AOP-based automated unit test classification of large benchmarks

Cyrille Artho
Research Center for Information Security (RCIS),
National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Zhongwei Chen
Kosesoft Inc., Tokyo, Japan

Shinichi Honiden
National Institute of Informatics, Tokyo, Japan

Abstract

Despite the availability of a variety of program analysis tools, evaluation of these tools is difficult, as only few benchmark suites exist. Existing benchmark suites lack the uniformity needed for automation of experiments. We introduce the design of a uniform build/installation platform, which constitutes an important part of the solution.

This platform is used to manage the build and test process, which is enhanced by a tool that analyzes the structure of unit tests. Benchmark applications lack detailed information about unit tests. Such knowledge is useful: For analysis algorithms that target specific program features, it is desirable to analyze only relevant tests. Using aspect-oriented programming, we wrap test execution and implement a tool providing coverage data of individual unit tests. Furthermore, the wrapper provides a front-end for the selection of subsets of a test suite. We successfully applied our tool to several large programs. This evaluation also gave us interesting insights about the quality of different test suites.

1. Introduction

There is a consensus in the run-time analysis community that the lack of common benchmarks makes evaluation and comparison of tools difficult [6, 7, 19]. One reason for this is the fragmentation of tools into different programming languages. It also is hard to obtain commercial programs for benchmarking, as even older software still has commercial value. Another major problem is the fact that tools cover a variety of purposes, such as analysis of data races [5, 17], general concurrency problems [8], contracts [13], fault injection [2], incorrect memory or resource usage [14, 15], or logic specifications [4], just to name a few.

Researchers would like to evaluate their tools on a large set of realistic applications. Existing benchmark suites contain small programs, of less than 1000 lines of code, which were produced by students in class room assignments. [6, 7]. We believe these examples are very useful to show the classification of failures, and for initial testing. Nonetheless, for a more serious evaluation of tools, larger programs are needed.

The Internet is a great resource of openly available applications [2]. However, such applications come with no documentation that relates them to the features investigated by a particular program analysis tool. Hence, many applications are not interesting for particular analysis: For instance, sequential programs (or test cases involving only one thread) will never exhibit data races. As information about the usage of particular programming constructs is primarily useful for tool developers only, one cannot expect applications to be documented in this way. Therefore, such information has to be reverse engineered. We propose a tool, which gathers information about all unit tests, and filters out tests of interest. Other analysis tools can then be applied to a subset of applications and unit tests, yielding interesting results quickly, even for analysis algorithms that impose a large overhead when applied to the entire test suite.

Our analysis tool is AOP-based and targets individual test cases and measures the usage of particular features, such as threads and I/O, of a programming environment. This differs significantly from traditional coverage tools, which measure execution of statements and branches by an entire run or test suite [9]. Furthermore, our tool uses dynamically gathered program information for test case selection, which differs from previous tools that relied on static criteria to select tests [3, 10, 16, 18].

This paper is organized as follows: Section 2 describes adaptation of benchmarks to a standardized file layout and build process. Wrapping of test execution, and AOP-based evaluation of per-unit-test coverage, is shown in Section 3. Section 4 shows results found in example applications. Section 5 introduces related work, and Section 6 concludes.

Directory name	Contents
proj	Top directory
proj/INSTALL	Installation instructions
proj/INSTALL.orig	Original installation guide, if any
proj/LICENSE	License (original file, renamed or converted to text if necessary)
proj/README	New README file
proj/README.orig	Original README file, if any
proj/build.properties	Specific build configuration
proj/build.xml	Generic ant build file
proj/docs	Documentation of project
proj/lib	Required libraries (e.g. jar files)
proj/src	Source code
proj/test	Source code of unit test classes

Table 1. File layout for application sources.

2. File layout and build process

In order to be useful, benchmarks have to be easy to evaluate. For a benchmark suite of non-trivial programs, management of the compilation, installation and test process becomes a challenge, because of dependencies on dozens of third-party libraries, and the different steps required for compilation and execution. A standardized file layout solves this problem and allows for automation of the build process across benchmarks. Our standardization has been applied successfully to eight benchmark programs, which were mostly taken from previous cases studies [2]. The size of these programs ranges from 1,100 to 77,000 LOC; five programs require between 23 and 33 external libraries.

2.1. Standardization

We use a generic build file, which is based on ant and easily customized for a project using a configuration file called build.properties.[1] Besides the benefit of having a uniform build and test process, this adaptation makes it easy to use our AOP-based analysis tool. We chose the layout described in Table 1, as it corresponds to the structure of many existing projects. Indeed, many benchmark programs from previous case studies [2] required only few changes.

Because the build and test process should also be evident to people who have not read this paper, two descriptive files are added to each archive, providing an overview:

1. README – project overview
   This file contains the project title, a short description of the project, information about the author(s), version, data, and home page (if available). A list of known bugs (that could affect running the tests or evaluation by verification tools) closes the document.
2. INSTALL – installation guide, covering the following issues:
   (a) Unpacking the sources.
   (b) Additional libraries required for compiling the sources (build dependencies).
   (c) How to build the program (standardized build process).
   (d) Additional libraries required for running the program (run-time dependencies).
   (e) How to execute the program and its tests.

Both text files are completely standardized across all projects. We also provide a working version of all third-party libraries with our set of benchmarks. Duplication of libraries is avoided through symbolic links pointing to a shared library used across all benchmarks. Thus, installation of third-party libraries is not necessary.

Adaptation of the build process is necessary for our evaluation tool, but could be done without using our generic build file. As our tool is AOP-based, compilation has to be changed (from javac to acj). Furthermore, command-line options control test filtering and evaluation. These adaptations could be performed manually for each project, without using our generic file. Alternatively, a build file manipulation tool, such as Makao [1], could be used to update all build files automatically.

2.2. Comparison to package managers

Our build file automates compilation, installation, and execution of benchmark programs. Existing package managers also fulfill these tasks (except for the execution of the unit test suite), so the question arises whether the ant-based solution for Java is ideal. We believe that our solution makes sense, because package managers have fundamentally different goals:

- Package managers typically install and update applications system-wide. Benchmark programs are not used outside the purpose of testing and should not interfere with the rest of the system.[2]
- Package managers have sophisticated mechanisms for maintaining dependencies of libraries, and to update these to the latest version when necessary and desired. However, benchmark programs should never be updated (their bugs serve as measures of the quality of analysis tools).

[1] Other programming languages have similar de-facto standard build tools, such as make. While we used ant for Java, our concepts can be easily applied to other platforms.

[2] This point could be addressed by creating a second repository of installed packages, which only holds benchmark programs.

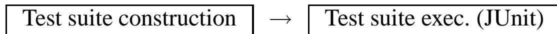

Figure 1. Standard flow of running tests in JUnit.

- Package managers are typically only well supported on platform, while build tools are much more portable.
- The adaptation of a uniform file layout has many advantages besides eliminating the need for complex configuration management. Analysis of source and object code are greatly simplified as well.

3. Test execution and evaluation

Some run-time analysis tools impose a considerable overhead. Memory analysis tool "valgrind" has an average run-time overhead of roughly a factor of 20 [15]. This is acceptable for small tests that run within milliseconds. Large tests may be many orders of magnitudes slower [3] – too slow to be executed on a frequent basis with such an expensive analysis tool. Automated analysis of unit tests w. r. t. run-time and features relevant to program analysis lets the developer quickly identify a small subset of tests. Execution of that subset may be sufficient to detect failures, at a fraction of the time required to execute all unit tests. For instance, tests which do not allocate certain data structures, or which do not use locking, can be excluded from certain types of analysis because the cannot produce the types of faults investigated. Alternatively, slow tests may be excluded because their run-time becomes prohibitive when factoring in the slowdown of a program analysis tool.

Our work is based on JUnit as the test framework [12]. The concepts generalize to other unit test frameworks and have been inspired by the unit test framework of JNuke [3]. In any modern unit test framework, a test suite consisting of individual test cases is constructed before it is executed. Construction may entail manual registration of test cases, or it may be automated through reflection or annotations. After a test suite is constructed, a test driver then runs the test suite (see Figure 1). We extend this process to gather coverage data and filter tests before they are executed. The remainder of this paper describes the architecture and implementation of a tool that automates this task.

Our tool fulfills two goals: Test filtering, and test evaluation (gathering of coverage data). Both goals can be achieved by one design, as both tasks involve wrapping each test. JUnit offers an extension called `TestDecorator` for this purpose [12]. Figure 2 shows how the test process is extended by a middle stage, which wraps and filters tests. The wrapper also maintains coverage data, which is displayed after completion of a test suite.

Category	Instrumentation with AOP
Thread usage	Calls to Thread.start()
Client socket	Calls to Socket.connect()
Server socket	Calls to ServerSocket.accept()
File, read-only	Instantiation of FileInputStream
File, (read-)write	Instantiation of FileOutputStream
File, random-access	Instantiation of RandomAccessFile
Console, out/error	Usage of System.out, System.err
Other I/O	Creation of PipedIn/OutputStream

Table 2. Point cuts for accumulating test data.

We implemented test wrapping by using our own test runner. The custom test runner wraps all incoming tests on the fly before passing the resulting test suite to the original JUnit test runner. After test execution, coverage data is shown. Wrapping incoming tests requires only a single change in the caller of the test suite, so AOP is not needed for this part. However, accumulation of test data cannot be done without extensive code instrumentation. We chose to use the AspectJ compiler [11] to insert coverage measurement code. The type of coverage that was interesting for our particular case study was the usage of threads and I/O operations per unit test.

Coverage measurement is not very complex, but differs from conventional coverage measurement. Because we are interested in data *per unit test,* this requires differentiation between each test. Our code uses an elegant design to avoid the need of a reference to coverage data inside application code: Coverage data of the current test is stored in a singleton instance. Before each unit test is run, that data is reset to zero. At the end of each test, the singleton instance is cloned and added to the database of coverage data. During test runs, the advice instrumented by AOP modifies the singleton instance by calling static methods that access it. This keeps the coverage measurement code simple and fast.[3]

Coverage analysis captures crucial operations involving I/O and concurrency (see Table 2). AspectJ allowed for an elegant implementation. Coverage of other features can be measured by adding a new declaration about coverage information, and by creating a new point cut for the AspectJ specification. The coverage framework is generic enough such that other steps are already taken care of. Figure 3 illustrates how the number of threads activated by a unit test is measured. Any other work (maintenance and display of coverage data) is taken care of by our coverage framework.

[3]Coverage data can be updated anywhere within an application. If a non-static reference to coverage data were used, it would have to be carried along each method call, creating a major overhead for instrumentation and at run-time. A static reference hides that singleton instance and is globally available.

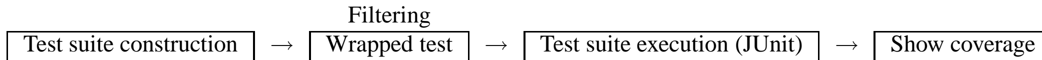

Figure 2. Extended test runner, adding filtering and coverage evaluation.

```
pointcut thread_start():
  call(* Thread.start(..))
after(): thread_start() {
  Coverage.incCounter(Coverage.Cov_Thread);
}
```

Figure 3. Aspect for measuring thread usage.

4. Experimental results

In our case studies, we used five applications that were part of a larger benchmark suite [2]. It could be confirmed statically that applications as a whole use multiple threads as well as network I/O. We were interested in which unit tests use this particular functionality. In order to extract this information, we applied our analysis tools to the five largest benchmarks that used JUnit version 3.8.1, which was the most current version when the tool was written. Table 3 summarizes the results, providing aggregated information from individual tables. It firsts lists the applications with their size, then shows the number of tests. The last three columns show how many unit tests use multiple threads, network I/O, and file I/O.

Table 3 has been derived from the output of our tool. Our tool provides much more detailed information, consisting of data per unit test. For each test, execution time and coverage information w. r. t. usage of certain features is provided. For brevity, we only show an excerpt of one of these tables generated. Table 4 lists all unit tests from jConfig which use threads, sockets, or files, and shows the number of these items used. Unit test names have been abbreviated where necessary. Note that this list allows someone to quickly identify and selected potentially useful unit tests for detailed analysis. For instance, for deadlock analysis, only unit tests with at least one extra thread (16 tests out of 97) are interesting.

Our coverage tool itself had no measurable overhead, as only minimal code instrumentation is required. The experiments managed to filter out a small subset of interesting test cases, and lead to interesting observations:

- Socket usage implies thread usage. This is because a socket cannot be tested without its counterpart; therefore, socket usage requires creation of another thread to act as a client or server.

- Thread usage often implies socket or file usage: Test threads were usually used in conjunction with I/O.

- Printing to the screen in unit tests is rare, as such output is not validated by the JUnit test harness.

- The Enforcer fault injection tool could not use the test suite of STUN, because it provided no coverage of any interesting features [2]. However, the tool found bugs in Echomine and jConfig. The low feature usage metrics in some of the other test suites hint that they do not test the program properly. Thus, the absence of bugs found in those cases is likely a result of low coverage. It is difficult to write unit tests for network I/O, as such tests have to be concurrent. This partially explains the absence of such unit tests, even though the programs as a whole heavily use these features.

5. Related work

Conventional coverage measurement tools work on a set of hard-coded low-level program features, such as statements and branches. Such tools do not allow for custom-specified coverage criteria.

Few tools that can measure custom-specified criteria exist, such as COMET, which provides first-order temporal logic predicates to specify coverage of hardware designs [10]. For software verification, AspectCov for C# [16] uses an AspectJ-like point cut language for specification of coverage criteria. Its goal is to provide custom-tailored coverage for an entire test suite. Unlike our tool, AspectCov does not provide per unit-test coverage. The custom point cut language of AspectCov encompasses lower-level properties, such as branches. Conventional AOP languages, e. g. AspectJ [11], hide such features.

Selection of a subset of all test cases for testing has been implemented in different ways before. The JNuke test framework uses static annotation about the performance of unit tests [3]. The FEAT tool allows for construction of concerns that select a set of test cases [18]. Our tool differs in that it selects test cases based on quantitative criteria (such as the number of threads used) rather than just qualitative criteria (e. g., whether any threads are used). The key difference lies in the fact that our tool uses dynamically measured information (provided by an initial test run) to select a subset of test cases for more extensive analysis. Previous tools relied on compile-time information alone. Our approach is therefore more flexible but requires an initial execution step of the entire test suite, making our approach suitable for complementing heavy-weight run-time analysis tools.

Application or library	Description	Size [LOC]	# unit tests	# threads	# sockets	# files
Echomine	Communication services API	14331	170	12	12	0
jConfig	Configuration library	9611	97	16	1	28
jZonic	Caching library	2142	16	0	0	1
SixBS	Java beans persistency	4666	30	20	0	0
STUN	Extensible programming system	1706	14	0	0	0

Table 3. Functionality used by all unit tests combined.

#	Test name	Thread	Socket	SrvSock	FileRO	FileW	FileRA	SysOut	SysErr	Other
20	>ig.ConfigurationTest.testGetConfigName	2	0	0	5	0	0	0	0	0
21	>nfig.ConfigurationTest.testSetCategory	0	0	0	1	0	0	0	0	0
34	>rationManagerTest.>AddPropertyListener	4	0	0	3	0	0	0	0	0
35	>rationManagerTest.>tInputStreamHandler	4	0	0	3	0	0	0	0	0
38	>andler.InputStreamHandlerTest.testLoad	0	0	0	2	0	0	0	0	0
39	>handler.PropertiesHandlerTest.testLoad	0	0	0	1	0	0	0	0	0
40	>.PropertiesHandlerTest.testFileChanged	4	0	0	2	0	0	0	0	0
41	>ler.XMLFileHandlerTest.testLoadAndSave	0	0	0	4	0	0	0	0	0
45	>config.handler.URLHandlerTest.testLoad	1	0	1	2	0	0	0	0	0
74	>g.jconfig.LoadSaveTest.testGetInstance	4	0	0	4	0	0	0	0	0
75	>nfig.LoadSaveTest.>AndSaveConfigEscape	4	0	0	6	0	0	0	0	0
76	>aConfigParserTest.>estParseCDataConfig	2	0	0	2	0	0	0	0	0
77	>dConfigParserTest.>stParseNestedConfig	2	0	0	3	0	0	0	0	0
80	>dConfigParserTest.>aveLoadNestedConfig	0	0	0	2	0	0	0	0	0
82	>onfig.handler.JDBCHandlerTest.testLoad	0	0	0	2	1	0	0	0	0
83	>andler.JDBCHandlerTest.testLoadAndSave	0	0	0	2	1	0	0	0	0
84	>er.JDBCHandlerTest.testReplaceVariable	0	0	0	2	1	0	0	0	0
85	>.handler.JDBCHandlerTest.testVariables	0	0	0	2	1	0	0	0	0
86	>r.JDBCHandlerTest.>eAndReplaceVariable	0	0	0	4	1	0	0	0	0
87	>ler.JDBCHandlerTest.testRemoveProperty	0	0	0	4	1	0	0	0	0
88	>andler.JDBCHandlerTest.testAddProperty	0	0	0	4	1	0	0	0	0
89	>g.IncludePropertiesTest.testGetInclude	2	0	0	3	0	0	0	0	0
90	>.InheritanceParserTest.testParseConfig	2	0	0	2	0	0	0	0	0
91	>ritanceParserTest.>eCircularDependency	4	0	0	4	0	0	1	0	0
92	>onfigCategoryTest.>estGetCategoryNames	2	0	0	3	0	0	0	0	0
93	>stedConfigCategoryTest.testGetCategory	2	0	0	3	0	0	0	0	0
94	>g.InheritanceTest.>GetAllCategoryNames	2	0	0	3	0	0	0	0	0
97	>jconfig.InheritanceTest.testSaveConfig	2	0	0	1	0	0	0	0	0

Table 4. Per-unit-test coverage for jConfig (where non-zero).

6. Conclusion and future work

Creation of a large benchmark suite provides many challenges, despite the abundance of open source programs. We think that a standardized file layout is necessary to create an easy-to-use collection of large benchmark programs. We have successfully applied this approach to several realistic benchmark programs.

For verification tools that impose a major overhead, it is desirable to select interesting and fast test cases before using a heavy-weight tool. We propose an AOP-based tool that gathers this information at run-time, at no significant overhead. AOP provided very elegant and easy-to-use ways of adding the necessary coverage measurement code. Together with a run-time library, the entire framework is very flexible and extensible. The results gathered by our tool can be used to select test cases of choice, and also give useful insights into the quality and scope of a test suite.

Future work consists of making the benchmark suite available for public download, and enhancements of the analysis tool with more features. Finally, the feedback from the analysis stage currently has to be used manually (for selecting sets of test cases). This task could be automated by integration of coverage evaluation into test case selection.

References

[1] B. Adams. Made in MAKAO. In *Proc. 3rd European Workshop on Aspects in Software (EWAS 2006)*, pages 41–42, Enschede, Netherlands, 2006. University of Bonn, Technical Report IAI-TR-2006-6.

[2] C. Artho, A. Biere, and S. Honiden. Enforcer – efficient failure injection. In *Proc. FM 2006*, Canada, 2006.

[3] C. Artho, A. Biere, S. Honiden, V. Schuppan, P. Eugster, M. Baur, B. Zweimüller, and P. Farkas. Advanced unit testing – how to scale up a unit test framework. In *Proc. AST 2006*, Shanghai, China, 2006.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based run-time verification. In *Proc. VMCAI 2004*, volume 2937 of *LNCS*, pages 44–57, Venice, Italy, 2004. Springer.

[5] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In *Proc. RV 2006*, volume 4262 of *LNCS*, pages 193–208, Seattle, USA, 2006. Springer.

[6] Y. Eytani, K. Havelund, S. Stoller, and S. Ur. Toward a framework and benchmark for testing tools for multithreaded programs. *Concurrency and Computation: Practice & Experience*, 19(3):267–279, 2006.

[7] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proc. IPDPS 2004*, page 266a, Santa Fe, USA, 2004. IEEE Computer Society Press.

[8] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. IPDPS 2003*, page 286, Nice, France, 2003. IEEE Computer Society Press.

[9] N. Fenton and S. Pfleeger. *Software metrics (2nd Ed.): a rigorous and practical approach*. PWS Publishing Co., Boston, USA, 1997.

[10] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv. User defined coverage – a tool supported methodology for design verification. In *Proc. DAC 1998*, pages 158–163, 1998.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 2001.

[12] J. Link and P. Fröhlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers, Inc., 2003.

[13] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[14] N. Mitchell and G. Sevitsky. LeakBot: an automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proc. ECOOP 2003*, volume 2743 of *LNCS*, pages 351–377. Springer, 2003.

[15] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proc. RV 2003*, volume 89 of *ENTCS*, pages 22–43, Boulder, USA, 2003. Elsevier.

[16] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *Proc. AOSD 2005*, pages 181–191, New York, USA, 2005. ACM Press.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[18] A. Souter, D. Shepherd, and L. Pollock. Testing with respect to concerns. In *Proc. ICSM 2003*, pages 54–63, Washington, USA, 2003. IEEE Computer Society.

[19] C. von Praun and T. Gross. Object-race detection. In *Proc. OOPSLA 2001*, pages 70–82, Tampa Bay, USA, 2001. ACM Press.