checking to refer to explicit-state model checking.

Another criterion used to categorize model checkers is based on their input, which could be either a model of the system or the system itself. The first group includes model checkers that are applied on models that describe the possible behavior of the systems. The first step to use these model checkers is to create a model of the system using a modeling language understandable by the model checker. However, the verification results are just as good as the model of the system [3, chapter 1]. Due to the loss of precision, the model of the system might not reflect the exact behavior of the actual system. These types of model checkers can be used to verify the design of the system but not its implementation. Another drawback of using these model checkers is that the input languages for these model checkers are modeling languages which are usually too simple to capture all the features of the system. Spin [30], SMV [38], and SAL [45] are examples of model checkers applied on a model of the system.

The second group of model checkers are directly applied to the actual system. Therefore, using this group of model checkers does not require creating a model of the system. That makes this group of model checkers much easier to use since they do not require the user to have comprehensive understanding of the code. These model checkers are able to verify both the design and the implementation of the system. By getting rid of the modeling process, the drawbacks from the former group of model checkers are not applicable to these model checkers. Some examples from this group are JPF [55] for Java code, VeriSoft [22] for C code, and CMC [40] for code in C and C++.

3.1 The Model Checking Process

The model checking process can be divided into four major steps explained below. Every model checker goes through some or all of these steps.
Modeling - modeling is the very first step of the model checking process. This step is performed by model checkers that are applied to a system model. In this step, the model of the system is generated from the system’s specification using a model description language understandable by the model checker. The system is usually modeled using a finite-state automaton. For example, consider Java code. States of the Java code may contain the values of variables in the system and snapshots of the stacks and the heap. Transitions take the system from one state to another. In the Java code, transitions may be represented by bytecode instructions. The model of the system should be validated to make sure that it specifies the behavior of the code accurately. Accurate modeling of the code can result in detecting mistakes such as incompleteness and ambiguities in the system’s specification [3, chapter 1].

There exist some model extractors that have been developed to automate the modeling process. They are usually combined with model checkers that accept a model of the system to make the whole model checking process automated. Basically, these tools receive the source code as input and extract a finite model of the system. The finite model is in the input language of an existing model checker. For example, Bandera [9] is applied to Java code and extracts a finite model of the code in three different modeling languages, namely the input languages for Spin, SMV, and SAL.

Properties Specification - in this step, the desired properties of the system under consideration are specified. This step precisely outlines what the code is supposed to do, and what it is not supposed to do. One usual way to state the properties at this stage is to specify them using temporal logics. Temporal logics are considered to be a useful way to formalize properties of concurrent code since they can be used to describe the behavior of code over time.

Running - in this step the model checker is initialized and, depending on the type of model checker, it is run on either the model or the actual system. The model checker algorithm considers all possible interleavings of the code and at each state it checks if the properties that are specified in the previous step hold. Usually when a property does not hold the model checker stops running and terminates to report the error. When the size of the state space gets too large, due to the state space explosion problem, the running process terminates by giving an out of memory error.

Analysis - after the running process terminates, the model checker provides the user with different results. Different scenarios can be reported: the code satisfies the desired properties, the code does not satisfy a property, or the model checker runs out of memory and terminates due to the state space explosion problem. If the results demonstrate that the property is not valid, an error is detected. The error can have different sources: a modeling error (for the group of model checkers which are applied on the system model), an error in the actual system, or an error in the property specification. The model checker
usually provides a counterexample that helps the user detect the source of the error. A counterexample is usually a trace of the code that leads to the error. For the group of model checkers that are applied on the source code, the trace of the error is directly mapped to the source code. However, traces produced by the model-based model checkers are not directly mapped to a trace in the actual system. There are some model extractors embedded into these model checkers that exploit intermediate languages to map the model to the source code, e.g., Bandera [9].

3.2 State Space Explosion Problem

The most challenging issue in model checking is dealing with the state space explosion problem. This problem is due to non-determinism. Non-determinism can be either thread non-determinism or data non-determinism. For code subject to thread non-determinism, the concurrent actions can be executed in any arbitrary order. Considering all possible interleavings of these concurrent actions usually leads to a very large state space. It can be shown that the number of states can increase exponentially with the number of interleaving components [3, chapter 2]. Consider $N$ components where the size of the state space for each of them is $k$. The maximum state space size for the system including these components would be $k^N$. For systems subject to data non-determinism, the domain of the variables influences the state space size. For data structures that can assume many different values, different choices may lead to different execution paths.

One of the common approaches used to deal with the state space explosion problem is partial order reduction (POR). The aim of the POR technique is to reduce the number of possible orderings to be analyzed by the model checker. This technique is based on identifying the execution path fragments, that lead to the same state of the system regardless of the order of concurrently executed actions. These actions are called independent actions. For example, actions accessing disjoint variables or accessing local variables of different processes, are considered as independent actions. However, two threads writing to the same variable or acquiring the same lock are considered as dependent actions. The sets of independent actions occurring in any order demonstrate the same behavior of the system. Consequently, instead of analyzing all possible orderings of this set of actions, only one or a few of them are analyzed as representatives.

This approach results in a state space which is a subset of the original state space. The resulting state space should be equivalent to the original state space with respect to the property of interest. The POR reduction technique can be applied either dynamically or statically [3, chapter 8]. The dynamic POR reduction is applied on-the-fly, that is, the reduced state space is generated during the model checking. However, in the static POR reduction, the reduced state space is generated before the model checking process starts. POR reduction techniques can have a significant impact on the size of the state space and they are used by almost every existing model checker.
3.3 Why Model Checking Java Code?

In this paper, we mainly focus on the model checking of code written in Java, since we believe that Java has several advantages that makes it very attractive and distinguished from other programming languages. In general, Java is a simple programming language. Using a simple programming language, programmers are less likely to make errors. Java syntax is easy to understand. By providing garbage collection and automatic memory allocation functionalities, Java does not require the programmer to get involved with memory management, unlike C and C++. That can make the programmer’s task much easier.

Moreover, Java is a safe language. Unlike C and C++, Java does not define pointers. In Java memory locations cannot be accessed by the Java code directly. Java also has a strong typing feature. Types of all variables have to be known at compilation time, which allows the compiler to detect errors at compile time.

Code is usually restricted to a particular hardware and operating system. But Java is platform-independent, that is, a single version of the Java code can run on any platform. Java provides this feature by compiling Java code into bytecode instead of compiling it into machine language like a C compiler. Bytecode is the same for all platforms and in order to be executed it needs a Java interpreter which translates bytecode into the executable code of the host machine. By providing a Java bytecode interpreter for a specific platform, any compiled Java code can run on that platform.

Finally, Java provides support for multithreading. Object oriented languages like C++ do not provide built-in support for multithreading features. However, they can be used to developed multithreaded code by allowing programmers to use the multithreading features of the operating system directly. That makes the concurrent programming in these languages more complicated than it is in Java.

These are some of the features that make Java a widely used programming language and a top choice for many developers. That can be seen from the graph of Figure 2. This graph shows the popularity of programming languages and it was released by TIOBE Programming Community Index in November 2010.

3.4 Tools for Model Checking Java Code

One of the most well-known software model checkers is Spin, which was developed by Holzmann [29]. Spin is an explicit-state model checker that can be used to verify the models of concurrent code. Its development is started in 1980 at Bell Labs. The input language of Spin is Promela, which is a high-level language to formalize the system model. Compared to other modeling languages, Promela has a rich modeling functionality. Spin relies on the fact that the behavior of asynchronous processes in distributed systems can be modelled by finite state automata. Concurrent code in Promela is described by a parallel composition of process templates which specify the behavior of the processes. Spin generates a finite automaton for each process template. Then it computes an asynchronous interleaving product of these automata to create the state space of the system.

6http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
To express the behavior that should never happen, Spin provides the *never* claim. Spin generates a Büchi automaton from the *never* claim. It computes the synchronous product of this automaton and the one representing the state space. The system does not violate the property if the language accepted by the resulting automaton is empty. Instead of hand-writing the *never* claim, the user can simply specify the undesirable behavior by a linear temporal formula which is translated to a *never* claim. Spin allows for dynamic allocation of processes. But it does not support dynamic allocation of data. An extension of the Spin model checker called dSPIN [13] has been developed. It provides support for dynamic memory allocation. However, both Spin and dSPIN impose a limit on the size of the state.

Spin was originally developed for verification of communication protocols. Today, Spin is considered a mature tool and it is used for a wide variety of software systems. It also serves as a back-end for other tools like an early version of Java PathFinder (JPF1), Bandera [9], and JCAT [12].

As mentioned earlier, several tools have been developed to automatically specify the system in the modeling language of an existing model checker. These tools are applied to avoid modeling the system manually. The first generation of Java PathFinder, JPF1, which was developed at NASA in 1999 is among these tools [25]. JPF1 receives the Java source code as input and automatically translates it into the Promela language. The motivation behind this project came from applying the Spin model checker on a multithreaded operating system written in LISP for NASA’s Deep Space 1 mission. They manually translated the LISP code into Promela and applied Spin, which led to the discovery of five previously

![Figure 2: TIOBE Programming Community Index for November 2010](image)
unknown concurrency errors [26]. Some of the features of Java that cannot be translated using JPF1 include packages, method overloading and overriding, recursive methods, strings, and floating point numbers. Moreover, JPF1 does not map error traces produced by Spin back to their corresponding traces in the Java source code.

JCAT [12] also receives the Java source code as input and translates it into the input language of Spin. JCAT can be used to detect deadlocks in multithreaded Java code. Using the Java2Spin translator tool, JCAT creates an abstract formal model from the Java code. Spin verifies the model and if any error is detected, JCAT maps the model trace that leads to the error to a sequence of states in the actual Java code. One of the limitations of the tool is that it cannot support polymorphism. In order to reduce the size of the state space, the Java2Spin translator applies static analysis on the Java code. It gets rid of the inherited class members that are not reachable and the object synchronization structures that are not used by any of the threads. JCAT also combines some state sequences into a compound state in a way that does not influence the system properties. Further, in JCAT, the user can use annotations to remove the variables that do not affect the concurrent behavior of the code.

Another tool that converts Java code into the Promela modeling language is Bandera [9]. Bandera automatically extracts a finite-state model from the source code and creates a model in the input language of one the supported verification tools, which are Spin, SMV, and SAL. Bandera is based on the Soot compiler [54], a framework for optimizing Java bytecode. To optimize Java bytecode, Soot transforms bytecode to multiple intermediate languages. One of these intermediate languages is Jimple. In Bandera, the Java code is translated to Jimple. In order to simplify the code and to reduce the state space of the model, Bandera applies three main techniques on the Jimple representation of the code. The first one is a slicing technique that removes from the code the statements that are irrelevant for checking the desired property. Another reduction technique used by Bandera is a data abstraction technique that reduces the unnecessary details associated with variables. The last technique is component restriction, which decreases the number of components involved in the code or limits the ranges of variables. Applying the component restriction technique, the model does not exhibit all the behavior of the code. After applying the techniques described above, Bandera generates the Bandera Intermediate Representation (BIR), which is a low-level intermediate presentation of guarded commands, from the reduced code. Then the finite-model of the code is created in the input language of a verification tool chosen by the user. After running the chosen verification tool, Bandera interprets the output of the tool. If the model does not satisfy a desired property, Bandera maps the trace back to the Java source code.

Another group of tools to verify Java code consists of model checkers that are based on executing the actual system. These tools use schedulers to execute the code in all possible ways. One of the model checkers of this type is the explicit-state SAL (Symbolic Analysis Laboratory) model checker [45]. Basically, SAL is a framework to merge different tools for model checking, abstraction, theorem proving, etc. SAL uses an intermediate language for specifying concurrent code. The SAL model checker is written in C++. It receives the model
in the SAL intermediate language and checks for deadlocks and assertion failures. It starts from the initial state and uses either breadth-first or depth-first search to explore the state space. It stores new states into a hash table to avoid recreation of states and to guarantee the termination of the model checking process. Java code can be represented in the SAL input language which also supports dynamic data structures. The SAL model checker has been adapted to work with Java code [45]. The input can be either Java source code or Java bytecode which is translated in the SAL intermediate language. Like in Bandera, in SAL, Java code is translated in the Jimple intermediate language. The idea of the SAL model checker is based on the VeriSoft [22] model checker. VeriSoft is a model checker for concurrent C code which uses a scheduler to systematically execute the code in all possible ways. However, since VeriSoft does not store the visited states, termination of the model checking process is not guaranteed. In order to terminate, VeriSoft imposes a limit on the depth of the search.

Bogor is a software model checking framework that can also be adapted to verify Java code [50]. Bogor analyses code that is specified in an extended version of BIR. Bogor also uses a scheduler to systematically execute the code in all possible ways. The three main modules included in Bogor are search, the scheduler, and the state manager module which are used to explore the state space. The goal of the Bogor project is to provide flexibility in the choice of input language, search algorithm, reduction techniques, and state-space representation. Since the Bogor framework is not tied to a particular setting, it can be adopted to efficiently model check any code.

The second generation of Java PathFinder, JPF, also relies on executing Java code. As was mentioned earlier, the first generation of JPF was simply a translator from Java to Promela. But this version of JPF worked on the source code of the Java application, which is not always available. Moreover, some features of the Java language cannot be easily represented in Promela, e.g. Promela does not support floating point numbers. Therefore, in 2000, JPF was refactored as a Java virtual machine (JVM) which can execute all Java bytecode instructions generated by a Java compiler.

JPF is an explicit state model checker that works directly with Java bytecode. JPF provides support for dynamic features, such as class loading, dynamic data structures, and method invocation. But, unlike Spin and dSPIN, it does not impose any limit on the size of the state. Compared to the other model checking tools, JPF is very flexible. It can be configured to use different functionalities, like different search algorithms and reduction techniques to reduce the state space. One of the very attractive features of JPF is that it can easily be extended. Furthermore, JPF is applied directly on the code and does not require the modeling process. In 2005, it became an open-source project on the source code repository SourceForge.net. In 2009, the JPF project was moved to the NASA server, Mercurial. Today, JPF is a mature tool with hundreds of active users and more than one hundred downloads every month. In Section 5, the structure of JPF is explained in more detail.
4 Race Detection

A (data) race is one of the notorious concurrency errors. Roughly speaking, a race on a shared variable arises in concurrent code if two threads access that variable simultaneously and the accesses are conflicting, that is, at least one of them writes to the variable. Although some races are benign, races are often representing errors in code. Hence, tools that detect them are invaluable to those writing concurrent code.

For an example of a race consider the following Java code.

```java
public class Account
{
    private int balance;

    public Account()
    {
        this.balance = 0;
    }

    public void deposit(int amount)
    {
        this.balance += amount;
    }
}

public class Deposit extends Thread
{
    private Account account;
    private int amount;

    public Deposit(Account account, int amount)
    {
        this.account = account;
        this.amount = amount;
    }

    public void run()
    {
        this.account.deposit(this.amount);
    }
}

public class Bank
{
    public static void main(String[] args)
```
This code consists of the three classes `Account`, `Deposit`, and `Bank`. These classes represent a bank with one account, with an initial balance of zero. In the `main` method of the `Bank` class a new account is created and two threads are depositing money into the account. By executing `new Deposit(account, 100).start()`, a new `Deposit` object is created, which is a thread, and its `run` method is invoked. The execution of the `run` method should amount to depositing 100 into the `account`. Similarly, by executing `new Deposit(account, 200).start()`, another thread deposits 200 into the `account`.

Now consider the scenario where one of the threads first reads the value of the variable `balance` and the other thread then reads the same value of the variable `balance`. In this scenario, the two invocations of the `deposit` method “overlap” and the final value of `balance` may be equal to 100 or 200 (depending which thread changes the value of `balance` last), rather than 300. This is caused by a race on the variable `balance`. This example demonstrates a race that makes the code behave in an unexpected way.

Many tools have been developed to detect races. These tools are based on the two types of race detection techniques: dynamic and static. In dynamic race detection, a single execution of concurrent code is checked for races. Dynamic race detection is NP-hard (see [43] and the references therein). The two key approaches to detect races dynamically are based on locksets and the happens-before relation. The former approach has been popularized by the Eraser tool [51]. During the execution, Eraser dynamically keeps track of the set of locks that have protected shared variables so far. Each time a thread accesses a shared variable, Eraser computes the intersection of the set of locks protecting the variable and the locks held by the thread. A race warning is issued if the intersection becomes empty. Another example of a dynamic race detector based on the lockset algorithm is Goldilocks [18].

The happens-before relation is a relation between the actions of the code. In order to define this relation, first we define the program-order and the synchronizes-with order. Consider multithreaded Java code. The actions are represented by bytecode instructions. The actions within a thread occur in the same order in which they show up in the code. This ordering of actions is called the program-order. Any action that releases a lock on an object synchronizes-with all subsequent actions that acquire the lock on the same object. Moreover, writing to a volatile variable synchronizes-with all subsequent reading of the same variable. The happens-before relation can be defined as the transitive closure of the union of the synchronizes-with order and the program-order [37]. For an example of a race detector based on happens-before relation we refer the reader to [52].

In static race detection, all potential executions are considered. Although this approach
gives rise to tools that are usually sound (that is, the races that are reported by the tool are real races), the tools are generally not complete (that is, not all races are always reported). Examples of static race detectors can be found in [19, 20, 42, 48]. Several different approaches exist to statically detect races. One of these approaches is model checking. For example, in [28] model checking is exploited to detect races in code written in an extension of C. The JPF model checker can be also used to detect races in Java code. JPF includes two different race detectors, which are discussed in Section 5.3.

5 Java PathFinder

In this section, we focus on the design and the structure of JPF. We explain its main components and also some of its interesting functionalities such as its extension mechanism, race detection, the way that it handles native calls, and some of its main reduction techniques.

5.1 Main Components of JPF

The structure of JPF contains two main components: JVM and Search. The JVM component of JPF executes the target application and during execution, it creates the state space of the code. The search component works as a driver for the JVM. It is responsible for making the JVM move along certain paths within the state space. The JVM and the Search components are explained in more detail in Section 5.1.1 and 5.1.2, respectively.

5.1.1 JVM Component

Since the core of the JPF model checker is a Java virtual machine (JVM), JPF can be considered as a JVM that executes Java applications. JPF is able to execute all of the bytecode instructions that are created by a Java compiler. JPF itself is written in the Java programming language. That means that JPF is running on top of another JVM which we call the underlying host JVM. Figure 3 demonstrates the different layers that are involved in model checking a target application using JPF.

![Diagram](image.png)

Figure 3: Different layers involved in running JPF on a target application
JPF is a special JVM. It explores all potential executions in a systematic way. Each execution is a sequence of transitions and each transition is a sequence of bytecode instructions that takes the system from one state to another. In contrast, an ordinary JVM executes the code in only one possible way. Moreover, as JPF executes the target application, it checks for certain properties. Some of the properties checked by JPF are unhandled exceptions, deadlocks, and user-defined assertions which are used to test the assumptions about the code’s behavior. While executing the code, JPF also generates the state space. Each state contains a snapshot of the current execution status of the code, the path that leads to this state, and possible choices in the current state where each choice value is associated with a transition. In order to represent the possible choices, JPF defines the ChoiceGenerator class. Every ChoiceGenerator object can capture either data nondeterminism or thread nondeterminism. The former is caused by Java constructs such as Math.random(), which returns random real numbers in the interval [0,1]. The latter is related to the scheduling of concurrent threads.

5.1.2 Search Component

As JPF explores the state space, in order to move from one state to another, it has to choose from the transitions leading out of the state. The way that JPF chooses the next transition is determined by its search component. The Search component can be configured to use different algorithms to traverse the state space, such as depth-first search (DFS) and breadth-first search (BFS). The Search component of JPF is captured by the class Search. This class contains an abstract method called search. Java classes such as DFSearch and BFSHeuristic implement different search algorithms by extending the Search class and overriding its search method. Each Search object has a JVM object which represents the JVM component. This allows the Search object to invoke methods on the JVM object, such as forward, backtrack, and restoreState. To move from the current state to a new state, it invokes the forward method on the JVM object. To backtrack, it invokes the backtrack method on the JVM object. To move to an arbitrary state, it invokes the restoreState method on the JVM object. The UML diagram in Figure 4 depicts the relationship between the classes Search and JVM.

5.2 Listeners

As was mentioned earlier, JPF has been developed in such a way that it can be easily extended. Listeners are considered the most important extension mechanism of JPF. Listeners interact with the JPF components, JVM and Search. They are used to retrieve useful information about the execution of JPF while it model checks the target application. As is shown in Figure 5, listeners are running at the same level as JPF, on top of the host JVM. They register themselves with the Search and/or JVM components in order to receive notifications on the occurrence of certain events. When any of those events occur, the component notifies the registered listener and it invokes a method of the listener which corresponds to the event. The notifications can be issued on a wide variety of events, from low level events
Figure 4: UML diagram representing the relationships between some of the main classes of JPF.

Figure 5: Listeners and JPF running on top of the underlying host JVM

like instructionExecuted, indicating that the next instruction was executed, to high level events like searchFinished, indicating that JPF is done exploring the state space. The JPF implementation includes two basic interfaces for listeners which are VMListeners and SearchListeners. Listeners are represented by classes that implement one or both of these interfaces. Listeners that implement the interface VMListeners register themselves with the JVM component. Similarly, listeners that implement the interface SearchListeners register themselves with the Search component. The race detectors of JPF are examples of listeners included in the implementation of JPF (see Section 4 and 5.3).

5.3 Race Detectors of JPF

As was mentioned earlier, JPF can be exploited to detect races in Java code. JPF has two different race detectors which are added to JPF in the form of listeners. Both of the race detectors are based on static race detection.
The lockset algorithm and its numerous variations are usually exploited for dynamic race detection. However, this algorithm has also been used for static race detection. The original race detector of JPF also implements a variation on the lockset algorithm to detect races statically. The other race detector of JPF is called the precise race detector. The idea behind this race detector is fairly simple. In every state that JPF visits, it checks all actions that can be performed next. If this collection of actions contains at least two conflicting accesses of a shared variable \( v \), then a race on \( v \) is reported. A similar approach in a considerably simpler setting has been proposed in [47].

The advantage of the precise race detector is that it is sound, that is, the races that are reported by the tool are real races. JPF’s original race detector is, however, not sound. The key idea behind the lockset algorithm, on which the original race detector is based, is the following implication. If for a shared variable \( v \) there exists at least one lock \( \ell \) such that \( \ell \) is held during all conflicting accesses of \( v \) in an execution, then the execution is free of races on \( v \). However, locking is not the only programming idiom that can be used to prevent races. Instead, for example, semaphores can be exploited. Assume that a semaphore and a variable are shared by two threads and the semaphore is initialized to 1. Assume also that both threads execute the following code snippet.

```java
semaphore.acquire();
variable++;
semaphore.release();
```

In this case, no locks are held when the variable is accessed. As a consequence, JPF’s original race detector reports a potential race on variable. This is however not a real race, as is confirmed by the precise race detector which does not report a race.

JPF has a number of search algorithms to traverse the state space such as depth-first search (DFS) and breadth-first search (BFS). The user can configure JPF to use any of these algorithms. But JPF’s original race detector has been built on the assumption that JPF uses DFS. Assume that two threads share a variable and that both threads increment the variable. If JPF uses DFS to traverse the state space, the original race detector reports a race on variable. However, if JPF uses BFS instead, then the original race detector does not report the race. Also for other traversal algorithms, the implementation of the original race detector needs to be changed. In contrast, the precise race detector is independent of the search algorithm of JPF and, hence, can be used with any search algorithm without any modification.

JPF’s original race detector only considers two types of shared variables: static fields and non-static fields. It is not able to detect races on array cells. For example, assume that an array \( a \) is shared by two threads and both threads increment \( a[0] \). In that case, there is a race on the array cell \( a[0] \). JPF’s original race detector does not report this race, since it simply does not consider array cells. The precise race detector can handle array cells.\(^8\)

JPF’s original race detector does not detect all races on static fields. Consider, for example, the classes depicted in the UML diagram in Figure 6. Assume that there are two

\(^8\)The version of the precise race detector included in the current JPF distribution does not handle array
threads. One of them increments \( C_1.x \) and the other increments \( C_2.x \). Since \( x \) is a static field of \( C \), \( C_1.x \) and \( C_2.x \) refer to the same field. Hence, there is a race on \( x \). However, JPF’s original race detector considers \( C_1.x \) and \( C_2.x \) as two different fields and, hence, does not detect this race. The precise race detector does report a race on \( x \).

To evaluate the performance of the precise race detector, we ran JPF on concurrent Java code. For each program, JPF was run a hundred times with three different settings. Table 1 shows the average running time in milliseconds. The first row contains the number of lines of code for each program. The second row is obtained using the default setting of JPF. The third and fourth row report the results with the old race detector and the new one enabled, respectively. The fifth row compares the overhead of the old and the new race detector, where

\[
\text{overhead ratio} = \frac{\text{old} - \text{default}}{\text{new} - \text{default}}.
\]

It can be seen that the overhead of both race detectors is very small. However, the precise race detector is more efficient than JPF’s original race detector.

### 5.4 Handling Native Calls

A method is called native if it is implemented in a language other than Java but it is invoked from a Java application. Many of the classes of the Java standard library include invocations of native methods. Therefore, it is essential for Java model checkers to provide a way to handle native calls. The JPF model checker provides different ways to handle native calls.
methods. But before explaining how JPF handles native calls, we describe how the ordinary Java virtual machine handles native methods.

5.4.1 Java Native Interface

Every JVM includes a Java native interface (JNI) that allows Java applications to call or to be called by functions which are written in other languages such as C, C++, and assembly. Basically, JNI is used to transfer the execution from the Java level to the native level. That can be seen in the following figure. Whenever the JVM comes across a bytecode instruction that invokes a method defined as native, the execution is transferred to the native level. After the native method returns, the execution is transferred back to the JVM. JNI allows us to use functions that have already been implemented in other languages. Moreover, in some cases, accessing code written in, for example, C and C++ from applications written in Java can improve the performance. Furthermore, JNI can be used when Java does not support certain platform-dependent features.

5.4.2 Model Java Interface

One of the most important features of JPF is its model Java interface (MJI). In analogy to JNI, which is used to transfer the execution from the Java level to the native level, MJI is used to transfer the execution from the JPF level to the underlying JVM level. The classes called native peers play a key role in the implementation of MJI. Native peers run on top of the underlying host JVM. Therefore, these classes are completely unknown to JPF and are not model checked by JPF at all. Later in Section 5.4.4, it is explained that how does JPF exploit MJI to handle native calls in Java programs. Figure 8 demonstrates the role of MJI.
JPF uses a specific name pattern to establish the correspondence between a native peer class and a class of the target application. It also relates the methods of these corresponding classes. For example, Figure 9 shows how the correspondence is established between the java.lang.StrictMath class and its native peer. The class on the left is part of the Java standard library and the one on the right is its native peer. When JPF gets to the bytecode invoking the method sin of java.lang.StrictMath, since sin is linked to a method in the corresponding native peer, it does not model check this method. Instead, the execution is transferred to the underlying JVM, and the host JVM executes the method sin of the class JPF.java.lang.StrictMath.

Since there is no compatibility between JPF objects and the underlying host JVM objects, all native methods in native peer classes are defined as static. Moreover, MJI provides native methods with at least two arguments. The first argument of native methods is of type MJIEnv. MJIEnv is an interface that allows native peers to access all internal features of JPF. The second argument of native methods is an integer representing the JPF object or the JPF class that has invoked the native method (i.e., in JPF, objects and classes are represented by a unique integer).

5.4.3 Model Classes

JPF has special classes called model classes. These classes can be considered as part of the target application, and they are model checked by JPF. The model classes are completely unknown to the underlying JVM. These classes are used as a replacement of Java classes.

Before executing the target application, JPF starts loading classes. To load classes, JPF starts from the directories that include model classes. Once a class is loaded, JPF is not going to load the other versions of the same class. By implementing model classes, we force JPF to use an alternative version of certain Java classes. Therefore, by defining model classes we can make JPF not model check certain classes, and instead use the model classes as alternatives.
5.4.4 The Ways that JPF Handles Native Calls

Recall that the java.lang.StrictMath class includes the sin method, which is defined as native. Since JPF only executes Java bytecode, running JPF on the following example crashes as soon as JPF hits this native call.

```java
public class Operation {
    public static void main(String[] args) {
        System.out.println(StrictMath.sin(0.3));
    }
}
```

JPF provides three different ways to handle native calls. Using this simple example, these methods are described below.

*Using only a model class* - The first approach to handle the sin native call is to create a model class java.lang.StrictMath that implements the sin method in pure Java. For example, using the following model class, JPF can run on the previous example without crashing.

```java
package java.lang;

public class StrictMath {
    public static double sin(double d) {
        return (d - (d*d*d/6) + (d*d*d*d*d/120) - (d*d*d*d*d*d*d/5040));
    }
}
```

Since there is a model class defined for java.lang.StrictMath, JPF never loads and model checks the standard class java.lang.StrictMath. Instead, it loads the StrictMath model class. Therefore, in the main method of Operation, JPF model checks the sin method of the StrictMath model class, not the sin method of the Java standard class java.lang.StrictMath.

*Using only a native peer class* - The second approach to handle the invocation of the sin native method of the StrictMath class is sending the execution to the underlying JVM level by using a native peer class. To achieve this, the following native peer class is created.

```java
package gov.nasa.jpf.jvm;

public class JPF_java_lang_StrictMath {
    ...
}
```
public double sin__D__D(MJIEnv env, int cref, double d) {
    return StrictMath.sin(d);
}

At load time of the class `java.lang.StrictMath`, using the specific name pattern, JPF maps this class to the native peer class `JPF_java_lang_StrictMath`. It also maps the `sin` method of the class `java.lang.StrictMath` to the `sin` method of the class `JPF_java_lang_StrictMath`. Furthermore, the method `sin` of `java.lang.StrictMath` is marked as native. When JPF gets to `StrictMath.sin(0.3)` in the `main` method of the `Operation` class, the execution is sent to the underlying host JVM which executes the method `sin` of the class `JPF_java_lang_StrictMath`. Since this method is executed by the underlying host JVM, it is not model checked.

Using both a model class and a native peer class - The third way to handle the `sin` native method is sending the execution to the JVM level by using both a model class and a native peer class. To achieve this, we use the following model class `StrictMath` and the native peer class `JPF_java_lang_StrictMath` explained above.

```java
package java.lang;

public class StrictMath {
    public static native double sin(double d);
}
```

At load time of the model class `StrictMath`, using the specific name pattern, JPF maps this class to the corresponding native peer class `JPF_java_lang_StrictMath`. It also maps the `sin` method of the class `java.lang.StrictMath` to the `sin` method of the model class `JPF_java_lang_StrictMath`. Also, the method `sin` of the model class is marked as native. When JPF gets to `StrictMath.sin(0.3)` in the `main` method of the `Operation` class, the execution is sent to the underlying host JVM which executes the method `sin` of the class `JPF_java_lang_StrictMath`.

Comparing the second and third approach reveals that as soon as the `StrictMath` model class is loaded, JPF treats it exactly the same way that it treats the `StrictMath` class from the Java standard library. In general, the first approach is suitable for the cases that the object/class invoking the method is either stateless (i.e., does not have any fields) or its state does not change during the execution. The second approach is suitable for the cases that the state of the object/class invoking the method changes during the execution. The third approach which uses both a model class and a native peer class is suitable for the cases where the class/object invoking the method contains some state, and some of the native calls change the state and some of them do not.
5.5 Reduction Techniques

In this section, we explain some of the techniques implemented in JPF to overcome the state space explosion problem.

5.5.1 Partial Order Reduction

JPF uses a POR technique to cut down the state space. The POR of JPF is applied on-the-fly. By putting the POR in effect, JPF combines a sequence of bytecode instructions in a thread, that do not have any effects outside the thread, into a single transition. Basically, the POR technique of JPF, for certain bytecode instructions, goes through some tests to decide if the instruction has any effects outside the current thread. Only if so, the transition is broken. 

To break the transition, JPF creates a new ChoiceGenerator object which captures the next possible choices from the current state. JPF performs the tests whenever it reaches a bytecode instruction that accesses a shared variable. There are four bytecode instructions that can access a shared variable, getfield and putfield for reading and writing to non-static fields, and getstatic and putstatic for reading and writing to static fields.

Next, we present just a few examples of the POR tests performed by JPF. One checks if the current thread is the only running thread in the system. If so, JPF does not break the transition. Another checks if the field being accessed is defined as a final field (final fields cannot change value once they are initialized). In that case, the transition is not broken. Yet another checks, if the field being accessed by the current thread is assumed to be protected by a lock. Also in that case, the transition is not broken.

In general, the POR technique of JPF proves to be very effective. To evaluate the effect of the POR tests on the size of the state space, we ran JPF on some concurrent Java code using two different configurations of the POR. In the first one, the POR is completely enabled. In the second configuration, the POR is almost completely disabled. That is accomplished by excluding the majority of the POR tests. The bar graph in Figure 10 compares the sizes of the state spaces obtained from the different settings of the POR for each run. It shows that almost completely disabling the POR increases the size of the state space considerably.

5.5.2 Garbage Collection

In Java, objects are created using the new statement. For each new object, JPF allocates memory in the heap. The heap is used to store all objects created during the execution of Java code. Java does not provide any explicit statement to deallocate the memory for an object that is no longer referenced by the code. We refer to such an object as garbage. Java is provided with a garbage collection mechanism which automatically removes all garbage from the heap. To model check Java code, it is necessary to implement the garbage collection mechanism. For example, consider the following code snippet [31].

```java
while (true)
{
    new C();
```
If JPF does not provide a garbage collection mechanism, model checking this example leads to an infinite number of states. The garbage collection algorithm implemented in JPF is called mark and sweep [31]. In the marking phase, JPF marks all the objects that can be referenced by the code. Next, in the sweeping phase, it checks all the objects stored in an object that represents the Java heap, and it removes all the objects that have not been marked in the previous phase.

The object that represents the Java heap is an instance of the class `DynamicArea`. JPF uses an object of type `StaticArea` to store references to static objects. Like an actual Java virtual machine, JPF associates each thread with a stack of the frames, which correspond to the method invocations.

The marking process is recursive. In this process, JPF uses the stack of every thread and marks all the local variables and operands in the operand stacks associated with each stack frame. It also uses the `StaticArea` object to mark all the static objects. In the sweeping phase, JPF uses the `DynamicArea` object and checks all the objects stored in the dynamic area. It removes the ones that have not been marked in the marking phase.
5.5.3 State Compression

Another issue in explicit-state model checking is dealing with the complexity of the states. To avoid analyzing the states that have been visited before, explicit-state model checkers need to store visited states. Due to the features that the Java programming language provides, capturing the state of the code in its original form needs a complex data structure. To address this problem, JPF extends the collapse method which was originally used in the Spin model checker [29]. The idea behind this method is that the transition that takes the system from the current state to a new one may change only a small part of the state.

In the collapse method of JPF, the information about the state of the Java code is divided into the following different pieces:

1. Values of static and non-static fields,
2. Monitors associated with classes and objects,
3. Stack frames, and
4. Other thread information.

JPF uses different types of components to store different pieces of information about the state. All the components of the same type are stored in the same data pool. The components within a pool are unique and each of them is associated with a unique index which indicates the location at which the component is located in the pool. To store a new state, for each component, JPF checks the content of the corresponding pool to make sure that the component has not been created before. If so, it adds the new component to the pool.

6 Summary

This paper discusses different software verification techniques that can be used to verify concurrent code. It presents the advantages and disadvantages of these techniques and it mainly focuses on the model checking technique. Furthermore, the structure of JPF, which is an explicit state model checker, is described. JPF works with Java bytecode and it checks for properties such as uncaught exceptions and deadlocks. JPF is provided with some interesting features. One of the very important features of JPF is that it can easily be extended. Using this feature, JPF can be improved in many different ways. Many functionalities, such as its race detection, have been already added to JPF.

One way to improve JPF would be providing it with automatic support for handling native calls. As is explained in Section 5.4, JPF provides different ways to handle native calls which are not automatic and require some work. Adding this automatic support can considerably extend the use of JPF. Part of our future plan is to use JPF to model check a group communication toolkit, JGroups\(^9\), which is written in Java. To make the whole process automatic, we are planning to provide JPF with an extension to handle native calls automatically.

\(^9\)http://www.jgroups.org/
References


