Identification and Annotation of Concurrency Design Patterns Using Static Analysis

Martin Mwebesa, Jeremy S. Bradbury
Software Quality Research Group
Faculty of Science (Computer Science)
University of Ontario Institute of Technology
Oshawa, ON, Canada
{*martin.mwebesa, jeremy.bradbury*}*@uoit.ca*

***Abstract*—The verification and maintenance of concurrent software are challenging activities due to the number and complexity of possible thread interleavings and the difficulty in detecting faults that may occur in only a small number of these interleavings. Concurrency design patterns are available to assist the developer in producing concurrent software that improves performance compared to a sequential implementation while minimizing the occurrence of hard to detect bugs. A lack of documentation combined with the distribution of design pattern elements across multiple source files mean a software maintainer may not be aware of the pattern and may inadvertently break the pattern while still producing correctly functioning code. Broken design patterns make the source code brittle and increase the difficulty in performing future maintenance tasks.**

In this paper we discuss our static analysis technique to detect concurrency design patterns in Java. Once a pattern is detected our approach adds commented Java annotations that aid in future maintenance of both the source code and the concurrency design pattern. We have evaluated our approach by analyzing 17 examples, mostly from open source repositories. The results of detecting 8 concurrency design patterns in the examples are reported as well as a discussion of the appropriateness of static analysis for detecting concurrency design patterns is given.

***Keywords*-design patterns; pattern detection; annotation; concurrency; static analysis; source code analysis; TXL;**

I. Introduction

> *"design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context"* [1].

There are many advantages of design patterns that are relevant to both the design and maintenance of software. With respect to maintenance, design patterns are beneficial because they explicitly defining the interaction of classes and objects [1].

Unfortunately, despite the benefits there are also negative implications of using design patterns on the maintenance of the software. The primary negative aspect of design patterns with respect to maintenance stems from the fact that design patterns are often poorly documented in the source code. Unlike the documentation of code units such as methods or classes which is typically placed before the unit, the documentation of design patterns does not have an obvious place in source code because design patterns typically encompass multiple code units and span across a number of different source files. The lack of clarity on the documentation of design patterns means that many design patterns that appear in source code are only partially documented.

Software maintainers rely heavily on the corresponding documentation (including code comments) when modifying existing code, adding new code or removing old code. If a design pattern is not documented and is obfuscated by its distribution across source files a maintainer may not be aware of the pattern and may inadvertently break the pattern while still producing correctly functioning code. Broken design patterns make the source code brittle and increase the difficulty in performing future maintenance tasks.

> *"It often happens that design patterns can not be identified in source code because the conceptual entity of the pattern at the design level is scattered over different parts of an object or even multiple objects on the implementation level. Intentions of design pattern and system specific intentions are superimposed in implementation and without explicitly distinguishing between sections implementing each of them, it becomes very difficult to identify the constructs constituting the pattern later. Due to the inability to identify individual parts of a pattern, they may be modified inadvertently which may result in breaking the pattern and losing the benefits gained by its application in the system."* [2].

Concurrent software is even more difficult to test then

sequential code - makes the problem of documenting and maintaining design patterns even more urgent for concurrent software.

We propose an approach to automatically detecting concurrency design patterns using static analysis. In Section II we will overview concurrency design patterns and the 8 specific design patterns considered in our research. In Section III we describe our overall approach to the detection and annotation of concurrency design patterns as well as give details of the underlying algorithms implemented in the source transformation language TXL [3]. Our evaluation methodology and results are presented in Section IV and related work is discussed in Section V. Finally, in Section VI we discuss conclusions and future work.

II. Concurrency Design Patterns

Concurrency design patterns differ from other design patterns discussed in the literature more in their purpose then in their implementation or description. In general, concurrency design patterns address 2 problems with concurrent software [4]:

1) *Access to shared resources:* it is necessary to ensure that shared resource access prevents threads from interfering with each other when accessing a resource at the same time. Incorrect management of shared resource access can lead to data races and deadlocks.
2) *Controlling the sequencing of operations:* it is not only important to ensure often exclusive access to shared resources, it is also important to ensure the order in which shared resources are accessed. Improper management of operation sequences can lead to some threads have to wait long times for access and can possibly cause starvation in which a particular thread never obtains access to a shared resource.

In our research we consider the following 8 concurrency design patterns [4]:

- *Single Threaded Execution (or Critical Section):* implements guarded methods and the simplest concurrency design pattern. In Java this pattern is achieved using synchronized methods and blocks.
- *Lock Object:* enables a thread to have exclusive access to multiple objects.
- *Guarded Suspension:* allows a thread to wait until specific pre-conditions have been met before proceeding.
- *Balking:* allows a thread to stop executing completely if a condition has not been met.
- *Scheduler:* allows for the ordering of threads.
- *Read/Write Lock:* enables concurrent reads and exclusive writes to operations in concurrent software.
- *Producer-Consumer:* allows for coordinated sequential producing and consuming of objects concurrent software.
- *Two-Phase Termination:* allows for the orderly shutdown of a thread.

```
class Queue{
   public synchronized pull(){
          while (isEmpty()){ //the precondition
                wait();
          }
          //...
          //...
  }
 public synchronized void push(){
        //...
        notify();
 }
}
```

Figure 1. Illustration of the Guarded Suspension design pattern in a queue class [4]

The above list is not exhaustive, but is representative of concurrency design patterns, making it an ideal set of patterns for our research.

A. Guarded Suspension Design Pattern

To assist in the explanation of our design pattern detection approach in Section III we will focus on the Guarded Suspension design pattern and use it as an example for the remaining sections of the paper. The Guarded Suspension design pattern is used in situations where a pre-condition exists but may not always be satisfied. This pattern allows for the execution of the method to be suspended until all pre-conditions have been met. A good illustration of when the Guarded Suspension design pattern could be useful is in the push and pull methods of a queue (see Figure 1) [4].

The push() method adds objects to the queue while the pull() method removes objects from the queue. Both methods are synchronized using the synchronized construct so that multiple threads can safely make concurrent calls to them. Because both methods are synchronized a problem could occur when the queue is empty and a call is made to the pull() method. The pull() method waits for the push() method to provide it with an object to pull, but because they are synchronized, the push() method cannot occur until the pull() method returns. However, the pull() method will never return until the push() method executes and therefore a deadlock has occurred. A solution to avoiding a deadlock would be to add an IsEmpty() precondition which when true would cause the execution of the pull() method to be suspended as long as the queue is empty. This solution clearly demonstrates the implementation of the Guarded Suspension design pattern.

III. Detection Approach

We have developed and used a generalized approach to detection of concurrent design patterns using static analysis which is applied individually to each concurrency design pattern. In this section we will explain the approach and illustrate the realization of the approach in the development of a tool for detecting and annotating the Guarded Suspension

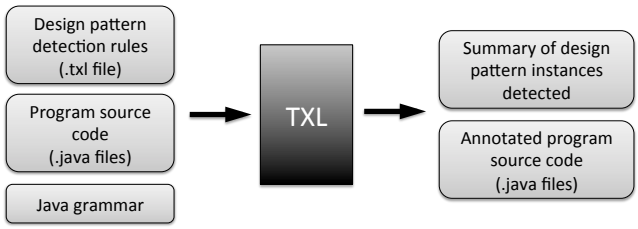

Figure 2. Our Concurrency Design Pattern Detection Approach

```
//*************************************************//
//*** Guarded Suspension pattern:                 ***//
//*** If a condition that prevents a method from  ***//
//*** executing exists, this design pattern       ***//
//*** allows for the suspension of that method    ***//
//*** until that condition no longer exists.      ***//
//*************************************************//
import java.util.ArrayList;

public class Queue {
    private ArrayList data = new ArrayList();

    //***Role = 1(Ensuring a method in the class is
        synchronized.
    //***Contains Role 1a.); ID = 1.
    //***Role = 1a(Ensure there is a notify() or notifyAll
        () statement.);
    //***ID = 1.
    synchronized public void put(Object obj) {
        data.add(obj);
        notify();
    } // put(Object)

    //***Role = 2(Ensuring a method in the class is
        synchronized.
    //***Contains Role 2a.); ID = 1.
    //***Role = 2a(Ensuring there is a while statement.
    //***Contains Role 2aa.); ID = 1.
    //***Role = 2aa(Ensuring there is a wait() statement.)
        ; ID = 1.
    synchronized public Object get() {
        while (data.size() == 0){
            try {
                wait();
            } catch (InterruptedException e) {
            } // try
        } // while
        Object obj = data.get(0);
        data.remove(0);
        return obj;
    } // get()
} // class Queue
```

Figure 3. Guarded Suspension design pattern roles in Java source code

design pattern in Java programs. We follow this approach in creation of all of our detection tools for the concurrency design patterns listed in Section II.

A high-level view of our detection approach is presented in Figure 2. The approach is constructed around the usage of the TXL interpreter [3] which performs the static analysis detection as well as the annotation of the source code with design pattern information. The TXL interpreter takes the following inputs:

- *TXL program(s)*: a TXL program contains the rules for detecting a specific concurrency design pattern and annotating the source code to let the maintainer know where instances of the design pattern were detected. The detection rules are based on a role-based specification of the design pattern discussed in the following sub-section.
- *Java source code*:- the program under analysis that is being checked for the presence of concurrency design patterns.
- *Java grammar*: a required input for TXL is a grammar file which enables the TXL interpreter to parse the source code prior to analysis.

The output from the TXL interpreter is the original Java source code withe with any instances of the design pattern identified and annotated.

A. Specification of Design Pattern Roles

The first step in our approach is to clearly specific the design pattern being detecting. We specify each design pattern as a set of roles that are composed to give the complete design pattern specification. A *role* is similar to a constraint that is specified for the different parts of a design pattern and describes each part's structure and operations. For example, in the Single Threaded Execution design pattern, one role (in fact, the only role) is that a method has to be synchronized.

An example of the specification of design pattern roles in a more complicated design pattern, Guarded Suspension, is shown in Figure 3 in which the roles are included as comments at the method-level. All of the roles in the Guarded Suspension can be composed to specify the entire design pattern. Each method-level role can be hierarchical and can be composed of subroles. For example, in Figure 3 the put() method should be synchronized and should include a notify() or notifyAll() method call. The specificity of the roles used in our detection approach are directly corelated to the recall and precision of the actual detection tool.

B. Creation of the design pattern detection and reporting rules in TXL

Once the roles have been determined we next proceed with creating TXL rules to correspond to the roles. Together the TXL rules form the core part of our concurrency design pattern detection program. This process is repeated for each of the eight concurrency design patterns. If the all of the TXL rules are statisfied then we have a complete design pattern match, if only some of the rules are satisfied we have a partial design pattern match - some of the structural elements of the pattern were detected but not all could be located. If none of the rules have been satisfied then no instances of the concurrency design pattern have been detected.

```
function main
  export Counter [number]
    0
  export CountFirstSynchMethIDs [number]
    0
  export CountSecondSynchMethIDs [number]
    0
  export numVarsIDCollection [repeat id]
    _
  export FirstSynchMethIDs [repeat id]
    _
  export SecondSynchMethIDs [repeat id]
    _
    export notifyCollection [repeat expression]
    _
  export waitCollection [repeat expression]
    _
  replace [program]
       P [program]
  by
    %First run the detection rule to identify all
    %instances of the design pattern
    P [findGuardedSuspensionPattern]
    %If no pattern was found print out the appropriate
    %message
    [printPatternNotFound]
    %If conditions are satisfied print out the details of
    %design pattern instances detected
    [printOutput] [printFirstSynchMethIDs]
    [printSecondSynchMethIDs] [printNotifyCollection]
    [printWaitCollection]
end function
```

Figure 4. Guarded Suspension detection program – the main function

```
rule findGuardedSuspensionPattern
  replace [class_declaration]
      CH [class_header] CB [class_body]
  construct NumVarInstancesFound [class_body]
    CB [findAllNumberVars]
  construct InstanceFoundFirstSynchMeth [class_body]
    CB [find1stSynchMethod1] [find1stSynchMethod2]
  construct InstanceFoundSecondSynchMeth [class_body]
    CB [find2ndSynchMethod1] [find2ndSynchMethod2]
  by
    'IDENTIFIED CH CB
end rule
```

Figure 5. Guarded Suspension detection program – the findGuardedSuspensionPattern rule

Regardless of the detection programs outcome, we print out the detection results prior to terminating the program. For example, in the case of the Guarded Suspension design pattern we printed out the names of the methods that satisfy Role 1 (i.e. the synchronized methods that have a notify() or notifyAll() statement within them) and Role 2 (i.e. the synchronized methods that have a loop within them and a wait() statement within the loop).

The main function in our TXL program to detect instances of the Guarded Suspension design pattern is shown in Figure 4. We start by declaring global variables using the TXL keyword export. These variables will be used later

Table I
ROLES IN THE GUARDED SUSPENSION DESIGN PATTERN

Role ID	Role Description
1.	Ensuring a method in the class is synchronized and contains Role 1a below.
1.a.	Ensuring there is a notify() or notifyAll() statement within the Role 1 method.
2.	Ensuring a method in the class is synchronized and contains Role 2a below.
2.a.	Ensuring there is a loop within the Role 2 method and contains Role 2aa below.
2.a.a.	Ensuring there is a wait() statement within the Role 2a loop.

```
% Function print out the number of Guarded Suspension
% design pattern instances found.
function printOutput
  replace [program]
    P [program]
  import Counter [number]
  where
    Counter [> 0]
  construct InstanceFound [stringlit]
    "** Instances of Guarded Suspsension Pattern found = "
  construct InstanceFoundPrint [id]
    _ [unquote InstanceFound] [+ Counter] [print]
  by
    P
end function
```

Figure 6. Guarded Suspension detection program – the printOutput function

in our TXL program to, amongst other things, obtain the number of Guarded Suspension design pattern instances and the number of synchronized methods. After declaring those global variables we use a replace-by clause to replace the program by itself with a number of rules and function applied. The first rule applied to the program (i.e. findGuardedSuspensionPattern) will detect instances of the design pattern and the remain rules handle the reporting of the program output to the software maintainer.

The findGuardedSuspensionPattern rule (see Figure 5) is responsible for identifying any parts of the program that satisfy the Guarded Suspension design pattern roles (see Table I for a description of each). For example, the find1stSynchMethod1 rule called from the findGuardedSuspensionPattern rule corresponds to role 1 of the Guarded Suspension design pattern and ensures that a method in the Java program under analysis matches one of two required synchronized methods defined in the specification (i.e., roles).

To demonstrate one of the reporting functions we have included printOutput in Figure 6. In this function we first check to see if we have detected any instances of the Guarded Suspension design pattern (i.e., Counter [> 0]). If the condition is satisfied we output the details for each instances of the pattern identified. Since we are use a

```
import java.util.ArrayList;

public class Queue {
    private ArrayList data = new ArrayList ();

    /* ''@GuardedSuspensionPatternAnnotation(patternInstanceID=1, roleID=1, roleD
escription='Ensuring a method in the class is synchronized - guarded')" */
    synchronized public void put (Object obj) {
        data.add (obj);
        /* ''@GuardedSuspensionPatternAnnotation(patternInstanceID=1, roleID=1a,
roleDescription='Ensure there is a notify() or notifyAll() statement.')" */
        notify ();}

    /* ''@GuardedSuspensionPatternAnnotation(patternInstanceID=1, roleID=2, roleD
escription='Ensuring a method in the class is synchronized - guarded')" */
    synchronized public Object get () {

        /* ''@GuardedSuspensionPatternAnnotation(patternInstanceID=1, roleID=2a,
roleDescription='Ensuring there is a while statement.')" */
        while (data.size () == 0) {
        {
            try {
                /* ''@GuardedSuspensionPatternAnnotation(patternInstanceID=1, roleID=2aa,
                roleDescription='Ensuring there is a wait() statement.')" */
                wait ();} catch (InterruptedException e) {
            }
        }} Object obj = data.get (0);
        data.remove (0); return obj;
    }
}
```

Figure 7. The Guarded Suspension design pattern with annotations

structural static analysis approach to detect the design pattern instances, it is possible that some of the results may be spurious.

C. Refinement of the TXL design pattern detection tool

Once we created an initial version of the 8 TXL programs - each design to detect one of the concurrency design patterns we included a refinement stage in the development to optimize the level of granularity of the matching rules so that we are capable of detecting as many different instances of a given design pattern as possible while still minimizing the number of false positives report. To assist in the refinement we evaluated each detection program against a code example provided with the design pattern description in [4]. We also varied the implementation styles of these code examples by hand to see if the tool was robust enough to detect these instances as well. At the end of the refinement stage we were able to detect the design pattern instances in the example code while reporting no false positives.

D. Source code annotations of the design pattern instances

After completing the initial development and necessary refinements for the detection of the concurrency design patterns in the Java source code we modified our TXL programs to enable them to actually transform the Java source code examples by adding commented Java annotations. The custom Java annotations we created, correspond directly to the concurrency design pattern roles (see Figure 7). These commented Java annotations identify specifically where in the Java source code the various concurrency design pattern roles that constitute the respective concurrency design patterns exist.

Our original intention was to use Java annotations and not use annotations in comments. However, the current support for annotations in Java limit their use to the class declaration, field declaration and method declaration levels. Many of the annotations we insert are most appropriate at the statement level which is currently not supported but maybe in the near future [5].

IV. Evaluation

In this section we present our evaluation of our 8 concurrency design pattern identification tools. First, we present our experimental setup and procedure methodology (Section IV-A) before presenting our results (Section IV-B) and threats to validity (Section IV-D). We will also elaborate on how differences in the structural nature (*"Structural patterns deal with the composition of classes or objects"* [1]) and behavioral nature (*"Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility"* [1]) of the different concurrency design patterns we targeted, influenced the outcome of the results.

Table II
LIST OF JAVA SOURCE CODE EXAMPLES USED FOR EVALUATION

Example program	Documented Concurrency Design Patterns Present	Source
Vector	Single Threaded Execution	java.util.Vector library
Data	Balking	http://oliverlee.googlecode.com/svn/
RequestQueue	Balking	http://oliverlee.googlecode.com/svn/
RequestQueue2	Balking	http://oliverlee.googlecode.com/svn/
SaverThread	Balking	http://oliverlee.googlecode.com/svn/
ChangerThread	Balking	http://oliverlee.googlecode.com/svn/
ReadWriteLockEx	Read/Write Lock	student implementation
BlockingQueue	Guarded Suspension	http://www.pushlets.com/src/ (nl.justobjects.pushlet.core package)
EventQueue	Guarded Suspension	http://www.pushlets.com/src/ (nl.justobjects.pushlet.core package)
RequestQueue3	Guarded Suspension	http://oliverlee.googlecode.com/svn/
ServerThread	Guarded Suspension	http://oliverlee.googlecode.com/svn/
CancellableThread	Two-Phase Termination	http://opensitesearch.sourceforge.net/ (ORG.oclc.util packages)
Component	Lock Object	java.awt library
Queue	Producer-Consumer	http://commons.apache.org/dormant/threadpool/ (org.apache.commons.threadpool package)
MTQueue	Producer-Consumer	http://www.clickblocks.org (org.exoplatform.services.threadpool.impl package)
ProducerConsumerEx	Producer-Consumer	student implementation
SchedulerEx	Scheduler	student implementation

A. Experimental Setup and Procedure

We will now discuss the experimental setup and procedure used to evaluate our approach to the detection and annotation of concurrency design patterns in Java source code. The purpose of our experiment is to establish how effective our static analysis detection technique is at identifying instances of the 8 different concurrency design patterns. Specifically, we are interested in the recall of our approach as well as the precision.

To evaluate the 8 detection tools we decided to primarily use open source programs which contained known concurrency design patterns. We selected 14 open source examples and supplemented them with 3 student written examples (see Table II). The student written examples were programmed by graduate students who were given a description of a concurrency design pattern and asked to create an example program that made use of the pattern. All three programs were written by different students who were not involved in the development of the detection tools or in conducting the empirical evaluation. All 17 of the programs are independent and were selected to avoid bias in our evaluation.

The experimental environment chosen for this evaluation was laptop computer with an Intel Core(TM)2 Duo CPU P8600 with 2.4GHz processors and 3GB of RAM.

The experimental procedure involved all 8 concurrency design pattern detection tools being run against all 17 examples. In the case of the 14 open source examples, only the files containing the design pattern were given as input to the tools while in the case of the 3 student written examples, the entire program was provided. The number of lines of code analyzed by the design pattern detection tools varied from 15 lines of code to 9453 lines of code. For each run of each tool we collected the output report by the tool as well as performance information (i.e., cpu execution time).

B. Results

We will now present the results of evaluating the 17 example programs on the 8 concurrency design pattern tools. The design patterns are discussed in order from simple to complex and for each design pattern detection tool we provide:

- The number of known design pattern instances present in the program.
- The number of complete design pattern instances detected. A complete match occurs when all of the design pattern roles are identified in the example program.
- The number of partial design pattern instances detected. A partial match occurs when only some of the design pattern roles are detected.
- The number of false positives.

To save space we have excluded programs from some of our tables if the programs contained no instances of a given design pattern and no false positives were detected. Even though this data has been excluded it is important to reiterate that all example programs were evaluated by each detection tool.

Single Threaded Execution Design Pattern. The Single Threaded Execution design pattern is the most common concurrency design pattern, occurring in almost all of the other concurrency design patterns. Of all the concurrency design patterns, the Single Threaded Execution design pattern is also the most basic in structure and has only one role. Therefore, it has one of the highest detection rates using our detection tools (see Table III). Interestingly, it

Table III
EVALUATION RESULTS FOR SINGLE THREADED EXECUTION DETECTION TOOL

Program	Instances Present	Complete Instances Detected	Partial Instances Detected	False Positives
Vector	37	36	0	0
Data	2	2	0	0
RequestQueue	2	2	0	0
RequestQueue2	2	2	0	0
ReadWriteLockEx	2	2	0	0
BlockingQueue	7	7	0	0
EventQueue	8	8	0	0
RequestQueue3	2	2	0	0
CancellableThread	3	2	0	0
Component	44	32	0	0
Queue	5	5	0	0
MTQueue	4	4	0	0
ProducerConsumerEx	2	2	0	0
SchedulerEx	2	2	0	0
Total	**122**	**108**	**0**	**0**

Table IV
EVALUATION RESULTS FOR BALKING DETECTION TOOL

Program	Instances Present	Complete Instances Detected	Partial Instances Detected	False Positives
Data	1	1	0	0
Total	**1**	**1**	**0**	**0**

turned out the instances of this pattern that weren't detected were the result of a bug in our TXL program which did not consider synchronization blocks at some parts of the program a match. A fix of this bug results in 100% detection of the 122 instances present in our example programs.

Balking Design Pattern. This design pattern works at the method level and in order for it to be detected 3 roles must be satisfied, namely:

1) Ensuring the method is synchronized - guarded.
2) Ensure an if statement that tests a flag right at the start of the synchronized method.
3) Ensuring an if statement or an else statement that tests the flag in 2 above does an immediate return - balking.

Table IV shows the results from running our Balking design pattern detection program against the 17 Java source code examples. Only the Data program had an instance of the Balking design pattern and it was successfully detected. The document for the SaverThread and ChangerThread classes indicated that both classes had an instance of the Balking. However, a manual inspection of the code confirmed our detection tool's result that there were no instances of the Balking design pattern present.

Guarded Suspension Design Pattern. The Guarded Suspension design pattern usually consists of two synchronized methods within a class. The first synchronized method will contain a notify() or notifyAll() and the second synchronized method will contain a loop and somewhere within the loop a wait() statement.

Table V
EVALUATION RESULTS FOR GUARDED SUSPENSION DETECTION TOOL

Program	Instances Present	Complete Instances Detected	Partial Instances Detected	False Positives
RequestQueue	1	1	0	0
RequestQueue2	1	1	0	0
ReadWriteLockEx	1	0	1	0
BlockingQueue	2	0	1	0
EventQueue	3	0	1	0
RequestQueue3	1	1	0	0
Queue	1	1	0	0
Total	**10**	**4**	**3**	**0**

Table VI
EVALUATION RESULTS FOR LOCK OBJECT DETECTION TOOL

Program	Instances Present	Complete Instances Detected	Partial Instances Detected	False Positives
Component	36	36	0	0
Total	**36**	**36**	**0**	**0**

Table V shows the results from evaluating our Guarded Suspension design pattern detection program on the 17 Java source code examples. Out of the 10 instances present, we were able to detect 4 completely and 3 partially. An interesting finding with respect to two of the completely detected design pattern instances occurred in the case of the RequestQueue and RequestQueue2 classes. In both cases an instance of the Balking design pattern was documented in the source code however the Balking detection tool did not detect any instances while the Guarded Suspension tool detected an instance in both classes. Manual inspection of the source code determined that the documentation was in fact incorrect.

For the instances of the Guarded Suspension design pattern that were present but were only partially detected or were not found at all, we manually inspected the code and found some interesting results. The problem we expected did in fact occur – our tool did not generalize the design pattern roles enough to detect the instance (e.g., in ReadWriteLockEx). In two other cases (BlockQueue and Event Queue), the 2 main roles that comprise an instance of the Guarded Suspension design pattern were found within the same method and our tool was not capable of handling this possiblity.

Evaluation results for Lock Object Design Pattern. The Lock Object design pattern has 3 major roles:

1) Role 1: Lock object - a static object in the class.
2) Role 2: Lock object method - a static method in the same class as Role 1 which returns an instance of the Lock object, Role 1.

Table VII
EVALUATION RESULTS FOR PRODUCER-CONSUMER DETECTION TOOL

Program	Instances Present	Complete Instances Detected	Partial Instances Detected	False Positives
Queue	1	0	1	0
MTQueue	1	0	1	0
ProducerConsumerEx	1	0	0	0
Total	**3**	**0**	**2**	**0**

Table VIII
EVALUATION RESULTS FOR READ/WRITE LOCK DETECTION TOOL

Program	Instances Present	Complete Instances Detected	Partial Instances Detected	False Positives
ReadWriteLockEx	1	0	0	0
Total	**1**	**0**	**0**	**0**

Table IX
EVALUATION RESULTS FOR SCHEDULER DETECTION TOOL

Program	Instances Present	Complete Instances Detected	Partial Instances Detected	False Positives
SchedulerEx	1	0	1	0
Total	**1**	**0**	**1**	**0**

Table X
EVALUATION RESULTS FOR TWO-PHASE TERMINATION DETECTION TOOL

Program	Instances Present	Complete Instances Detected	Partial Instances Detected	False Positives
CancellableThread	1	0	0	0
Total	**1**	**0**	**0**	**0**

3) Role 3: Synchronized calls to method Role 2. We consider each synchronized call to the Role 2 method, an instance of the Lock Object design pattern.

The Lock Object design pattern was present in just one of our example programs and was identified correctly (see Table VI). This design pattern is more structural in nature than most of the other concurrency design patterns – many of which contain both structural and behavioural elements. Being more structural in nature makes it a more suitable candidate for identification using a static analysis techniques like ours. The results are shown in .

Producer-Consumer Design Pattern. This design pattern is comprised of 3 main roles which encompass 3 classes. These 3 roles are a Producer class, a Consumer class and a Queue class, all of which interact with one another. The Producer-Consumer design pattern is very behavioural and this explains the low rate of identifying instances of it within our example programs (see Table VII). Behavioral design patterns tend to vary more widely in their implementation, thus making them more difficult to detect using static analysis without any dynamic analysis to identify the behaviour.

Read/Write Lock Design Pattern. Similar to the Producer-Consumer design pattern the Read/Write Lock design pattern is very behavioral. As a result our detection tool had difficulty and was unable to completely (or even partially) identify the instance in ReadWriteLockEx (see Table VIII).

Scheduler Design Pattern. This pattern usually consists of: a scheduler class, a schedule ordering interface, a request class (implements the schedule ordering interface) and a processor class (delegates scheduling of the request object's processing to the scheduler class, one at a time).

The Scheduler design pattern is another one of the concurrency design patterns that is behavioral. Similar to the Read/Write Lock design pattern and the Producer-Consumer design pattern before it, this certainly had an impact on the results. As explained earlier, the more behavioral a pattern is, the more difficult it is to detect pattern instances using only static analysis.

Table IX shows our results regarding the Scheduler design pattern detection tool. Only SchedulerEx had a complete instance of the Scheduler design pattern and our program identified only the schedule ordering interface.

Two-Phase Termination Design Pattern. Of the 8 concurrency design patterns discussed in this paper, the Two-Phase Termination design pattern, is probably the most varied in the ways that it can be implemented. Like with the the Producer-Consumer design pattern, the Read/Write Lock design pattern and the Scheduler design pattern, the Two-Phase Termination design pattern is also more behavioral than structural. As our results in Table X show, no complete match or even partial match of the Two-Phase Termination design pattern instance was identified in the CancellableThread example.

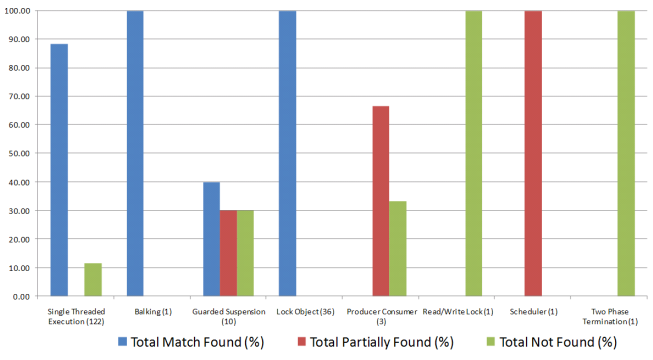

Figure 8. Summary of the effectiveness of the 8 concurrency design pattern detection tools

C. Discussion

Overall, the effectiveness of our static analysis approach to detecting concurrency design patterns is mixed (see Figure 8). For the concurrency design patterns that are more structural, we achieved very high detection rates while the design patterns that contained a large amount of behavioural properties were detecting partially or not at all. Even, though none of the more behavioral patterns were completely detected, the partial detection of the Producer-Consumer and Scheduler design patterns is promising because it at least provides a software maintainer with an indication that a concurrency design pattern may be present.

An important result that should not be overlooked is that there were zero false positives. This result is indicative of the fact that we pursued a more rigid approach in creating the TXL rules in our detection tools. More flexible rules may have lead to more design patterns detected but would probably have seen a number of false positives. This trade-off is an important avenue of future work as we would like to further tune the TXL rules to increase the effectiveness of detecting concurrency design patterns.

We also considered performance in our evaluation and measured the CPU time required by each detection tool with each example program. The time required to evaluate an example ranged from 1 millisecond to 1.52 seconds. There was very little variation between the execution times for different detection tool when applied to the same example program. Also, as expected, the programs with the largest size tended to take longer to analyze. For example, the largest program, Component, was 9453 lines of code and always took the longest time to analyze (1.14 seconds to 1.52 seconds).

D. Threats to Validity

We have taken several steps to mitigate threats to validity in our evaluation of the 8 concurrency design pattern detection tools presented in this paper:

- the majority of examples (14 or 17) used in the evaluation were open-source programs written by third-party developers.
- the three examples developed by students were programmed independently and were not influenced by the authors.
- the programs used to refine and test the detection tools prior to evaluation were not the same example programs used during the evaluation and were also developed independently.

Despite our best effort threats to validity still exist. The biggest threat is the lack of examples for some of the concurrency design patterns. The only way to mitigate this problem is to continue to identify example programs and conduct further evaluations.

V. Related Work

We are not aware of any other approaches that specifically focus on the detection or annotation of concurrency design patterns.

One area of relevant related work is the prior work in the general area of design pattern detection – in particular object-oriented design pattern detection. Object-oriented design patterns can be sub-categorized as creational, structural or behavioural and the underlying approach to detection can vary depending on the sub-class of design patterns targeted. A number of approaches have been proposed to automatically detect object-oriented design patterns including the use of static analysis [6], the combined use of static and dynamic analysis [7] and even the use of similarity scores between graph vertices [8].

Another area of related work focuses on the annotation of object-oriented design patterns. The annotation approaches included below all differ from our work in that they all require manual annotation of the source code by the developer or maintainer. An interesting example of related work in this area is by Sabo and Poruban who aim to assist in resolving broken design patterns during software maintenance by using annotation [2]. Their approach is to manual annotate Java source code with structural constraints for a given object-oriented design pattern. During software maintenance, each source code change will be followed by a conformance check to detect if the design pattern has been broken by the code modification. We plan to extend our research to include a conformance check for already detected design patterns. Other examples of related work are by Rasool, Philippow and Mader [9], Meffert [10] and He, Li and He [11]. Rasool, Philippow and Mader have an approach to the recovery (extraction) of design patterns from legacy systems to aid in maintenance [9]. This work relies on manual annotation of Java source code for the design pattern recovery to occur. The approach by Meffert uses custom Java annotations to detect design patterns [10]. Meffert's work is unique because it expresses both the source code intentions (using custom Java annotations) and the design pattern intentions (external to the source code). He, Li and He's work also involves Java annotations but is interested in extracting and visualizing the design pattern instantiation information [11].

VI. Conclusion

In this paper we have presented a solution to the challenge of preserving the integrity of concurrency design patterns during software maintenance activities. Design patterns are often not well documented and can be difficult to identify since they are generally spread across multiple source files. These factors can lead to design patterns being broken during source code modifications, making the source code brittle and harder to maintain in the future.

Our approach is to use static analysis to detect concurrency design patterns based on the structure and roles of the individual design patterns. In our research, we consider 8 concurrency design patterns: Single Threaded Execution (or Critical Section), Lock Object, Guarded Suspension, Balking, Scheduler, Read/Write Lock, Producer-Consumer and Two-Phase Termination. For each design pattern we have implemented a tool in TXL that can detect instances of the design pattern and annotate the source code with details about the design pattern for use by software maintainers.

We have evaluated our detection tools on 17 example programs, 14 of which were from open source software. The results of our study were promising with respect to the more structural design patterns but our tools had difficulty completely detecting more behavioural design patterns using only static analysis and pattern matching in TXL.

One topic of future work is to combine the use of static and dynamic analysis to detect those concurrency design patterns that have both structural and behavioural elements (e.g, Producer-Consumer, Read/Write Lock, Scheduler, Two-Phase Termination). Further evaluation with more example programs is also needed with respect to these design patterns.

A second topic of future work is automatic repair of concurrency design patterns during software maintenance. We view the detection and annotation of concurrency design patterns as a first step in our research. Next we plan to extend our technique to not only identify the concurrency design patterns but to also detect and report if already identified design patterns have been broken. Once we identify a broken design pattern we would be interested in using genetic programming techniques to fix the design pattern [12].

ACKNOWLEDGMENT

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

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] M. Sabo and J. Poruban, "Preserving design patterns using source code annotations," in *Journal of Computer Science*, 2009, pp. 519–526.

[3] J. R. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, vol. 61, no. 3, pp. 190–210, 2006.

[4] M. Grand, *Patterns in Java, Volume 1*. Wiley Publishing, Inc., 2002.

[5] M. D. Ernst, "Type annotations specification (JSR 308)," Web page: http://types.cs.washington.edu/jsr308/java-annotation-design.html (last accessed: November 5, 2011), 2010.

[6] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06, Washington, DC, USA, 2006, pp. 123–134.

[7] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic design pattern detection," in *Proc. of the 11th IEEE International Workshop on Program Comprehension*, ser. IWPC '03, Washington, DC, USA, 2003, pp. 94–.

[8] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Transactions on Software Engineering*, vol. 32, pp. 896–909, 2006.

[9] G. Rasool, I. Philippow, and P. Mader, "Design pattern recovery based on annotations," in *Advances in Engineering Software*, 2008, p. 123.

[10] K. Meffert, "Supporting design patterns with annotations," in *Proc. of 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, 2006, pp. 437–445.

[11] C. He, Z. Li, and K. He, "Identification and extraction of design pattern information in Java program," in *Proc. of 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD '08)*, 2006, p. 828834.

[12] S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues, "A genetic programming approach to automated software repair," in *Proc. of the Genetic And Evolutionary Computation Conference (GECCO 2009)*, 2009, pp. 947–954.