Towards Model Checking of Computer Games with Java PathFinder

Nastaran Shafiei* and Franck van Breugel*†
*DisCoVeri Group, Department of Computer Science and Engineering
York University, 4700 Keele Street, Toronto, M3J 1P3, Canada
†Department of Computer Science
University of Oxford, Parks Road, Oxford, OX1 3QD, UK

***Abstract*—We show that Java source code of computer games can be checked for bugs such as uncaught exceptions by the model checker Java PathFinder (JPF). To model check Java games, we need to tackle the state space explosion problem and handle native calls. To address those two challenges we use our extensions of JPF, jpf-probabilistic and jpf-nhandler. The former deals with the randomization in the source code of the game, which is a cause of the state space explosion problem. The latter handles native calls automatically. We show how JPF enhanced with our extensions can check games such as the text based game Hamurabi and a graphics based version of rock-paper-scissors.**

I. INTRODUCTION

Numerous clips showing *bugs* in *computer games* can be found on YouTube. A notorious example is the so-called "Jesus shot", where Tiger Woods is able to walk on water. EA Sports, who developed the game, subsequently posted a clip in which Tiger Woods does walk on water with the message "*It's not a glitch. He's just that good.*" These clips demonstrate that, as in other types of software, bugs are omnipresent in the code of games. Therefore, *tools* that can detect those bugs are valuable to game developers.

Testing is the most commonly used method to detect bugs and it is also used for games. Although test suites may be developed for some components of the game, there is not a lot of automated testing [14]. The game as a whole is tested as well. The latter is often done manually by humans simply playing the game [25]. Overall, very few tools are used to test games.

The source code of most games contains some sort of *randomization*.[1] This provides games with the ability to surprise players, which is a key factor to their long-term appeal [23, page 350–353], but which is also a source of *nondeterminism*: although the player uses the same strategy, the game may evolve differently due to the randomization.

It is well known that nondeterminism causes difficulties for testing. First of all, tests usually cannot control the nondeterminism and, hence, running a test multiple times does not guarantee that different executions are tested. Secondly, if a test detects a bug, it may be difficult to reproduce the bug. *Model checking* is an alternative to testing. Rather than checking a single execution, as is done in testing, model checking attempts to systematically check all potential executions. However, the number of potential executions can be exponential in the number of nondeterministic choices in the code. The problem of dealing with such a huge number of executions is known as the *state space explosion problem*. Tackling this problem is one of the major challenges in model checking.

Due to the presence of nondeterminism, we believe that alternatives to testing, such as model checking, will be useful for detecting bugs in the source code of games. Our aim is to develop tools, based on model checking, that can automatically detect bugs in the source code of games. This paper describes our first attempt towards that goal.

Until recently, software that uses randomization was only verified by analyzing a model of the software, rather than the *source code* itself. A model is usually simpler than the source code and, hence, the model is generally easier to verify. However, the model abstracts from certain details not considered relevant to the verification effort and, hence, violations of certain properties might not be detected. But such violations might well be detected if we consider the source code. Only recently, tools have been developed that work directly with the source code (see [13], [28]). Whereas a tool that checks properties of a model is usually exploited to find errors in algorithms, a tool that considers the source code is generally used to detect *coding errors*. Hence, both types of tool play their role in the verification process.

In this paper, we focus on model checkers that work directly with the source code. We restrict our attention to *Java*. This is one of the most popular programming languages.[2] Although C++ is most widely used for games, many, in particular online games, are written in Java. For example, *Minecraft*, of which more than nine million copies have already been sold, is written in Java.[3] According to Guinness World Records, *RuneScape*, also implemented in Java, is the world's most popular free massively multiplayer online role-playing game with more than 200 million registered accounts.

[1] The way in which randomization is used in games is fundamentally different from the way it is used in systems verified by model checkers such as PRISM [15]. Whereas the probabilities that capture the AI logic of a game are hard coded in the source code of the game, the probabilities in systems checked by PRISM are estimates obtained by experiments.

[2] The TIOBE programming community index is an indicator of the popularity of programming languages and can be found at http://www.tiobe.com/index.php/content/paperinfo/tpci.

[3] http://www.joystiq.com/2012/05/25

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Several model checkers for Java have been developed. Most of those, including Bandera [9], Borgor [22], and an extension of SAL [19], translate the Java code to a model and subsequently verify properties of that model. The model checker *Java PathFinder*[4] (JPF) [27] is an example of a tool that works directly with Java bytecode.

The development of JPF started at NASA in 1999. Initially, JPF also translated Java bytecode to a model, which was subsequently passed to the SPIN model checker. In 2000, JPF was refactored as a Java virtual machine (JVM). Since JPF itself has been written in Java, it runs on top of another JVM, which we call the *underlying JVM*. In contrast to an ordinary JVM, JPF systematically explores all potential executions of the code, rather than a single one. While exploring those executions, JPF tries to find violations of properties such as uncaught exceptions.

JPF can easily be extended to check for further properties. A number of such extensions are available. For example, the extension *jpf-numeric*[5] checks for arithmetic overflows and underflows and the extension *jpf-ltl*[6] allows for the verification of properties expressed in linear temporal logic (LTL). For example, the property that players can always reach the end of the game, no matter how they have played so far—something that is desirable in many games—can be expressed in LTL. Many other interesting properties can be expressed in LTL. Zhang and Van Breugel's extension *jpf-probabilistic*[7] [29] is essential for our approach. It extends JPF so that it can handle Java code with randomization. Another reason to focus on JPF is that it is an open-source project.

The Java code of most games contains native calls. Such a *native call* invokes code that is written in a language different from Java such C. Calls related to graphics, networking and sound are usually native. To make our tools of use to game developers, native calls should be handled automatically: in addition to battling the state space explosion problem, this is a second major challenge. We base our solution on our extension *jpf-nhandler*[8] [24] which provides support for native calls.

Rather than developing our tools from scratch, we extend JPF. This approach allows us to develop prototypes quickly. The development of similar tools for C++ would take considerably more time. However, we believe that our techniques should be transferable to model checkers for C++ and we consider that a promising direction for future research. As we have already mentioned, in this extension we need to address two major issues: the notorious state space explosion problem and handling native calls. Although numerous techniques have been developed to combat the state space explosion problem, none of them suffice when considering the source code of games. Hence, a new approach that complements existing ones is needed. Several ways to handle native calls have been proposed, but none of them is automatic, something that is essential if we want to increase the productivity of game developers. Therefore, we aim for an automatic method to deal with native calls.

[4]http://babelfish.arc.nasa.gov/trac/jpf
[5]http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-numeric
[6]https://bitbucket.org/michelelombardi/jpf-ltl
[7]http://bitbucket.org/discoveri/jpf-probabilistic
[8]https://bitbucket.org/nastaran/jpf-nhandler

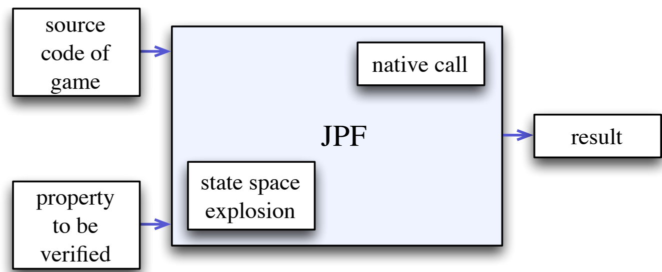

Fig. 1. Overview of our tool

There are numerous types of bugs in games. A taxonomy of these bugs can be found in [17]. Here, we focus on coding errors such as uncaught exceptions, arithmetic overflows and underflows, since JPF already provides support to detect such bugs. Other types of bugs are left for future research.

An overview of our tool is given in Fig. 1. Starting from JPF, we extend it in such a way that we can combat the state space explosion problem as manifested in games, and handle native calls, which are omnipresent in games. This extension of JPF takes as input the Java bytecode of the game and a property to be verified. The latter could for example be the absence of uncaught exceptions. The result of the verification effort of the extended model checker can be either confirmation that the property is satisfied by the code (in the case that JPF does not run out of memory), a lower-bound of the probability that the property holds (in the case that JPF runs out of memory without detecting a violation of the property), or a counterexample showing that the property does not hold. All three types of result are useful. Obviously, a counterexample is useful to correct the bug that has been detected. But also the lower-bound provides us with useful information. It can be viewed as a measure of reliability of the game. As far as we know, no such tool, which forms the basis for a valuable addition to every game developer's toolbox, exists yet.

Although we concentrate on games, focus on model checkers that work directly with source code, restrict our attention to Java code, and implement our ideas in JPF, we should stress that many of our objectives and results are applicable to a much wider range of software and model checkers.

II. Limitations of Existing Work

The only work on tools for finding bugs in the source code of games of which we are aware is the paper by Lewis and Whitehead [16]. In their paper, they describe a tool called *Lakitu* which detects and corrects bugs. To use the tool, one has to manually insert events into the source code of the game. A bug is captured by a violation of an invariant specified in terms of the events that have been inserted. One has to specify these invariants and the code that corrects the bug. They successfully applied Lakitu to the computer game *Super Mario*

World. As already pointed out by the authors, a drawback of their approach is that the events have to be added manually. Also, a run of their tool only checks a single execution of the game and Lakitu does not guarantee that multiple runs of the system check different executions of the game. In contrast, we aim for tools that are automatic and that explore as many (fragments of) executions of the game as possible.

A. State Space Explosion Problem

Numerous approaches have been developed to combat the state space explosion problem (see, for example, [4, Section 2.3]). Two commonly used techniques are partial order reduction and state compression. Partial order reduction reduces the number of executions that need to be checked by considering concurrently executed instructions that do not affect each other. Many model checkers, including JPF, implement some form of partial order reduction. In state compression, each state is decomposed into several components so that different states that agree on a component can share that component. JPF also implements some form of state compression. As argued in, for example [21], the best way to combat the state space explosion problem is to combine several techniques.

Since the source code of games usually contains randomization, the state space explosion problem is unavoidable despite the powerful techniques that have been developed to combat the problem. As a consequence, when we model check the source code of a game, the model checker usually runs out of memory after some time. Having waited for several minutes, hours or even days, a message like "out of memory" is not very informative, not to mention also frustrating. To address this problem, Zhang and Van Breugel [30] developed a general notion of *progress measure* for a model checker. This notion is defined in terms of a Markov chain and the property under verification. The Markov chain captures the randomization in the source code under verification. The amount of progress made by a model checker towards verification of the property is captured by a real number between 0 and 1. The larger the number, the more progress the model checker has made. A similar notion has been proposed by Pavese et al. [20].

A progress measure provides a lower-bound on the mass of the set of executions satisfying the given property. For example, assume that the progress towards verifying the property φ is 0.9999. Then, the probability that we encounter a violation of φ when we run the code is at most 0.0001. The property that players can always reach the end of the game, no matter how they have played so far is in general undecidable. However, using the progress measure we can provide an upper-bound of the probability that this property is violated. Hence, even if the model checker fails by running out of memory due to the state space explosion problem, this progress measure provides useful information about the code, possibly making the verification a success.

B. Native Calls

Several different approaches have been developed to handle native calls. For example, let us consider two such approaches developed for JPF. In [5], Barlas and Bultan introduce a framework called *NetStub* to model check distributed Java applications. The framework consists of several packages that reimplement those parts of the Java standard library related to network communication and containing native calls. These reimplemented packages simulate a network. Each component of the distributed application is represented by a thread so that the whole distributed application runs in a single JVM and, hence, can be verified by JPF.

Artho et al. [3] propose a different approach to model check distributed Java applications. Only one component of the distributed application is model checked, the other ones are simply executed. Since the model checker attempts to systematically check all potential executions of that one component, one has to be careful to prevent communications between that component and the other components from being repeated. To address that problem, Artho et al. introduce a cache that keeps track of those communications. As in the work of Barlas and Bultan, those classes that contain native calls are reimplemented.

To apply either of these approaches to our setting, one would have to reimplement those classes used by the game that contain native calls. Since native calls are omnipresent in the source code of games, this would be a huge amount of work. Furthermore, for some classes only the Java bytecode may be available (and not the Java code), making it even harder to reimplement the class. In Section IV we describe an alternative to the above described approaches that handles native calls automatically.

III. Handling Randomization with JPF-probabilistic

If we model check Java code that contains randomization, JPF does not take into account any probabilities associated with the probabilistic choices in the code. The extension *jpf-probabilistic* [29] takes the probabilities seriously. To express those probabilistic choices in the Java code, the class `Choice` has been introduced. This class which contains the static method `make(double[] p)`. Given an array `p`, which represents a probability distribution on the set $\{0, \ldots, \texttt{p.length} - 1\}$, the method call `Choice.make(p)` returns i with probability `p`$[i]$. For example, the call `Choice.make(0.5, 0.5)` returns either 0 or 1, each with probability 0.5. For convenience, the extension also contains the classes `Coin`, `Die` and `UniformChoice`.

The extension has been developed in a modular way such that it can deal with other methods that can be used to express probabilistic choices such as the `nextInt(int n)` method of the `Random` class in the future in the same way as it handles the `make` method. For now, method calls such as `nextInt` need to be replaced manually with a corresponding code snippet using `make`. Game developers can also use, for example, `(int) (Math.random() * n)` to express probabilistic choices. Detecting all probabilistic choices in the source code is a challenge that we leave for future research.

JPF can traverse the state space in different ways by using, for example, a depth-first search or a breadth-first search. Since jpf-probabilistic takes the probabilities into account, it can use them to guide the search. For example, jpf-probabilistic contains a random search strategy which randomly selects the next state to explore. The probability that a particular state is further explored is proportional to the probability of the path along which the state was discovered by the search (see [28, Chapter 5] for more details). Since our ultimate goal is to handle multiplayer real-time games with huge state spaces, efficiency is key. Therefore, we reimplemented Zhang's random search.

In Zhang's implementation of random search, which provided just a proof of concept, the potential states to be explored next are stored in a list. Each state is associated with a probability, which is the probability of the path along which the state was discovered during the search. During the search, these probabilities need to be summed regularly. To reduce the effects of rounding errors, it is beneficial to keep the probabilities sorted. As a consequence, updating the list takes time linear in the size of the list. Instead of a list, we use a red-black tree, the nodes of which are decorated with probabilities. The update operations become logarithmic in the size of the tree. A similar data structure has been proposed by, for example, Stoelinga [26, page 133].

As we already mentioned earlier, Zhang and Van Breugel's progress measure provides a quantitative notion of reliability of the code in case the model checker runs out of memory before detecting any bugs. In jpf-probabilistic, the progress can be computed when verifying *invariants*. These invariants form an important class of properties. In particular, for source code this class plays a key role. For example, we may want to check that the code never gives rise to any uncaught exceptions, or that it never causes overflow. These types of property are all expressed as invariants. During JPF's verification effort, the extension jpf-probabilistic builds a Markov chain representing the probabilistic choices in the code. It hands this Markov chain to the probabilistic model checker MRMC [12]. Subsequently, MRMC computes the probability of reaching a particular (so-called sink) state in the Markov chain (for more details, see [28, Chapter 7]). It has been shown in [30] that this probability coincides with the progress measure for invariants (for properties other than invariants see [11]).

In comparison to Zhang's implementation, we slightly simplify the representation of the Markov chain that is built. Furthermore, instead of handing the Markov chain to MRMC, we serialize the Java object representing the Markov chain. This has several advantages. First of all, MRMC need not be installed to use jpf-probabilistic, since jpf-probabilistic now contains the application `Progress` which computes the progress. Furthermore, the serialized Java object can also be fed to other tools that can compute the reachability probability. For example, one could exploit a GPU to compute the reachability probability (see [6], [10]).

The application `Progress` uses a standard approach to compute the reachability probability (see, for example, [4, Section 10.1.1] for details). First, those states that cannot reach the sink state are computed. Next, those states that always reach the sink states are computed. Finally, Jacobi's algorithm is applied to the remaining states. Improving the efficiency of this application is left for future work.

IV. Handling Native Calls with jpf-nhandler

If we model check Java code that contains native calls, JPF generally crashes by throwing an error. Fortunately, JPF provides two ways to handle native calls. Both approaches require the user of JPF to reimplement the classes containing the native methods. This is tedious and error prone since native methods are often not documented, or even only the Java bytecode is available (and not the Java code). Since we cannot expect game developers to reimplement numerous classes containing native calls in order to find bugs in their game, we exploit the extension jpf-nhandler [24] to handle native methods automatically.

Model classes provide a way to handle native calls in JPF. It uses these classes as alternatives to actual classes from Java libraries. For example, including the model class `java.lang.System` forces JPF to ignore the `System` class from the standard Java library, and instead model check the model class. To handle a native call, one can model its implementation in the corresponding model class.

Another way to handle native calls is to use JPF's *model Java interface* (MJI). JPF uses MJI to transfer the execution from JPF to the underlying JVM. The so-called *native peer* classes play a key role in the MJI implementation. JPF uses a specific name pattern to associate the native peer classes and their methods with the corresponding classes containing native methods. For example, the native peer associated with `java.lang.System` is named `JPF_java_lang_System`. Whenever JPF gets to a call associated with a native peer method, it delegates the call to the underlying JVM. Hence, the native call is not model checked (which is impossible since JPF can only handle Java bytecode), but executed in the underlying JVM.

To handle native calls automatically, jpf-nhandler relies on MJI and native peers. Whenever JPF encounters a native call, jpf-nhandler automatically intercepts and delegates the execution of the native method from JPF to the underlying JVM. It creates bytecode for native peers on-the-fly (referred to as OTF peers from now on) using the BCEL library[9]. To delegate the execution of a method to the underlying JVM, jpf-nhandler adds a method in the corresponding OTF peer which implements the following three main steps. For example, consider the native call `System.mapLibraryName(s)`. This call is delegated by jpf-nhandler as follows.

1) First, the JPF representation of the string `s` is transformed to a corresponding JVM object. (For non-static method, the object on which the method is called needs to be transformed as well.)

[9]The byte code engineering library: http://commons.apache.org/bcel

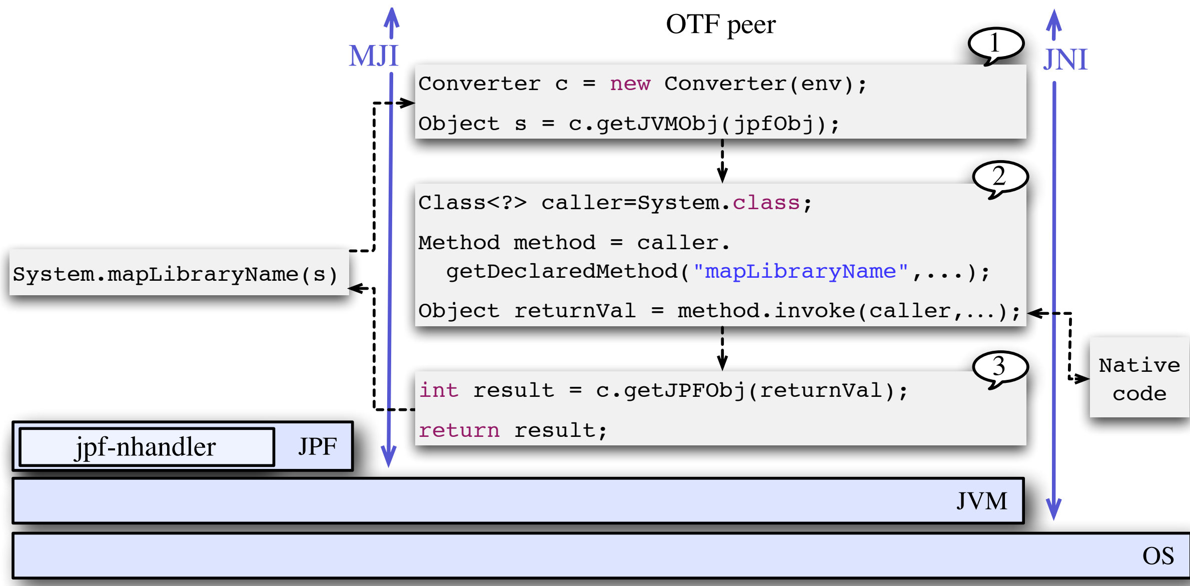

Fig. 2. Overview of the handling of native calls by jpf-nhandler

2) Then the execution is delegated to the underlying JVM by calling the original native method `mapLibraryName` with the JVM representation of `s` as its argument.
3) Finally, the result of the method call, which is a string in our example, is transformed from its JVM representation to its JPF representation.

Since objects are represented differently in JPF and the underlying JVM, jpf-nhandler has to be able to transform objects from JPF to the underlying JVM and back. Our class `Converter` accomplishes that. Its method `getJVMObj` transforms JPF objects to JVM objects and `getJPFObj` transforms objects in the other direction.

Fig. 2 shows how jpf-nhandler copes with the native call `System.mapLibraryName(s)`. Note that the execution of `mapLibraryName` is transferred all the way down to the native level where it is eventually executed. jpf-nhandler provides a way for the user of JPF to specify which (native and non-native) methods need to be delegated. This feature is essential when we model check games by means of JPF and its extension jpf-nhandler.

V. Model Checking of Hamurabi

As a first game, we consider the text-based game *Hamurabi*. This is one of the very first computer games. It was implemented in BASIC by Ahl in the seventies [1, page 128]. We ported the BASIC code to Java (see Fig. 3). Although this game does not use any graphics, it still uses native calls to read from the keyboard and write to the console. Although the latter native calls are handled by JPF, the former are not. Therefore, if we were to model check this Java game with JPF (even with our extensions jpf-probabilistic and jpf-nhandler enabled), JPF would crash.

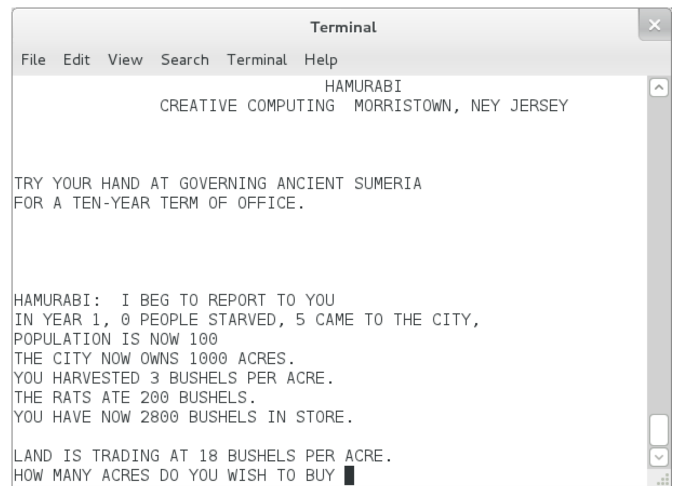

Fig. 3. Screenshot of Hamurabi

In Java, reading from the keyboard is done by reading from the "standard" input stream. This stream is captured by the attribute `in` of the class `System`. This class provides also static methods that can manipulate the JVM (for example, `gc()` runs the garbage collector). Since JPF itself is a JVM, it should come as no surprise that JPF reimplements this class (partly as a model class, partly as a native peer). Although JPF's `System` model class contains a declaration of the `in` attribute of type `InputStream`, this attribute is not initialized. Hence, JPF throws a `NullPointerException` when it encounters `System.in`.

To address this problem, we have modified JPFs' `System` model class. Its attribute `in` is initialized as `in = new ConsoleInputStream()`. The class

`ConsoleInputStream` is a model class that we have added to JPF. We also added a corresponding native peer to JPF. The model class extends the class `InputStream` (to ensure that the above assignment is valid), it contains a default constructor with an empty body, and all its methods are declared native. As a consequence, whenever JPF encounters a method called on `System.in`, it does not model check it but, instead, delegates the execution of the corresponding native peer method to the underlying JVM. The methods in the native peer simply call the method on the JVM's `System.in`. For example, the body of the native peer method `read()` simply returns `System.in.read()`.

After having made these adjustments to JPF and enabling our extensions jpf-probabilistic and jpf-nhandler, we are able to model check Hamurabi with JPF. Each game of Hamurabi needs at least one and at most fourty integers as input from the player. Rather than providing the player's input to the game by means of the keyboard, we created a file consisting of a sufficient number of integers (roughly half a million integers—recall that JPF attempts to explore all potential executions of the game) and redirected the input so that JPF gets it from the file instead of the keyboard.

In the default configuration, JPF uses depth-first search and checks for uncaught exceptions. JPF, with the default configuration, ran out of memory after four hours. In that time, it had checked almost half a million different executions of the game and found no uncaught exceptions.[10] It took less than two minutes to compute the progress: 0.80. Hence, from this verification effort by JPF we can conclude that the probability of encountering an uncaught exception, provided that the game is played as specified in the input file, is at most 20%.

We also configured JPF to use our random search. In this case, JPF ran on average out of memory in less than ten minutes. JPF ran out memory much faster due to the extra memory needed for the decorated red-black tree used by our random search. In this case, JPF checked on average only 50,000 different executions of the game and again found no uncaught exceptions. However, the progress only took a few seconds to compute and was on average 0.97. Despite the fact that random search checks considerably fewer executions, it checks a more important (from a probabilistic perspective) part of them and, hence, provides a much better upper bound on the probability of encountering an uncaught exception (3% rather than 20%).

VI. Model Checking of Rock-Paper-Scissors

We consider *rock-paper-scissors* as our second example. We model check a simple version of the game in Java (see Fig. 4). Instead of implementing metastrategies which defeat second-guessing, triple-guessing, etcetera, we simply randomly select rock, paper or scissors. Since the game uses graphics, it relies heavily on native calls. Although our extension jpf-nhandler takes care of native calls, if we were to model check this

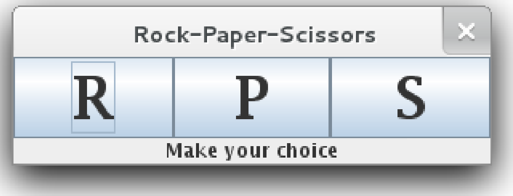

Fig. 4. Screenshot of rock-paper-scissors

game using JPF with the extensions jpf-nhandler and jpf-probabilistic, JPF would crash.

To model check this game, we first configured jpf-nhandler to delegate all native calls. The part of the code that creates the GUI of the game includes a native call `loadNativeDirFonts()`. The current version of jpf-nhandler can only handle a native call if its side effects are limited to its return value, its arguments, and the class or object on which the native method is called. But the side effects of `loadNativeDirFonts()` go beyond that, since it initializes some attributes in the underlying JVM. This causes JPF to throw a `NullPointerException`, because these attributes have not been initialized by JPF, but they are used by JPF. To solve this problem, we configured jpf-nhandler to delegate the whole (non-native) method that contains the native call `loadNativeDirFonts()`.

Moreover, to handle rock-paper-scissors we had to extend the functionality of jpf-nhandler. To model check this game it is essential that jpf-nhandler keeps track of all the JVM objects that it creates using the `getJVMObj` method of the `Converter` class. By keeping a reference to these objects, JPF can observe (some of the) changes made to these objects by the underlying JVM. This turned out the be essential to deal with rock-paper-scissors.

After having extended and configured jpf-nhandler, we can successfully model check rock-paper-scissors with JPF. For this very simple game, the major challenge is to handle the native calls automatically: whereas our extension of JPF can do this, neither JPF nor any of its other extensions can. Since the state space is small, JPF does not run out of memory. Hence, there is no need to compute the progress.

VII. Conclusion

We have presented a model checker that can verify properties of computer games. We believe this to be the first such tool. We have shown that our extension of JPF can handle simple games such as Hamurabi and rock-paper-scissors. Although our initial results are promising, a lot of work remains to be done as most games are much more complex.

Even for a considerably simplified version[11] of Hamurabi, there are more than 10^{10} different plays. For each play, it takes JPF more than two minutes to verify the game for, for example, uncaught exceptions. Verifying all different plays one by one

[10] Our initial implementation of Hamurabi contained a bug which caused an exception. This bug was found by JPF within seconds.

[11] Instead of ten years, the player is only in office for one year.

would take more than four millennia. Hence, in such a case, we have to restrict ourselves to "interesting" plays. As pointed out by, for example, Cadar and Engler [7], generating input (in our case, plays) that will explore all the "interesting" behaviour in the tested program (in our case, the game) remains an important open problem in software testing research. In the case of our simplified version of Hamurabi, there are only six different "interesting" plays, which can be checked by JPF within 15 minutes. Identifying those "interesting" plays, possibly with the help of the game developer, is a topic we plan to address in the future.

One obvious way to reduce the number of plays is to represent them symbolically. However, *jpf-symbc*[12] [2], a symbolic extension of JPF, when applied to the source code of our considerably simplified version of the text-based game Hamurabi runs out of memory in our experiments. We conjecture that this is caused by the randomization in the code of the game. Recently, Claret et al. [8] proposed a novel way to symbolically execute probabilistic programs. A key ingredient of their approach is the use of algebraic decision diagrams. These are a generalization of the well-known binary decision diagrams. They allow for a compact representation of probability distributions and an efficient implementation of operations on probability distributions. By extending jpf-symbc so that the randomization of the game is represented by algebraic decision diagrams, we may obtain a symbolic model checker that does not run out of memory when confronted with a simple game such as Hamurabi.

Games typically implement sophisticated graphical user interfaces (usually much more complicated than the one for rock-paper-scissors). The extension *jpf-awt*[13] [18] of JPF provides a framework to model check graphical user interfaces. In op. cit., Mehlitz et al. propose a scripting language that captures the interactions with the graphical user interface. These scripts are interpreted by JPF. This approach is not automatic as the scripts describing the interactions need to be written. The approach is also limited to graphical user interfaces, whereas most games also read from the keyboard. We aim to extend this approach so that it can applied to games and we also intend to automate the approach as much as possible.

Acknowledgements

The authors would like to thank Chris Kingsley, Daniel Kroening and Peter Mehlitz for discussion, Steven Xu and Xin Zhang for their help with the development of jpf-probabilistic, and the referees for their numerous and constructive comments.

REFERENCES

[1] David H. Ahl, editor. *101 BASIC Computer Games*. DEC, 1973.
[2] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In *TACAS*, pages 134–138, 2007.
[3] Cyrille Artho, Watcharin Leungwattanakit, Masami Hagiya, and Yoshinori Tanabe. Efficient model checking of networked applications. In *TOOLS*, pages 22–40, 2008.
[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.
[5] Elliot D. Barlas and Tevfik Bultan. NetStub: a framework for verification of distributed Java applications. In *ASE*, pages 24–33, 2007.
[6] Dragan Bošnački, Stefan Edelkamp, Damian Sulewski, and Anton Wijs. Parallel probabilistic model checking on general purpose graphics processors. *International Journal on Software Tools for Techology Transfer*, 13(1):21–35, 2011.
[7] Cristian Cadar and Dawson Engler. Execution generated test cases: how to make systems code crash itself. In *SPIN*, pages 2–23, 2005.
[8] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. Bayesian inference for probabilistic programs via symbolic execution. Report MSR-TR-2012-86, 2012.
[9] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE*, pages 439–448, 2000.
[10] Elise Cormie-Bowins. A comparison of sequential and GPU implementations of iterative methods to compute reachability probabilities. In *GRAPHITE*, pages 20–34, 2012.
[11] Elise Cormie-Bowins and Franck van Breugel. Measuring progress of probabilistic LTL model checking. In *QAPL*, pages 33–47, 2012.
[12] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Performance Evaluation*, 68(4):90–104, 2011.
[13] Mark Kattenbelt. *Automated Quantitative Software Verification*. PhD thesis, University of Oxford, 2010.
[14] Chris Kingsley. Personal communication, 2012.
[15] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: verification of probabilistic real-time systems. In *CAV*, pages 585–591, 2011.
[16] Chris Lewis and Jim Whitehead. Runtime repair of software faults using event-driven monitoring. In *ICSE*, pages 275–280, 2010.
[17] Chris Lewis, Jim Whitehead, and Noah Wardrip-Fruin. What went wrong: a taxonomy of video game bugs. In *FDG*, pages 108–115, 2010.
[18] Peter C. Mehlitz, Oksana Tkachuk, and Mateusz Ujma. JPF-AWT: model checking GUI applications. In *ASE*, pages 584–587, 2011.
[19] David Y. W. Park, Ulrich Stern, Jens U. Skakkebæk, and David L. Dill. Java model checking. In *ASE*, pages 253–256, 2000.
[20] Esteban Pavese, Victor Braberman, and Sebastian Uchitel. My model checker died!: how well did it do? In *QUOVADIS*, pages 33–40, 2010.
[21] Radek Pelánek. Fighting state space explosion: review and evaluation. In *FMICS*, pages 37–52, 2008.
[22] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *FSE*, pages 267–276, 2003.
[23] Katie Salen and Eric Zimmerman. *Rules of Play: Game Design Fundamentals*. The MIT Press, 2004.
[24] Nastaran Shafiei and Franck van Breugel. Automatic on-demand delegation of calls in Java PathFinder. Available at http://www.cse.yorku.ca/~franck/research/drafts, 2013.
[25] Karla Starr. Testing video games can't possibly be harder than an afternoon with Xbox, right? *Seattle Weekly*, 2007.
[26] Mariëlle Stoelinga. *Alea jacta est: Verification of probabilistic, real-time and parametric systems*. PhD thesis, Katholieke Universiteit Nijmegen, 2002.
[27] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.
[28] Xin Zhang. Measuring progress of model checking randomized algorithms. Master's thesis, York University, 2010.
[29] Xin Zhang and Franck van Breugel. Model checking randomized algorithms with Java PathFinder. In *QEST*, pages 157–158, 2010.
[30] Xin Zhang and Franck van Breugel. A progress measure for explicit-state probabilistic model-checkers. In *ICALP*, pages 283–294, 2011.

[12] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
[13] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-awt