Volume 5, Issue 11, November 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Hunting For Bugs in Libraries for Concurrent Programming

[1]Emmanouela Stachtiari, [2]Apostolos Chatzopoulos, [3]Ignatios Deligiannis, [4]Panagiotis Katsaros
[1, 2, 4] Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece
[3] Alexander Technological Educational Institute of Thessaloniki, 57400 Sindos, Greece

Abstract: Synchronization bugs in concurrent applications result in errors (deadlocks, race conditions etc.), which are not easily reproducible and it is therefore practically impossible to find them with ordinary testing. Software libraries for concurrency programming are supposed to provide proven solutions for synchronization, in terms of safety (e.g. mutual exclusion, correct ordering of operations) and liveness (e.g. absence of starvation). However, in most cases they lack proper documentation or formal specifications of the semantics of the provided operations. Software model checking is a viable alternative against testing for concurrency verification, but it is only feasible if the API is considered as part of a larger system with a specific functionality. In this work, we present our experience on the software model checking of an open source library using NASA's JavaPathFinder (JPF) model checker. Our approach was based on the API's test classes, which were transformed into an analyzable input for the JPF tool. The results can reveal bugs or can simply provide valuable feedback towards the formal specification of the API's operation semantics.

Keywords: API, concurrency, software model checking.

I. INTRODUCTION

Concurrency APIs offer a set of synchronization methods such as semaphores, mutexes and barriers that are used to ensure thread-safety. This allows multiple threads to work concurrently on shared resources without data corruption.
However, such an API is usually classified – modulo some differences in the terminology – into three categories, namely thread-safe, conditionally safe and no thread-safe. Any API-based application will be thread-safe, if the library is used in a manner consistent with the provided guarantees. Only the APIs of the first category are guaranteed to be free of race conditions, when accessed by mutltiple threads simultaneously. In the second case, not all the operations of the API are thread-safe. Thread-safety guarantees may also include design solutions to prevent or limit the risk of different forms of deadlocks, but deadlock-freedom cannot be ensured at the API level alone.
Most APIs lack an adequate documentation or formal specifications of the semantics of the provided operations. This is the main source of synchronization bugs in concurrent applications, which are not easily reproducible. Ordinary testing is therefore useless and more advanced verification methods are needed.
In this work, we cope with the problem of software model checking open source concurrency libraries using NASA's JavaPathFinder (JPF) model checker [1, 2]. JPF uses its own JVM implementation to perform enumerative state space exploration of the Application Under Test (AUT). In a concurrent AUT, all the non-determinism is attributed to two sources: inputs from the environment and scheduling choices made by the scheduler. JPF explores the set of all behaviors by analyzing the behavior of the process under all possible inputs and schedules. A dynamic partial-order reduction approach is followed by initially exploring an arbitrary interleaving of the concurrent threads, while dynamically tracking their interactions, in order to identify backtracking points where alternative paths in the state space need to be explored.
Still, the verification of open source concurrency libraries is a challenging task, due to the fact that model checking is only feasible if the library is considered as part of a larger system with a specific functionality. We propose using appropriate API test classes that adequately cover widely used scenarios of API call sequences. Our approach was applied to the Amino Concurrent Building Blocks API [6] and the Java Grande Forum Benchmark Suite [7]. We show how the API's test classes were transformed into an analyzable input for the JPF tool.
The results can reveal bugs in the software library implementation or they can simply provide valuable feedback towards the formal specification of the API's operation semantics.
The rest of the paper is structured along the following lines. Section II discusses related work on concurrency API verification. In Section III, we review the process of software model checking with the JPF tool. Section IV shows the process of transforming an API's test classes into an analyzable input for the JPF tool. The paper concludes with a critical review of the presented approach.

II. RELATED WORK

Verifying concurrent applications is always a challenge, regardless of the tools employed and the state of specifications for the AUT. This is because software model checkers explore all possible execution paths of the AUT, which often leads to a computationally intractable state space. With a closer look in the AUT, it can be observed that the length of these paths and therefore the size of the state space depend on the number of the different fields and parameters used by

the AUT. However, not all of these fields are important in the process of verifying a software model of the AUT and as a result they can be abstracted from it.
The act of creating an abstract model that derives from the original one is the most crucial step in the verification process. The new model must have the same behavior regarding its functionalities and at the same time it will have to reduce the fields and methods used. This will minimize the risk of state space explosion and will therefore allow the quality engineers to produce many more tests and expose more defects of a model in an acceptable time frame.
Many studies have turned their focus on this subject and most of them have a common starting point, which is the JPF tool [3]. When dealing with the special case of software libraries, most approaches aim to model complex library classes at a higher level of abstraction. The model classes need to have the same interface as the actual library classes but they can exhibit reduced behavior and state. Writing such model classes has proven to be both error prone and time consuming, so the need for special skills and tools is always in order.
A recent study has presented two alternative methods of creating abstract models for later input to the JPF [4]. This approach consists of two algorithms, which automatically generate model classes for model checkers. The first algorithm generates models using slicing to produce classes that implement the behavior associated with all the relevant fields. On the other hand, the algorithm may introduce dependencies on the irrelevant fields, something that may lead to problems during the actual usage of the model class. The second algorithm is a combination of slicing and value generation. The generated model classes include the behavior associated with relevant fields. Dependencies on irrelevant fields are then replaced with dependencies on local fields with a default value or values generated at random at runtime.
Another relevant work in the direction of Java software libraries is the implementation of a new abstraction method that claims to have better results than java.util.concurrent software library [5], when combined with JPF. Due to thread interleaving, the number of states that the model checker has to verify can grow rapidly and this impedes the feasibility of the verification. In the Java language, the source of thread interleaving apart from the AUT is also the Java Development Kit (JDK) itself. This approach aims to reduce the state space of the AUT, but it also provides support for native calls within the Java Concurrency Utilities that have not been supported in the JPF core project.
Overall, there are three key points in using JPF for verifying AUTs using concurrent APIs: appropriately modifying the AUT in order to be processed by the tool, carefully create an abstract model of the API and also define the right execution parameters that are needed, depending on the testing goals.
In this paper, we are interested in the characterization of concurrency APIs, for their thread-safety potentiality of AUTs relying on them. The focus is therefore moved from the problem of method abstraction towards an effective model checking procedure for the aforementioned goal.

III. SOFTWARE MODEL CHECKING WITH JPF

JPF is an explicit state software model checker for Java bytecode. Nowadays, JPF is involved in all sorts of runtime based verification purposes, as it integrates model checking, program analysis and testing. The JPF model checker employs library models, along with native peers, to tackle the state space explosion problem. Furthermore, JPF uses state compression to handle big states, partial order and symmetry reduction, slicing, abstraction and runtime analysis techniques to reduce the state space. This basically means that JPF is a Java virtual machine that executes your program not just once (like a normal VM), but theoretically in all possible ways, checking for property violations like deadlocks or unhandled exceptions along all potential execution paths.
In terms of bug detection, JPF can search for deadlocks, missed signals and unhandled exceptions (e.g. null-pointer exceptions and assertion errors), but the user can provide his own property classes or write listener extensions to implement other property checks. A number of such extensions like race condition and heap bounds checks are included in the JPF distribution.
In general, JPF is capable of checking every Java program that does not depend on unsupported native methods. The JPF VM cannot execute platform specific, native code. This especially imposes a restriction on what standard libraries can be used from within the AUT. While it is possible to write these library versions, there is currently no support for java.awt, java.net, and only limited support for java.io and Java's runtime reflection. Another restriction is given by JPF's state storage requirements, which effectively limit the size of checkable applications. Because of these library and size limitations, JPF has been mainly used so far for applications that are models, but require a full procedural programming language. JPF is especially useful to verify concurrent Java programs, due to its systematic exploration of scheduling sequences.
In order to tackle the continuously growing size of AUTs, JPF presents enhanced extensibility, with the employment of three major extension mechanisms.

- Search and VM Listeners
- The Model Java Interface (MJI)
- Configurable Choice Generators (CGs)

Search and VMListeners provide a convenient way to extend JPFs internal state model, to add more complex property checks, direct searches, or to simply gather execution statistics. The MJI is a mechanism to separate and communicate between state-tracked execution inside the JPF JVM and non-state tracked execution inside the underlying host VM (executing JPF itself). This can be used to build standard library abstractions that significantly reduce the application state space. CGs are used to implement application specific heuristics for "non-deterministic" data acquisition and scheduling policies, without the need to modify test drivers. ChoiceGenerators can be specified and parameterized via the normal JPF configuration mechanism and can be used from native methods and listeners within the MJI.

Regarding software libraries, JPF executes all library code used by the system under test, even though their size often significantly exceeds the application size. The Model Java Interface (MJI) provides a suitable mechanism to replace real library code with abstractions that can be executed outside of the JPF VM, for example modeling IO operations. Using MJI to abstract standard Java libraries is a major step towards applying JPF to real Java production code.

IV. SOFTWARE MODEL CHECKING CONCURRENCY APIS

In order to hunt for concurrency bugs in Java APIs, we analyzed the latest versions of two widely used APIs using the PreciseRaceDetector listener that is included in the main distribution of the JPF tool. Specifically, we performed the analysis on the Concurrent Building Blocks (CBBs) [6] and the Java Grande Forum Benchmark Suite (JGF) [7].

The Concurrent Building Blocks API provides a set of proven solutions, such as data structures, patterns and methods, which can be used to develop parallel/multi-threaded Java applications. The Java Grande Forum Benchmark Suite is a framework created to help investigating the development of the so-called Grande applications in Java. Grande applications are real scale applications, which are typically used in science and have large requirements in memory and processing power.

Each executable class of the two projects was analyzed for deadlock freedom and race condition. CBBs like the majority of APIs, is made available as a set of source classes that do not contain concrete applications for these classes. However, the project's extensive set of test cases could be used for the analysis after some instrumentation on the test classes, so that they constitute appropriate input for the JPF model checker.

Instrumentation was performed through transforming the original abstract syntax tree of each test class using the Eclipse Java Development Tools (JDT). JDT provides an API that represents the source code of a Java program as a structured document (i.e. document object model - DOM) and allows for the document's refactoring.

Since code instrumentation inserts invocations of methods declared in the gov.nasa.jpf.util.test.TestJPF class of the JPF source code, every test class would have to import and extend the TestJPF class. The extension was necessary, because the typical AssertXX() Junit methods should be replaced by the ones implemented in TestJPF. However, since not all the AssertXX() methods are implemented by the TestJPF class, (assertNotSame and assertNotEquals were missing), we implemented them in a new java class, which extends the TestJPF class, namely the ExtTestJPF class. Therefore, we transformed the test classes to extend the created ExtTestJPF class.

Then, a main method was added to all test classes, so that the tests can be executed by the JPF. The call to runTestsOfThisClass method makes it possible to either execute all test methods (i.e. those annotated with @Test) or only those whose names were passed as command-line arguments.

```
public static void main(String[]  testMethods){
        runTestsOfThisClass(testMethods);
}
```

Apart from the addition of a specific main, each test method's body was enclosed in a conditional branch that is entered if the verifyNoPropertyViolation() method returns true. This is a way to avoid the execution of the test method from the host VM (i.e. from JUnit). Instead, the verifyNoPropertyViolation() method creates and runs a new instance of JPF, which is set to execute the test method from which it was invoked. After the JPF instance has provided analysis results, an AssertionError might be thrown, if any of the expected properties will be found to be violated.

The JPF's output includes output written by the class under test while it is executed, as well as the possible errors (i.e. data races) with the associated information to reason about those errors. For the data races, localization of the error is provided, in terms of the involved class, variable and the line of code where the race was detected.

```
gov.nasa.jpf.listener.PreciseRaceDetector
race for field ClassA@152.v

Thread-1 at ClassA.run(ClassA.java:165)
              "double w = Cel * ClassA.v; "
Thread-2 at ClassA.run(ClassA.java:237)
    "ClassA.v = 0; "
```

Figure 1: Precise Race Detector's output for a data race

Figure 1 shows the output for a data race detected on the field v of ClassA. Two program threads where involved in the race, the one by reading and the other by updating v's value. Apart from the race location, a program trace is obtained in terms of code segments executed by the different threads. The trace represents the interleaving which led to the data race violation.

V. CONCLUSION

We presented a technical approach to address the problem of characterization of concurrency APIs, in terms of their thread-safety features with respect to applications, which rely on them. Our proposal is a software model checking technique based on the JPF tool. The details of the performed transformations, in order to detect data races in two software libraries were presented.

ACKNOWLEDGMENT

This research work is co-founded by the European Social Fund and National Resources, ESPA 2007-2013, EDULLL, “Archimedes III” program.

REFERENCES

[1] K. Havelund, T. Pressburger, (2000) "Model checking Java programs using Java Pathfinder", *Software Tools for Technology Transfer (STTT)* 2(4), pp72–84.

[2] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, (2003) "Model checking programs", *Automated Software Engineering Journal*, Vol. 10.

[3] W. Visser, P. Mehlitz, (2005) "Model checking programs with Java PathFinder", SPIN. Vol. 3639, pp27.

[4] M. Ceccarello, O. Tkachuk, (2014) "Automated generation of model classes for Java PathFinder", ACM SIGSOFT *Software Engineering Notes*, 39(1), pp1-5.

[5] M. Ujma, N. Shafiei, (2012) "jpf-concurrent: an extension of Java PathFinder for java.util.concurrent", *arXiv preprint arXiv*:1205.0042.

[6] Amino project - Concurent Building Blocks API. Available at http://amino-cbbs.sourceforge.net/ .

[7] Java Grande Forum Benchmark Suite. Available at https://www2.epcc.ed.ac.uk/computing/research_activities/jomp/grande.html .

AUTHORS

Emmanouela Stachtiari is a researcher and Ph.D. candidate at the Aristotle University of Thessaloniki, Greece, in the field of formal methods for web services. She received the B.S. degree in applied informatics from the University of Macedonia, Greece, in 2009, the Ms.C. degree in computer science from the Aristotle University of Thessaloniki, Greece, in 2011.

Apostolos Chatzopoulos is a localization project manager and researcher in Information Technology field. Apostolos received his Bachelor from the Information Technology department of Aristotle University of Thessaloniki and a Master's degree in Information Systems in the same department.

Ignatios Deligiannis is Professor at Technological Education Institute of Thessaloniki, Greece, and research associate at the University of Macedonia and the Aristotle University, Greece. He was also member of ESERG (Empirical Software Engineering Research Group at Bournemouth University, UK) . His main interests are object-oriented software assessment, and in particular design heuristics and measurement. He received his B.Sc. in computer science from the University of Lund, Sweden, in 1979, and then worked for several years in software development at Siemens Telecommunications industry.

Panagiotis Katsaros is an Assistant Professor in the Department of Informatics of the Aristotle University of Thessaloniki, Greece. His research interests are focused in the areas of formal verification, static program analysis and simulation-based performance analysis of systems and software. He has published over 80 papers in international journals and conference proceedings and he has scientifically coordinated a number of research projects funded by the European Commission and the European Space Agency.