Combining Unit Tests for Data Race Detection

Jochen Schimmel*, Korbinian Molitorisz*, Ali Jannesari†‡, Walter F. Tichy*
*Karlsruhe Institute of Technology (KIT), Germany
{schimmel, molitorisz, tichy}@kit.edu
†German Research School for Simulation Sciences, Aachen, Germany
‡RWTH Aachen University, Aachen, Germany
{a.jannesari}@grs-sim.de

***Abstract*—Multithreaded programs are subject to data races. Data race detectors find such defects by static or dynamic inspection of the program. Current race detectors suffer from high numbers of false positives, slowdown, and false negatives. Because of these disadvantages, recent approaches reduce the false positive rate and the runtime overhead by applying race detection only on a subset of the whole program. To achieve this, they make use of parallel test cases, but this has other disadvantages: Parallel test cases have to be engineered manually, cover code regions that are affected by data races, and execute with input data that provoke the data races.**

This paper introduces an approach that does not need additional parallel use cases to be engineered. Instead, we take conventional unit tests as input and automatically generate parallel test cases, execution contexts and input data. As can be observed, most real-world software projects nowadays have high test coverages, so a large information base as input for our approach is already available. We analyze and reuse input data, initialization code, and mock objects that conventional unit tests already contain. With this information, no further oracles are necessary for generating parallel test cases. Instead, we reuse the knowledge that is already implicitly available in conventional unit tests.

We implemented our parallel test case generation strategy in a tool called *TestMerge*. To evaluate these test cases we used them as input for the dynamic race detector CHESS that evokes all possible thread interleavings for a given program. We evaluated *TestMerge* using six sample programs and one industrial application with a high test case coverage of over 94%. For this benchmark, *TestMerge* identified all previously known data races and even revealed previously unknown ones.

***Index Terms*—Data Races, Unit Testing, Multicore Software Engineering**

I. Introduction

Many developers use unit tests to find defects in their programs and for regression testing. A unit test is a small piece of code that tests a single method or functionality of a program. However, unit tests are rarely used to detect concurrency bugs such as data races or atomicity violations. If these bugs manifest depends on the schedule used during execution. Conventional unit tests might be categorized as spurious tests, which is a code smell. Therefore, unit testing is not popular for finding concurrency bugs; instead, tools such as data race detectors are used. However, most data race detectors are not sound, present many false positives or have intensive resource requirements.

To overcome these limitations, recent research combines unit testing and data race detection. Szeder [1] introduces parallel unit tests. These test cases are executed by a data race detector, not by a conventional test runner. Szeder proposes to write such test cases manually - a time intensive and error prone task. Therefore, automatic generation of parallel unit tests gets into focus [2], [3]. In this paper, we present a novel approach for automatic generation of parallel unit tests: Combining existing conventional unit tests. When a parallel test case for concurrent execution of two given methods *m1* and *m2* is required, we search available conventional test methods for *m1* and *m2*. We then combine the setup code of both source methods into a new test method, calling both *m1* and *m2* after the initialization. A data race detector can execute the new parallel test case.

Using our approach, the knowledge of the test case authors about the program under test is reused: We do not need oracles for input values, we do not have to cope with external resources, and faking code from the source test methods is reused.

The paper is structured as follows: Section II presents parallel unit test and their generation by capturing executions. Section III presents our approach to combine test cases, our implementation is shown in IV. Section V shows our results. We show related work in Section VI and conclude in Section VII.

II. Parallel Unit Tests

The major difference between conventional and parallel unit tests is the number of Methods Under Test (MUTs): While conventional tests that follow accepted coding guide lines [4] contain one single MUT and verify one distinct assertion, parallel tests consist of at least two methods, depending on the number of threads that are involved in the observed scenario. Parallel tests hereby adhere to the well-known execution pattern for conventional tests *arrange-act-assert* from [5]:

1) *Arrange*: Define and initialize input variables, call required initialization methods, and set up dependencies.
2) *Act*: Execute the MUT.
3) *Assert*: Define and validate expected constraints caused by the execution of the MUT.

Although parallel unit tests follow this execution pattern, we extend the pattern and add an additional stage for safe multithreading. It is therefore called *arrange-act-wait-(assert)*:

1) *Arrange*: Define and initialize input variables and dependencies for all MUTs, and execute required initialization

978-1-4673-7022-6/15 $31.00 © 2015 IEEE
DOI 10.1109/AST.2015.16

code sequentially for all MUTs.
2) *Act*: Instantiate a thread for each MUT and execute it.
3) *Wait*: Wait for the termination of all threads executing MUTs.
4) *Assert*: Validate assertions. This stage is optional.

Before executing the MUT in parallel, the unit test must have performed all necessary initializations. After the execution, the test execution has to wait until all MUTs have successfully terminated. In contrast to conventional unit tests though, the assert stage is not used to decide over a successful or unsuccessful execution, so it is made optional. As a simple assertion statement cannot identify racy code, the validation of success in parallel test cases is done using data race detection. Nevertheless, assertions can still be used for conventional assertions. As parallel test cases can include assertions for correct program semantics and absence of data races, we call such test cases *enriched*.

A. Generation of Parallel Unit Tests

Currently, automatic generation tools for parallel unit tests mainly use capture and replay approaches [3] or static inspection of class structures [2]. However, these approaches have several drawbacks: Capture and replay approaches require input data that lead through program paths that contain the potential data race code sections. Static inspection on the other hand relies on oracles, so these approaches suffer from the same problems as conventional unit test generation. Our decision to reuse already existing tests as the starting point for parallel test generation implicitly solves these problems: Most unit tests are still written by hand today, especially when following test-driven methods like extreme programming. From this fact we infer that relevant input data and correct object initialization code can be found when analyzing these tests. Additionally, we make use of testing infrastructure already available and in use, which cannot be used by both aforementioned approaches. This involves these three aspects:

- Usage of test object hierarchies
- Usage of fakes
- Usage of existing assertions

a) Usage of Test Object Hierarchies: Today, many developers use patterns and well-known architecture approaches to design their unit tests [5], [4]. They create helper classes specifically designed for test cases. Such approaches make test code more readable, trustworthy, and easier to update when the code under test changes. Captured parallel codes tends to be neither readable nor easy to update. When combining hand-written sequential unit tests, we create parallel test cases that re-use helper classes and architectures.

b) Usage of Fakes: A major drawback of capture and replay approaches is that everything is captured: When an application uses objects to encapsulate database accesses or web service calls, the full state is captured. For sequential unit tests, fake objects are used to overcome such external dependencies. Such fake classes are either created by hand or by an isolation framework [6], [7]. Testers use these classes when needed. If we use the test contexts of two sequential unit tests to create a parallel test context, we automatically use these fake classes.

c) Usage of Existing Assertions: The goal of a parallel unit test is to find parallelization bugs, not to find conventional bugs. Therefore, a parallel unit test usually has no assert statements - as the decision whether a parallelization bug exists cannot be cast by an assert statement, but by an external testing tool. Therefore, most generation approaches for parallel unit tests omit any assert statements - conventional bugs may still be found using conventional unit tests. When using combinational unit test generation, we have the option to reuse the assert statements from our source test cases and create parallel tests that also check the correctness of the methods under test. We call such unit tests *enriched*.

III. Combining Unit Tests

In this section we describe, how we automatically generate parallel unit tests from existing source code.

A. Finding Methods Under Test

The input for our approach are method pairs that are executed in parallel at runtime and might contain a data race. In preliminary work [3] we implemented a tool called AutoRT that automatically identified these code sections, but it is also possible to provide them manually. In another preliminary work [8] we extended AutoRT to also detect correlated variables.
In order to create a parallel unit test for a given method pair, we need to find all conventional unit tests that test either method. Methods under test (MUTs) directly write to variables from assert statements or mock objects. We also follow naming conventions that is cited in current literature [5]. It recommends to name test methods after the method under test. We identify the MUT by taking the immediate method call before the first assert statement. For the future we will enhance this by checking whether the return value of this method call is actually used within assertions. Whenever we are unable to identify the method under test, we ask the user to add an attribute annotation to the corresponding test method.

B. Extracting and Combining Test Contexts

After the MUT has been identified, we construct the basic test corpus: We consider any statement before the MUT call to be test initialization. These statements have to be copied to the parallel test case. Any statements that follow the MUT call are required for assertion, so they may be ignored or added to the parallel test case turning it into an enriched test case. Some conventional tests contain clean up code after the MUT call, especially when using a testing framework. Such code is usually contained in special cleanup methods that can easily be identified by name.
Conventional unit tests often contain mock objects. In contrast to stub objects, mocks contain assertions that are necessary to evaluate the success of a test execution. With enriched

```
[TestMethod]
public void AddLine_ValidLine_Success()
{
    StubTextFormatService service = new StubTextFormatService();
    string line = TestData.SAMPLE_LINE_1_UNFORMATTED;
    service.FormatLineString = (x) =>
        { return TestData.SAMPLE_LINE_1_FORMATTED; };

    TextManager tm = new TextManager(service);
    tm.AddLine(line);

    Assert.IsTrue(tm.Lines.Count == 1);
}
```

```
[TestMethod]
public void AddChapter_ValidChapter_Success()
{
    StubTextFormatService service = new StubTextFormatService();
    string chapter = TestData.SAMPLE_CHAPTER_1_UNFORMATTED;
    service.FormatChapterString = (x) =>
        { return TestData.SAMPLE_CHAPTER_1_FORMATTED; };

    TextManager tm = new TextManager(service);
    tm.AddChapter(chapter);

    Assert.IsTrue(tm.Chapters.Count == 1);
}
```

Fig. 1. Two conventional unit tests. The dashed lines indicate the arrange-act-assert structure.

parallel unit tests, these mocks will deliberately be reused. For non-enriched parallel test cases, any assert statements will be removed from the mock objects.

C. Example

Figure 1 shows two test cases for a method pair that may run in parallel, so we create a parallel unit test from both tests shown in Figure 2. Both tests are split into the three parts *initialization*, *MUT call*, and *assertion*. Both conventional test cases adhere to best practice guidelines and use a single mock object (`tm`). The resulting parallel test case, however, includes both source mock objects. We assume one mock object for each MUT to be valid, as long as the objects aren't used by multiple methods at the same time. The assert statements in both conventional tests refer to different fields of the class that contain the MUT. In case the assert statements tested the value of the same field, at least one of the assert statements would fail in the parallel test case, because this value would now be influenced by two methods; the assertions might even check for different values. For this situation, it is not possible to reuse the assertions in a successful execution of the parallel test case. This explains why pure parallel test cases without enrichment are still necessary. Luckily, we can detect this situation, as it occurs when an assertion variable is written by both MUTs. Whenever we detect this situation, we execute the parallel test case and expect it to fail. We then forward this information to the engineer to correct the assertion.

D. Compatible Test Contexts

The set of method calls and objects that are required before a method under test can be executed is called the *test context*. The most complex task when merging two or more conventional test cases is merging each test's context. It is possible that we want to merge test cases that set different values to the same shared variable. If this is the case, we call the test contexts incompatible, otherwise we say the test contexts are compatible.

If contexts are compatible, it is straight forward to merge them: The new parallel context contains all statements that can be found in the original test context code. However, we have to check if an object reference from the originating contexts refers to the same object: for example, if both methods under test have to be called with the same object instance.

```
[TestMethod]
public void AddChapter_AddLine_ParallelTest()
{
    StubTextFormatService service = new StubTextFormatService();

    string line = TestData.SAMPLE_LINE_1_UNFORMATTED;
    service.FormatLineString = (x) =>
    { return TestData.SAMPLE_LINE_1_FORMATTED; };

    string chapter = TestData.SAMPLE_CHAPTER_1_UNFORMATTED;
    service.FormatChapterString = (x) =>
    { return TestData.SAMPLE_CHAPTER_1_FORMATTED; };

    TextManager tm = new TextManager(service);

    Task.WaitAll(
        Task.Factory.StartNew(() => { tm.AddChapter(chapter); }),
        Task.Factory.StartNew(() => { tm.AddLine(line); })
        );

    Assert.IsTrue(tm.Chapters.Count == 1);
    Assert.IsTrue(tm.Lines.Count == 1);
}
```

Fig. 2. Generated parallel test case based on test cases from figure 1.

If test contexts are incompatible, they may be altered to obtain compatibility: For example, if two test cases set a common global variable to different values, we might decide for one of these values for the resulting parallel test context. If this is the case, we generate a parallel test case for each combination. Alteration may break some of the assertions, and may even result in different control flow. This is not critical for parallel unit tests, as we remove assertions. For enriched unit tests, only compatible test contexts should be used.

E. Data Races and Isolation Frameworks

An advantage of combining unit tests is the implicit usage of fakes and isolation frameworks. Fake classes remove code from unit tests that is not intended to be tested. As can be seen in figure 1, a common approach is to have a fake class return expected values instead of performing time-consuming computations. This is important for unit testing: units may be tested before their dependencies are tested or implemented at all. Even worse: If the original code were executed, it would be far more complex to track down bugs, as it wouldn't be obvious if the error stems from the actual method under test or wrong dependency behavior. This is also true for parallel unit testing: A parallel unit test aims to detect whether or

TABLE I
PARALLEL TEST CASE GENERATION RESULTS OF TESTMERGE.

Program	Simple Bank	Bank	Stack	Bounded Queue	Double Wrapper	Fakes	Dynamics CRM Add-On
Lines of Code	6	14	24	29	22	13	3410
Lines of Test Code	15	12	43	36	28	23	4180
Conv. Test Cases	4	4	9	5	7	2	134
Parallel Test Cases	2	1	6	7	2	2	0
Detected MUTs	3	3	4	4	7	1	47
MUT identified in Conv. Test Cases	100%	100%	100%	100%	100%	100%	96%
Input Method Sets	5	3	7	8	6	3	15
Parallel Test Cases generated	13	6	87	62	32	72	78
Known Data Races	3	0	3	0	2	2	0
Data Races Detected	3	0	4	0	2	2	7
False Positives	0	0	0	0	0	0	2

not a race exists within the methods under test - not within their dependencies. This is a drawback of capture-and-replay approaches: they execute all external dependency code. Tools such as CHESS [9] rely on short running unit tests; calling dependency code instead of fakes leads to extended runtimes and more schedules that have to be executed - possibly rendering the test case useless.

IV. IMPLEMENTATION

We implemented the approach in C#. *TestMerge* offers two modes: *test-analysis* and *test-generation*. In test-analysis mode, TestMerge receives a Microsoft Visual Studio solution file as input. *TestMerge* identifies all unit test projects and files in the solution. We analyze all test cases found using Microsoft Roslyn [10] and identify the methods under test covered by each test method in the project. We save MUT information in a MUT-info file. We use OpenCover [11] as a coverage analysis framework to generate and append IDs for each MUT branch covered by a given test method to the MUT-file.

In test-generation mode, *TestMerge* accepts a list of method sets. Each method set may be extended by path IDs, detailing which paths in the MUTs should be reached by the combined test cases. For each set, we search test cases for these methods and generate according parallel test cases. If path IDs are included, only test methods reaching these paths are considered for combination. If no path information is specified, parallel test cases are generated to visit all possible paths - if fitting conventional test cases are available. If multiple test methods are available for a method, we generate several parallel test cases. Our tests contain necessary metadata to be executed by CHESS.

TestMerge combines test context creation code of both input test methods. The state of both source test contexts may not be fully compatible: If, for example, a singleton object has different states in both source tests, we generate two resulting parallel test methods - one with each object state. We do not take assert statements into the resulting parallel test case. However, if assert statements are included in mock methods, they will still be called by a data race detector executing our tests. As assertions are usually performed by a single class in a given unit test framework, this can easily be solved: We will mock the assert class itself in future releases to perform no action for parallel test cases.

V. RESULTS

We evaluated our implementation using different example applications from the CHESS race detector. We also used some samples written on our own. These programs are intended to verify our tool in different situations, for example valid test case generation in the presence of mock objects. As a larger example, we used an add-on for Microsoft Dynamics CRM [12]. A summary of these examples is given in table I.

The CHESS samples are small applications accompanied with conventional as well as parallel test cases. For each example, we defined a set of method pairs, for which we wanted *TestMerge* to generate parallel test cases. As the samples are small, the method pairs are obvious. Using the generated parallel test cases as input for CHESS, the races where successfully reported. After fixing the races, our unit tests did no longer report any races. We also successfully generated functional test cases with three MUTs, which might be useful for more complex parallel methods.

The *Stack* sample application does not contain intentional races. Therefore, we removed synchronization and created 3 data races. Using our generated test cases, CHESS could identify all of them. In *Bounded Queue*, we could identify all documented data races as well as an additional, undocumented data race.

The only data races that we were unable to find were atomicity violations: CHESS is not able to detect all possible atomicity violations. In the *Bank* sample, a unit test is designed to contain an assert statement to identify this violation if a certain schedule is applied - this is to verify that CHESS is able to produce such schedules; without knowing that this violation exists, no tester would have added such an assert statement. Therefore, our generated test cases cannot detect this race. However, a race detector designed to detect such violations should succeed.

As a larger example, we generated parallel unit tests using a double-opt-in newsletter registration add-on for Microsoft Dynamics CRM [12]. The add-on contains 134 conventional unit tests; it consists of 3410 code-lines accompanied by 4180 lines of test code. The test code includes faking of CRM server logic to enable test execution in absence of a CRM server. Currently, this add-on is executed using multiple threads. However, transactions deny any parallel accesses to

relevant data, restarting the add-on, if necessary. So the races we found are benign. However, due to the task based structure, true parallel execution is possible without code changes. The races we found will then require explicit synchronization.

Some samples generate many parallel test cases. For example, *Stack* generates 74 test methods. This may happen if there exists more than one unit test per method in a method pair that consists of *m1* and *m2*: In this case, we currently combine any unit test for *m1* with any unit test for *m2*. It is also possible that the context of such tests is contradictory, for example a global object might contain different values. This raises two possible combinations. We also used method sets with three and four methods, combining three or four test methods to a single parallel unit test with as many threads. At the moment, we generate all possible distinct combinations. The large number of test cases does not harm precision: no false positives are introduced. However, avoiding unnecessary test cases is best practice, so we will develop strategies for test case pruning in the future.

VI. Related Work

Ballerina [2] is a tool to automatically generate parallel unit tests for Java classes. These test cases aim to identify racy usage scenarios of the public interface of the class under test. Ballerina uses Java PathFinder for thread interleaving exploration and checks for serializability. In contrast to our work, Ballerina generates test cases using static class analysis and does not take existing unit tests into account. The test methods generated by Ballerina also use the arrange-act-wait pattern proposed in our work. As in our work, only the methods under test are executed in parallel, any other method calls are executed sequentially in front. In our case, these method calls stem from a conventional unit test, whereas Ballerina randomly generates a method call sequence and guesses input values.

In ConSuite [13], a test case is defined as a triple consisting of object creation code, a sequence of method calls on the object, and a schedule. The sequence of method calls is generated by ConSuite in such a way, that synchronization points are covered. This is accomplished using a genetic algorithm. ConSuite is only designed to test the public interface of classes. Ballerina and ConSuite could in principle also make use of existing tests using seeding [14].

Krena et al. [15] built a framework for search-based testing and apply it to data race detection by linking it to ConTest. They use the built-in metrics of ConTest to define fitness rules for schedule selection. They do not generate unit tests. However, their approach might be applied to the parallel unit tests generated in our work.

DejaVu [16] is a capture-and-replay system for Java. It ensures deterministic replay by recording results of non-deterministic events such as date functions. Thread switches are separated into deterministic and non-deterministic switches: Deterministic switches are triggered by events such as notify or monitor exit. Such events can be replayed as DejaVu records the lock state for each thread. Non-deterministic switches include timed events such as sleep or thread preemption. DejaVu can handle these switches, as they only occur on yield points, which are fixed points in execution defined within Jalapeño. DejaVu focuses on deterministic replay of a given recording, which can be seen as a test case. However, they rely to complete program execution, while we focus on unit tests. Such replay approaches may be applied to our parallel test methods.

VII. Conclusion

We presented an approach to generate parallel unit tests using existing unit tests. These test cases can be used as input for a race detector. In contrast to other approaches, we avoid the problems of realistic test context creation, as we inherit them from existing cases. We also use existing fake classes. Our evaluation showed that we can generate test cases and detect real data races in different applications. We proposed the generation of enriched parallel test cases containing assertions. Future work will investigate their utilization.

Acknowledgment

We thank Emre Kasif Selengin for his support during design, implementation and experimentation of TestMerge.

References

[1] G. Szeder, "Unit testing for multi-threaded java programs," in *PADTAD '09: Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. ACM, 2009.

[2] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: automatic generation and clustering of efficient random unit tests for multithreaded code," in *Proceedings of the 2012 International Conference on Software Engineering*, 2012.

[3] J. Schimmel, K. Molitorisz, A. Jannesari, and W. F. Tichy, "Automatic generation of parallel unit tests," in *Proc. of the 8th International Workshop on Automation of Software Test (AST), San Francisco, CA, USA*. ACM, May 2013, pp. 40–46.

[4] G. Meszaros, *xUnit Testing Patterns*. Addison-Wesley, 2013.

[5] R. Osherove, *The Art of Unit Testing*. Manning, 2013.

[6] B. Pasternak, S. Tyszberowicz, and A. Yehudai, "Genutest: a unit test and mock aspect generation tool," in *Proceedings of the 3rd international Haifa verification conference on Hardware and software: verification and testing*. Springer-Verlag, 2008.

[7] "Nsubstitute: A friendly substitute for .net mocking frameworks." [Online]. Available: http://nsubstitute.github.io/

[8] A. Jannesari, N. Koprowski, J. Schimmel, F. Wolf, and W. F. Tichy, "Detecting correlation violations and data races by inferring non-deterministic reads." in *ICPADS*, 2013, pp. 1–9.

[9] S. Q. Madanlal Musuvathi and T. Ball, "Chess: A systematic testing tool for concurrent software," Microsoft Research, Tech. Rep., Nov 2007.

[10] Microsoft. (2014) .net compiler platform ("roslyn"). [Online]. Available: https://roslyn.codeplex.com/

[11] S. Wilde. (2014) Opencover. https://opencover.codeplex.com/.

[12] Microsoft. (2015) Microsoft dynamics crm. [Online]. Available: http://www.microsoft.com/de-de/dynamics/crm.aspx

[13] S. Steenbuck and G. Fraser, "Generating unit tests for concurrent classes," in *ICST'13: Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, 2013.

[14] S. Yoo and M. Harman, "Test data regeneration: generating new test data from existing test data," *Software Testing, Verification and Reliability*, vol. 22, no. 3, pp. 171–201, 2012.

[15] B. Krena, Z. Letko, T. Vojnar, and S. Ur, "A platform for search-based testing of concurrent software." in *PDATAD*, J. Loureno, Ed. ACM, 2010, pp. 48–58. [Online]. Available: http://dblp.uni-trier.de/db/conf/issta/padtad2010.html#KrenaLVU10

[16] J.-D. Choi and H. Srinivasan, "Deterministic replay of java multithreaded applications," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, ser. SPDT '98. New York, USA: ACM, 1998, pp. 48–59.