UnitCheck: Unit Testing and Model Checking Combined*

Michal Kebrt and Ondřej Šerý

Charles University in Prague
Malostranské náměstí 25
118 00 Prague 1
Czech Republic
michal.kebrt@gmail.com, ondrej.sery@dsrg.mff.cuni.cz
http://dsrg.mff.cuni.cz

Abstract. Code model checking is a rapidly advancing research topic. However, apart from very constrained scenarios (e.g., verification of device drivers by SLAM), the code model checking tools are not widely used in general software development process. We believe that this could be changed if the developers could use the tools in the same way they already use testing tools. In this paper, we present the UNITCHECK tool, which enhances standard unit testing of Java code with model checking. A developer familiar with unit testing can apply the tool on standard unit test scenarios and benefit from the exhaustive traversal performed by a code model checker, which is employed inside UNITCHECK. The UNITCHECK plugin for ECLIPSE presents the checking results in a convenient way known from unit testing, while providing also a verbose output for the expert users.

1 Introduction

In recent years, the field of code model checking has advanced significantly. There exist a number of code model checkers targeting mainstream programming languages such as C, Java, and C# (e.g., SLAM [2], CBMC [6], BLAST [11], JAVA PATHFINDER [15], and MOONWALKER [7]). In spite of this fact, the adoption of the code model checking technologies in the industrial software development process is still very slow. This is caused by two main reasons (i) limited scalability to large software, and (ii) missing tool-supported integration into the development process.

The current model checking tools can handle programs up to tens of KLOC and often require manual simplifications of the code under analysis [13]. Unfortunately, such program size is still several orders of magnitude smaller than the size of many industrial projects.

* This work was partially supported by the Czech Academy of Sciences project 1ET400300504, and the Q-ImPrESS research project (FP7-215013) by the European Union under the Information and Communication Technologies priority of the Seventh Research Framework Programme.

Apart from the scalability issues, there is virtually no support for integration of the code model checkers into the development process. Although some tools feature a user interface in the form of a plugin for a mainstream IDE (e.g., SatAbs [5]), creation of a particular checking scenario is not often discussed or supported in any way. A notable exception is Slam and its successful application in the very specific domain of kernel device drivers.

These two obstacles might be overcome by employing code model checking in a way similar to unit testing – we use the term *unit checking* first proposed in [10]. Unit testing is widely used in industry and developers are familiar with writing test suites. Providing model checking tools with a similar interface would allow developers to directly benefit from model checking technology (e.g., of exploration of all thread interleavings) without changing their habits. Moreover, applying model checking to smaller code units also helps avoiding the state explosion problem, the main issue of the model checking tools.

Goals and structure of the paper. We present the UnitCheck tool, which allows for creation, execution and evaluation of checking scenarios using the Java PathFinder model checker (JPF). UnitCheck accepts standard JUnit tests [1] and exhaustively explores the reachable state space including all admissible thread interleavings. Moreover, the checking scenarios might feature nondeterministic choices, in which case all possible outcomes are examined. The tool is integrated into the Eclipse IDE in a similar way as the JUnit framework and also an Ant task is provided. As a result, users familiar with unit testing using JUnit might immediately start using UnitCheck.

Test example. Examples of two JUnit tests that would benefit from the analysis performed by UnitCheck (in contrast to standard unit testing) are listed in Figure 1. The code under test, on the left, comprises of bank accounts and bankers that sequentially deposit money to an account. The first test creates two bankers for the same account, executes them in parallel, and checks the total balance when they are finished. UnitCheck reports a test failure because the line marked with (1) is not synchronized. Therefore, with a certain thread interleaving of the two bankers the total balance will not be correct due to the race condition. In most cases, JUnit misses this bug because it uses only one thread interleaving. The second test demonstrates the use of a random generator (the `Verify` class) to conveniently specify the range of values to be used in the test. UnitCheck exhaustively examines all values in the range and discovers an error, i.e., the negative account balance. When using standard unit testing, the test designer could either use a pseudorandom number generator (the `Random` class) and take a risk of missing an error, or explicitly loop through all possible values, thus obfuscating the testing code.

2 Tool

In this section, the internal architecture of UnitCheck is described. Additionally, we discuss three user interfaces to UnitCheck which can be used to employ the tool in the development process.

```
public class Account {
   private double balance = 0;

   public void deposit(double a) {
      balance = balance + a; //(1)
   }
   public void withdraw(double a) {
      balance = balance - a;
   }
   public double getBalance() {
      return balance;
   }
}

public class Banker
       implements Runnable {
   private Account account;
   private double amount;
   private int cnt;

   @Override
   public void run() {
      for (int i=0; i < cnt; ++i) {
         account.deposit(amount);
      }
   }
}
```

```
@Test
public void testDepositInThreads() {
   Account account = new Account();
   Thread t1 = new Thread(
      new Banker(account, 5, 5));
   Thread t2 = new Thread(
      new Banker(account, 10, 5));

   t1.start(); t2.start();
   t1.join(); t2.join();
   assertEquals(account.getBalance(),
      25 + 50, 0);
}

@Test
public void testDepositWithdraw() {
   Account account = new Account();
   int income = Verify.getInt(0, 10);
   int outcome = Verify.getInt(0, 10);

   account.deposit(income);
   assertEquals(account.getBalance(),
      income, 0);
   account.withdraw(outcome);
   assertEquals(account.getBalance(),
      Math.max(income - outcome, 0), 0);
}
```

Fig. 1. Tests that benefit from UnitCheck's features

2.1 Architecture

An overview of the UnitCheck's architecture is depicted in Figure 2. The core module of UnitCheck, the actual integration of JPF and JUnit, is enclosed in the central box. It is compiled into a Java library so that it can be easily embedded into other Java applications (e.g., into an IDE). As an input, the core module takes an application under analysis, the JUnit-compliant test cases, and optionally additional properties to fine-tune JPF. The analysis is then driven and monitored via the `UnitCheckListener` interface.

It is important to note that neither JPF nor JUnit functionality and structures are directly exposed outside the core. The `UnitCheckListener` interface hides all JPF and JUnit specific details. This solution brings a couple of advantages. (i) Extensions (e.g., Eclipse plugin) implement only the single (and simple) listener interface. (ii) In future, both JPF and JUnit can be replaced with similar tools without modifying existing extensions.

Inside the core, `UnitCheckListener` is built upon two interfaces – JPF's `SearchListener` and JUnit's `RunListener`. `SearchListener` notifies about property violations (e.g., deadlocks) and provides complete execution history leading to a violation. `RunListener` informs about assertion violations and other uncaught exceptions. UnitCheck processes reports from both listeners and provides them in a unified form to higher levels through `UnitCheckListener`.

When analyzing the test cases, two Java virtual machines are employed. The first one is the host virtual machine in which UnitCheck itself and the underlying JPF are executed. The second one is JPF, a special kind of virtual machine, which executes JUnit. Subsequently, JUnit runs the input test cases (in Figure 2, the code executed inside the JPF virtual machine is explicitly marked). The information about test case progress provided by the JUnit's `RunListener` interface is available only in the JPF virtual machine. To make this information accessible in the host virtual machine, the JPF's Model Java Interface (MJI) API is used. It allows to execute parts of the application under analysis in the host virtual machine instead of the JPF virtual machine. Each class that is to be executed in the host VM has a corresponding peer counterpart. This mechanism is used for the `TestReportListener` class.

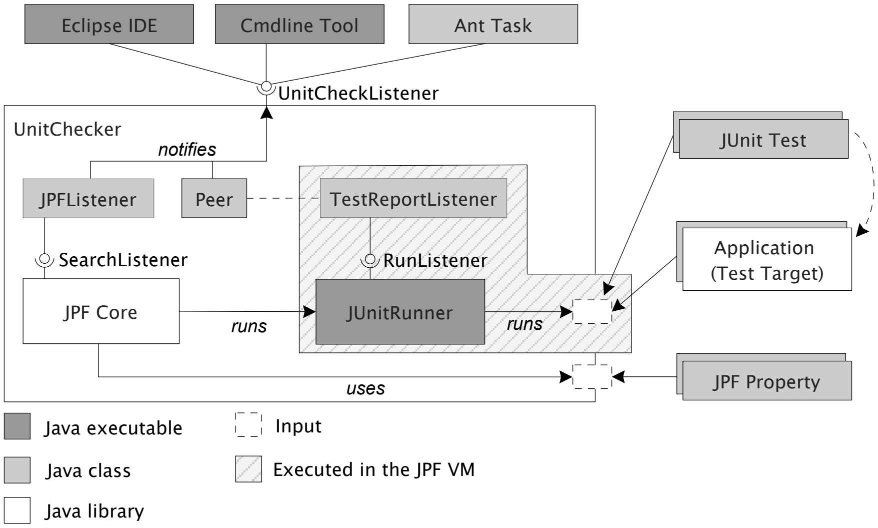

Fig. 2. JPF and JUnit integration

2.2 User Interface

Currently, there are three different extensions that provide user interface for checking JUnit tests using UnitCheck – simple command-line application, Ant task, and Eclipse plugin. The interfaces are designed to make their usage as close to the usage of the corresponding JUnit tools as possible. As an example, the Eclipse plugin provides a user interface very similar to the JUnit plugin for Eclipse, including test summaries, progress reports, and run configurations. On the other hand, the tools provide also a verbose output for the expert users which are familiar with model checking tools. In addition to the easy-to-understand JUnit-like result summaries, the Eclipse plugin provides also a navigable panel with a detailed error trace obtained from JPF with different levels of verbosity.[1]

[1] The UnitCheck tool and all three user interfaces are available for download at `http://aiya.ms.mff.cuni.cz/unitchecking`

3 Related work

The notion of unit checking was first used in [10]. The authors study the problem of symbolic model checking of code fragments (e.g., individual functions) in separation. In a similar vein, we use the term unit checking in parallel to unit testing. However, the focus of UnitCheck is in providing users with tool support for integration of model checking into the software development process.

To our knowledge, the Slam project [2] is by far the most successful application of code model checking in real-life software development process. Nevertheless, its success stems from the very constrained domain of device drivers, where the environment is fixed and known in advance by the tool's developers. In contrast, UnitCheck offers benefits of code model checking in a convenient (and familiar) interface to developers of general purpose Java software.

Another related project is Chess, which is a testing tool that can execute the target .NET or Win32 program in all relevant thread interleavings. Chess comes in a form of a plugin into Microsoft Visual Studio and can be used to execute existing unit tests. As well as with UnitCheck, the user of Chess is not forced to change his/her habits with unit testing and gets the benefit of test execution under all relevant thread interleaving for free. In contrast to UnitCheck, Chess cannot cope with random values in tests, because it uses a layer over the scheduler-relevant API calls. The presence of a random event in a test would result in the loss of error reproducibility.

Orthogonal to our work is the progress on generating test inputs for unit tests for achieving high code coverage [9, 16, 17]. To name one, the Pex tool [14] uses symbolic execution and an automated theorem prover Z3 [12] to automatically produce a small test suite with high code coverage for a .NET program. We believe that similar techniques can be used in synergy with unit checking.

There are other approaches for assuring program correctness than code model checking. Static analysis tools (e.g., Splint [8]) are easy to use and scale well. However, there is typically a trade off between amount of false positives and completeness of such analysis. The design-by-contract paradigm (e.g., JML [4], Spec# [3]) relies on user provided code annotations. Specifying these annotations is a demanding task that requires an expertise in formal methods.

4 Conclusion

We presented the UnitCheck tool that brings the benefits of code model checking to the unit testing area the developers are familiar with. Of course, not all tests are amenable for unit checking. Only tests for which the standard testing is not complete (i.e., tests that feature random values or concurrency) would benefit from exhaustive traversal using UnitCheck[2]. As UnitCheck accepts standard JUnit tests, developers can seamlessly switch among the testing engines as necessary.

[2] Of course, only the (increasing) portion of the standard Java libraries supported by JPFcan be used in the tests.

References

1. JUnit testing framework, `http://www.junit.org`.
2. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, 2006.
3. M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The spec# programming system: Challenges and directions. In B. Meyer and J. Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer, 2005.
4. P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with jml and esc/java2. In *Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer.
5. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.
6. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
7. N. H. M. A. de Brugh, V. Y. Nguyen, and T. C. Ruys. Moonwalker: Verification of .net programs. In S. Kowalewski and A. Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 170–173. Springer, 2009.
8. D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.
9. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, USA, 2005. ACM.
10. E. L. Gunter and D. Peled. Unit checking: Symbolic model checking for a unit of code. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 548–567. Springer, 2004.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *SIGPLAN Not.*, 37(1):58–70, 2002.
12. L. D. Moura and N. Bjorner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer.
13. J. T. Mühlberg and G. Lüttgen. Blasting linux code. In L. Brim, B. R. Haverkort, M. Leucker, and J. van de Pol, editors, *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2006.
14. N. Tillmann and J. de Halleux. Pexwhite box test generation for .net. In *2nd International Conference on Tests and Proofs*, pages 134–153, April 2008.
15. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.
16. W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 97–107, New York, USA, 2004. ACM.
17. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 365–381. Springer.