Using Clone Detection to Identify Bugs in Concurrent Software

Kevin Jalbert, Jeremy S. Bradbury
Software Quality Research Group
Faculty of Science (Computer Science)
University of Ontario Institute of Technology
Oshawa, Ontario, Canada
{kevin.jalbert, jeremy.bradbury}@uoit.ca

***Abstract*—In this paper we propose an active testing approach that uses clone detection and rule evaluation as the foundation for detecting bug patterns in concurrent software. If we can identify a bug pattern as being present then we can localize our testing effort to the exploration of interleavings relevant to the potential bug. Furthermore, if the potential bug is indeed a real bug, then targeting specific thread interleavings instead of examining all possible executions can increase the probability of the bug being detected sooner.**

***Keywords*-active testing, bug patterns, clone detection, concurrency, fault localization, testing, static analysis**

I. Introduction

Concurrency is increasingly utilized in software applications in order to exploit the benefits of multi-core processors [1]. Concurrent software is traditionally viewed as being more difficult to program and test when compared with sequential software due to the existence of possible thread interleavings. Furthermore, concurrency bugs are difficult to identify since they often involve multiple code fragments that only exhibit incorrect behaviour during some of the thread interleavings.

Various techniques exist to detect concurrency bugs. Model checking is one technique that uses state exploration to search all path executions and finds a path that produces the concurrency bug [2]. Noise makers represent another technique that cause subtle random delays during execution in hopes of forcing unusual thread interleavings that may exhibit abnormal behaviour [3], [4]. Two criteria that are often used to assess concurrency bug detection techniques are effectiveness and efficiency. Effectiveness represents the ability of a given technique to find bugs in a software system. Efficiency represents the amount of effort and resources required to find these bugs. Exhaustive state exploration and the use of noise makers can usually be classified as having high effectiveness with low efficiency. One possible improvement to these techniques is to use pre-processing (e.g., static analysis) to reduce the search space and increase efficiency while maintaining the level of effectiveness. This approach is called active testing and operates by first localizing the potential faults and then specifically testing them [5].

We propose an approach to active testing that is based on localizing the testing effort by identifying potential bugs using clone detection. Our approach uses previously identified bug patterns as input to find similar bug patterns in concurrent software. The identification of cloned source fragments from the clone detection process makes it possible to identify the location of potential concurrency bugs. If these potential bugs can be identified then it becomes possible to localize the testing effort to a specific section of the source code. For example, in the case of noise makers we can localize the insertion of delays around the bug's code fragment locations. In general, the aim of our work is to increase the efficiency of existing testing tools by providing potential locations of concurrency bugs; the first phase of active testing.

In Section II we present background information on detecting bugs using clone detection and existing concurrency bug detection tools. The formal specification of concurrency bug patterns and an example of a bug pattern specification follows in Section III. Section IV outlines our approach for the identification of potential bugs. Specifically, we overview our tool's architecture as well as the overall detection process. Our proposed evaluation is described in Section V and finally in Section VI we present our conclusions and future work.

II. Background

A. Detecting Bugs using Clone Detection Tools

Clone detection is a technique that is capable of detecting similar code fragments in source code. These clones are not restricted to exact matches and thus source code normalization is often used to generalize the input. Additionally, objects in the source code can be abstracted to identifiers so that two code fragments can match even though the objects within them are different. By using these techniques it becomes possible to detected the following clone types [6]:

- Type I - *"Identical code fragments except for variations in whitespace, layout and comments."*
- Type II - *"Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments."*
- Type III - *"Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments."*

- Type IV - *"Two or more code fragments that perform the same computation but are implemented by different syntactic variants."*

Several tools use clone detection to help detect bugs in sequential code: CP-Miner [7], CEeN [8] and an approach using context-based detection of clone-related bugs [9].

B. Detecting Concurrency Bugs

Concurrency involves two or more threads running in parallel and these threads are executing statements that can interact with each other through shared data. These interactions allow for collaboration between multiple threads and can increase the performance of the software when programmed correctly. One issue with the usage of concurrency in software is that it produces many different thread interleavings instead of a single execution path. For example, the scheduler, the number of processors, the operating system and other executing processes can all affect the interleavings.

It is common to safe guard shared objects by using synchronization idioms to lock out other threads from accessing a critical region until the current thread is finished. Synchronization helps alleviate the issue of data inconsistency, although it also creates additional concurrency bugs (e.g., deadlock) if applied incorrectly. Numerous variations of concurrency bugs exist, however almost all can be categorized into general anti-patterns [10].

The two most common concurrency bugs are data races and deadlocks. A data race occurs when there is inconsistent synchronization around an object that is shared with two or more threads (see Figure 1). A deadlock occurs when two or more threads are holding locks that each other needs in order to proceed and neither will release it's lock (see Figure 2).

The challenge of detecting concurrency bugs has led to the creation of new techniques and tools to aid in testing concurrent software. Most tools test for concurrency bugs through exploration of the interleaving space using model checking, custom thread schedulers, and the insertion of noise (i.e., delays). For example, NASA's Java PathFinder [11] is a model checking tool that exhaustively search the explicit state space of a concurrent software system in order to detect states that exhibit a concurrency bug. Microsoft's CHESS [12] is a similar tool to Java PathFinder, but handles the C++ programming language. IBM's ConTest [4] is a noise maker tool that instruments Java bytecode with random thread delays in order to encourage different thread interleavings that may expose a concurrency bug. ConFuzzer [5] is an interesting approach since it makes use of active testing.

Code fragment #1:

```
var1 = obj.read();
```

Code fragment #2:

```
obj.write( var1 );
```

Fig. 1. Data race concurrency bug

Code fragment #1:

```
synchronized ( lock1 ){
  synchronized ( lock2 ){
    var1 = obj.read();
  }
}
```

Code fragment #2:

```
synchronized ( lock2 ){
  synchronized ( lock1 ){
    obj.write( var1 );
  }
}
```

Fig. 2. Deadlock concurrency bug

III. Specification of Bug Patterns

Concurrency bug patterns can be general and represent well known concurrency problems (e.g., classic deadlock). Bug patterns can also be domain- or application-specific and represent previously found concurrency bugs (e.g., an application-specific atomicity violation that was detected in a previous iteration of regression testing). A formal specification of bug patterns in XML is shown in Figure 3 and Figure 4. In each case the specification of the bug pattern contains the following information:

- **Type:** the type or name of the bug pattern (e.g., classic deadlock).
- **Tester:** the name of the tester who created and/or maintains the bug pattern.
- **Description:** information about the bug pattern including an overview of the pattern, the behaviour it exhibits, when the bug occurs and more.
- **Solution:** in the case of a previously found bug or a general bug pattern it is useful to include information on how to fix the bug if it is detected.
- **Original Fragment(s):** the required code fragments for this bug pattern to exist. Concurrency bugs are typically composed of multiple code fragments in different threads and it is the interaction of these fragments that produces the unwanted behaviour. Individually it may be impossible to determine if a code fragment actually contains a bug.
- **Term(s):** occurs in fragments and follow the naming convention:

  `<fragmentId>.<term>`

  A term is a object and will be used in the specification of a bug pattern rule (discussed next). Examples of terms are shared variables and lock objects.
- **Rule:** defines the interaction that occurs between multiple code fragments in order to produce the unwanted behaviour of the bug pattern. Specifically, a rule is a logical statement that must be satisfied to show that the interaction between fragments' terms may potentially cause abnormal behaviour. Rules are critical to our bug identification technique since clone detection will most likely find non-exact clones, which can result in textual differences of the detected code fragments and terms.
  Rules have limited expressiveness and only allow for simple relations using Boolean logical operators (||, &&, ==, !=, !). Our current approach uses the names of terms within the rule system. However, the same object may have several names, that is several different variables may reference the same object. We plan to address the limitation in the future by adding additionally analysis to achieve true object equivalence. To increase

```
<bugPattern id="0" sourcePath="/bp/bug_pattern_0.xml">
  <type>Data race – no locks</type>
  <tester>John Smith</tester>
  <description>This bug exhibits inconsistent data of
      the obj object.</description>
  <solution>Synchronize both code fragments.</solution>
  <originalFragment sourcePath="/src/bp_code/
      bug_pattern_code_0_0.java" countLines="0"
      patternId="0" fragmentId="0">
    <term id="F0.obj" line="0" tokenPosition="3"/>
  </originalFragment>
  <originalFragment sourcePath="/src/bp_code/
      bug_pattern_code_0_1.java" countLines="0"
      patternId="0" fragmentId="1">
    <term id="F1.obj" line="0" tokenPosition="0"/>
  </originalFragment>
  <rule>(F0.obj == F1.obj && !F0.obj.IS_SYNCHED &&
      !F1.obj.IS_SYNCHED)</rule>
</bugPattern>
```

Fig. 3. Data race bug pattern

Figure 3 shows a complete data race bug pattern using the code fragments from Figure 1.

```
<bugPattern id="1">
  ...
  <originalFragment fragmentId="0">
    <term id="F0.lock1"/>
    <term id="F0.lock2"/>
  </originalFragment>
  <originalFragment fragmentId="1">
    <term id="F1.lock2"/>
    <term id="F1.lock1"/>
  </originalFragment>
  <rule>(F0.lock1 == F1.lock1 && F0.lock2 == F1.lock2)
  </rule>
</bugPattern>
```

Fig. 4. Deadlock bug pattern

Figure 4 shows a deadlock bug pattern in a reduced form, using the code fragments from Figure 2. In this figure we have excluded all bug pattern information except the code fragments and the rule.

the expressiveness of rules the concept of applying special properties to terms was necessary. Properties of terms within rules follow the naming convention:

`<fragmentId>.<term>.<property>`

Currently there is only one property available – the `IS_SYNCHED` property. This property adds additional context to the rule and checks to see if the term (i.e., object) is guarded within a synchronization block or method. In the future additional properties maybe used to provide even more expressiveness to the rules.

IV. Detection of Potential Bugs

The goal of our research is to develop an active testing approach for concurrent software using clone detection to pre-process the source code. Clone detection localizes the search space around code fragments that match existing bug patterns. This effectively reduces the number of thread interleavings that need to be explored during testing. We believe this approach may be ideal for regression testing of concurrency system, in order to ensure that previous found bugs do not reappear. We will now describe the architecture of our detection tool as well as the process that is involved to identify potential concurrency bugs using clone detection.

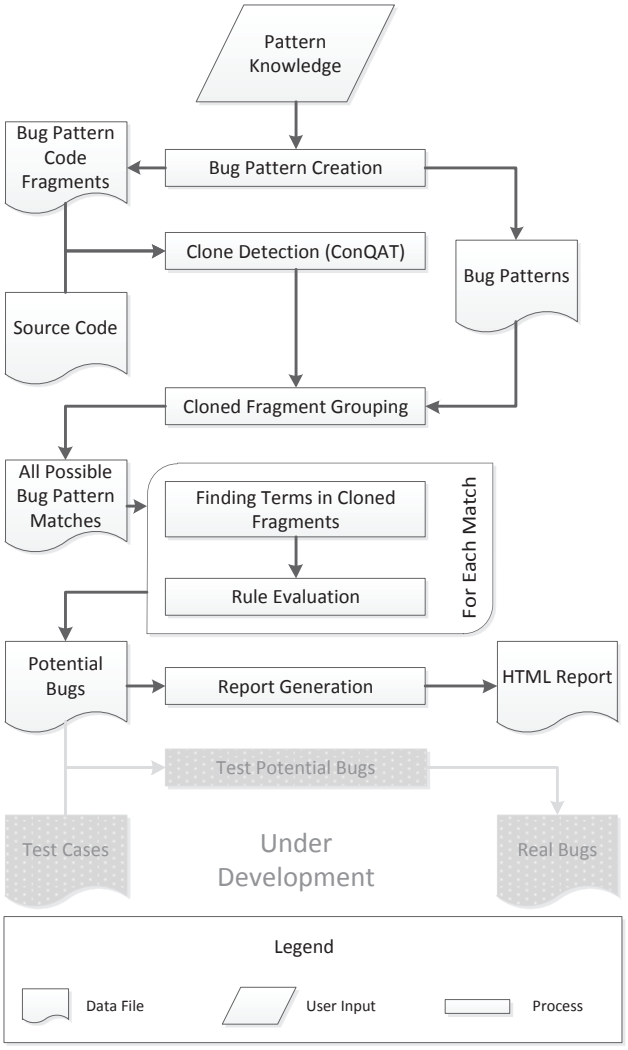

Fig. 5. Concurrency Bug Detection Process

Our tool is designed to integrate with ConQAT, an open source quality assurance framework that includes a customizable clone detection algorithm [13], [14]. ConQAT's clone detection is ideal for our research, however it is not the only available tool we could have used – it is simply the one we chose.

Our detection process is shown in Figure 5 which includes details regarding user inputs, data flow and system output. The steps in the process are as follows:

- **Bug Pattern Creation:** Bug patterns are created using knowledge of general bug types or application-specific bugs. The creation of a bug pattern will produce two outputs: a bug pattern XML file as well as the code snippet files that contain the bug pattern's code fragments. Bug patterns are specified by the tester using our Bug

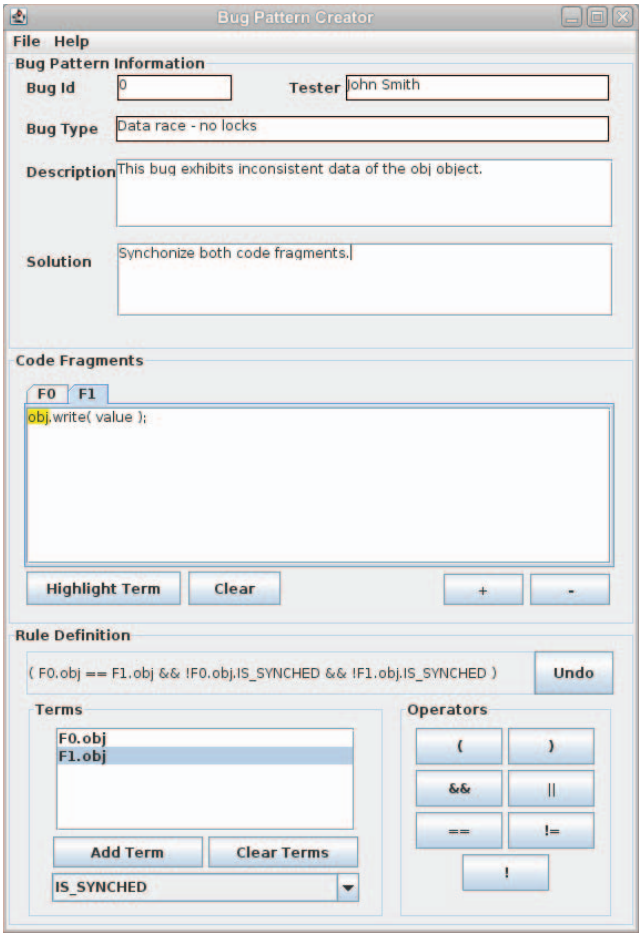

Fig. 6. Bug Pattern Creator

Pattern Creator tool (see Figure 6).

- **Clone Detection (using ConQAT):** ConQAT's clone detection algorithm automatically normalizes the inputs and then detects all the type I, II and III clones. This process requires two inputs: the source code of the software system under test, and the code fragment (i.e., code snippet) files. We are interested in finding clones of the original code fragments within the system under test's source code. Since this process will find type II and III clones, the textual naming of clones can be different as well as clones can contain gaps. These gaps can either be an *INSERT* or a *DELETE* gap. In both cases the line with the gap was removed before the clone was matched.
- **Cloned Fragment Grouping:** Bug pattern information is now used to form all possible bug pattern matches using a combinatorial approach. A match is found if we identify a set of code fragments (detected in the clone detection step) that are clones of all of the code fragments for a given bug pattern. The output of this process is an XML file of all the possible code fragment groups (called matches) for each group of bug pattern fragments. An XML bug pattern match is shown in Figure 7, which corresponds to a match of the deadlock example (see Figures 2 and 4).

```
<match ...>
  <cloneFragment sourcePath="/src/bug.java" startLine="91"
    countLines="6" patternId="1" fragmentId="0">
    ...
    <gaps>
      <gap origin="CLONE" line="1" newLine="1"
        type="DELETE"/>
      ...
    </gaps>
    <terms>
      <term id="F0.lockA" line="0" tokenPosition="2" .../>
      ...
    </terms>
  </cloneFragment>
  <cloneFragment sourcePath="/src/bug1.java" ...
    fragmentId="1">
    ...
  </cloneFragment>
</match>
```

Fig. 7. Deadlock bug pattern match

- **Finding Terms in Cloned Fragments:** We now have clones of each bug pattern fragment and we have grouped the cloned source code fragments into sets that match each bug pattern's fragments. Although we have identified the clones in the source code we have not yet identified the terms specified in the bug pattern fragments. In the case of type I and type II clones, terms on the original fragments of the bug pattern can be accurately matched to the source code clone fragments with relative ease. However, in the case of type III clones where gaps can occur the matching of terms can be non-obvious. In this case we use a modified implementation of the Levenshtein algorithm to score each of the potential terms. The best scored term has the highest chance of being the correctly matched term in respect to the term's location on the original fragment.
- **Rule Evaluation:** At this step in the process all clone fragments within all matches have identified terms. In order to determine if a set of code fragments in our system under test actually interacts as described in a bug pattern we must evaluate the bug pattern's interaction rule. We evaluate the original bug pattern rule using the terms from the code fragments in the matching group. The evaluation of the new rule, uses static analysis to assess any `IS_SYNCHED` properties that are encountered. If the rule evaluates to true then this match has a higher probability of being a real bug, and is classified as a high-potential bug otherwise it is classified as a low-potential bug. The output of this process is an XML file of all high- and low-potential bug matches.
- **Report Generation:** Now that we have detected the potential bug matches we transform the XML report generated by the previous step and produce an HTML report that organizes all the found bugs as high- and low-potential bugs.
- **Test Potential Bugs (*under development*):** All of the previous steps combine to form the pre-processing portion

of active testing. We next plan to use the potential bugs from the pre-processing to localize the testing effort with a tool like Java PathFinder or ConTest. An XML file of all the potential bug matches along with a set of test cases are used as input to this step and the output will be a list of bugs detected.

V. Proposed Evaluation

In order to comprehensively evaluate our active testing research we need to satisfy the following three goals:

1) *Ensure that our specification notation for concurrency bug patterns is expressive enough to handle many different types of concurrency bugs.* Currently, we have used our specification notation to describe different deadlock and data race anti-patterns [10]. In these examples we found the rule notation was capable of capturing the interaction behaviour between threads involved in a deadlock or data race.
2) *Assess our bug detection process and the use of clone detection with finding concurrency bugs.* First, we would like to determine if we can detect known bugs that match a given bug pattern, and second we would like to determine the percentage of high- and low-potential bugs that are spurious results. Currently we have used our tool with several small examples and been able to find known bugs. However, we have not assessed the rate of false positives when using our current approach. In the future, we would like to evaluate our technique on several large systems with real concurrency bugs. In addition, concurrency mutation operators [15] can also be applied to software systems to simulate the introduction of concurrency bugs.
3) *Evaluate the benefits of using the high-potential bugs to localize testing effort.* Currently we are in the planning stages for this evaluation.

VI. Conclusions and Future Work

Our research uses clone detection to find sets of matching code fragments of bug patterns. Interaction rules are then applied to identify high- and low-potential bugs based on the interactions of the code fragments.

Localization of testing effort around potential concurrency bugs is vital to increasing the efficiency and effectiveness of concurrency testing tools. The search space of concurrent software is very large and the ability to reduce this search space using clone detection, even with the possibility of false positives, can be beneficial in improving testing efficiency.

Additional work is required to complete our active testing process and future studies. Experimentation with our tool is required to assess the benefits when compared to existing active testing tools like ConFuzzer.

Acknowledgment

The authors would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for funding this research.

References

[1] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.
[2] J. S. Bradbury, "Using Program Mutation for the Empirical Assessment of Fault Detection Techniques: A Comparison of Concurrency Testing and Model Checking," Ph.D. dissertation, Queen's University, 2007.
[3] S. D. Stoller, "Testing Concurrent Java Programs using Randomized Scheduling," in *Proc. of the 2nd Workshop on Runtime Verification (RV'02)*, 2002.
[4] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation." *IBM Systems Journal*, vol. 41, no. 1, pp. 111–125, 2002.
[5] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "CalFuzzer: An extensible active testing framework for concurrent programs," in *Proc. of the 21st International Conference on Computer Aided Verification (CAV 2009)*, 2009, pp. 675–681.
[6] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.
[7] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, Mar. 2006.
[8] P. Jablonski and D. Hou, "CReN: A Tool for Tracking Copy-and-Paste Code Clones and Renaming Identifiers Consistently in the IDE," in *Proc. of the Eclipse Technology Exchange Workshop (ETX'07)*, 2007, pp. 16–20.
[9] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE 2007)*, 2007, pp. 55–64.
[10] J. S. Bradbury and K. Jalbert, "Defining a Catalog of Programming Anti-Patterns for Concurrent Java," in *Proc. of the 3rd International Workshop on Software Patterns and Quality (SPAQu'09)*, 2009, pp. 6–11.
[11] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs," *Automated Software Engineering Journal*, vol. 10, no. 2, pp. 203–232, 2003.
[12] "Systematic concurrency testing using CHESS," in *Proc. of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD 2008)*, 2008.
[13] F. Deissenboeck, E. Juergens, B. Hummel *et al.*, "Tool support for continuous quality control," *IEEE Software*, vol. 25, no. 5, pp. 60–67, 2008.
[14] E. Juergens, F. Deissenboeck, and B. Hummel, "CloneDetective - a workbench for clone detection research," in *Proc. of the 31st International Conference on Software Engineering (ICSE'09)*, 2009, pp. 603–606.
[15] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *Proc. of the 2nd Workshop on Mutation Analysis (Mutation 2006)*, Nov. 2006, pp. 83–92.