Alpen-Adria-Universität Klagenfurt
Software Engineering Research Group
Technical Report Series

Extracting Parallel Control Flow Graphs with Synchronization Information from Java Programs

Giovanni Liva, Francesco Spegni, Luca Spalazzi, Andreas Bollin, Martin Pinzger

Report AAU-SERG-2017-001

AAU-SERG-2017-001

Published, produced and distributed by:

Software Engineering Research Group
Institute of Informatics Systems
Faculty of Technical Sciences
Alpen-Adria-Universität Klagenfurt
Universittsstrae 65-67
9020 Klagenfurt
Austria

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://serg.aau.at/bin/view/Main/Publications

For more information about the Software Engineering Research Group:
http://serg.aau.at

Extracting Parallel Control Flow Graphs with Synchronization Information from Java Programs

Giovanni Liva*, Francesco Spegni†, Luca Spalazzi†, Andreas Bollin‡ and Martin Pinzger*
*Software Engineering Research Group, Alpen-Adria Universität Klagenfurt, Austria
Email: {giovanni.liva,martin.pinzger}@aau.at
†Dipartimento di Ingegneria dell'Informazione, Università Politecnica delle Marche, Italy
Email: {f.spegni,l.spalazzi}@dii.univpm.it
‡Informatics Didactics, Alpen-Adria Universität Klagenfurt, Austria
Email: andreas.bollin@aau.at

***Abstract*—Developers spend a significant amount of their time on understanding programs, especially if these programs are large and use multi-threading. Regarding such programs, one key aspect concerns the understanding of how shared data is accessed by the different threads, for instance to prevent race conditions. Most of the existing approaches address this issue by using dynamic analysis of execution traces. However, dynamic analysis has two main shortcomings: first, the program needs to be executable, and, second, execution traces typically do not cover all possible scenarios. In this paper, we propose an approach that uses static analysis to extract Parallel Control Flow Graphs with Synchronization Information (PCFGs-SI) from source code. While CFGs help to understand the control flow of source code statements executed in parallel, synchronization edges highlight the shared access to critical code blocks protected by a locking mechanism. We evaluated our approach and heuristics with five open source Java projects that use two specific locking mechanisms to handle the communication between different threads, namely synchronized code blocks and synchronized methods. The results show that our approach extracts synchronization edges between synchronized code blocks with a precision of 97.50% and recall of 99.50%. Edges between calls to synchronized methods are extracted with a precision of 100% and recall of 96.35%. We demonstrate the usefulness of our approach with two examples, in which our approach is used to detect shortcomings in the use of locking in Java.**

I. Introduction

Developers spend a significant amount of their time on understanding programs, especially if these programs are large and use multi-threading. As presented by Dano et al. [1], reverse engineering a multi-threaded program is, using their analogy, *a trip down to the Hades*. Other studies [1], [2] show that comprehension activities alone consume about $40\%-60\%$ of all the resources available. This is mainly due to two reasons: (i) high staff-turnover is an issue in software maintenance, and (ii) the frequent evolution of programs makes it complicated to keep documentation up to date. Therefore, an automated technique to reverse engineering models from source code that helps to understand and document multi-threaded programs is beneficial.

The extraction of software views from multi-threaded or distributed programs to use as documentation is a challenging task. Joyce *et al.* [3] show that the monitoring of a distributed system is difficult because it requires the dynamic extraction of information about the interaction between different processes. Garcia *et al.* [4] show that similar problems can be found in the debugging of multi-threaded programs. The main issue concerns understanding the distribution of knowledge among multiple parts that intercommunicate with synchronization mechanisms, such as locks. It is hard to produce an exhaustive coverage of all possible scenarios to understand how shared data is accessed by the different processes and threads of a system. However, such an understanding is needed to avoid or fix race conditions.

Since this is a known problem, several approaches have been developed to analyze multi-threaded programs, such as [5], [6], [7] which make use of the runtime instrumentation of the programming language to collect execution traces. Afterwards, the traces are processed to discover undesired behavior in the program under analysis. These approaches use dynamic analysis. Dynamic analysis, however, has two main shortcomings: first, the program needs to be executable, and second, execution traces typically do not cover all possible scenarios. Depending on the phase of the project, not all versions of a software system can be compiled into an executable program. If a system can be executed, it is impossible to execute all possible scenarios to get a complete view of the behavior of that system. Static analysis is used to address these two shortcomings, but only few static analysis approaches exist to analyze the synchronization in software systems. For instance, the Chord framework [8] is capable of detecting statements that can cause a deadlock, atomicity violation, or race conditions, and for each such situation outputs a warning. Static approaches have in common that they detect specific problems in the source code. However, we are not aware of any static analysis approach that provides a comprehensive view on how the different pieces of code in a system synchronize the access to critical code blocks.

In this paper, we propose a novel static analysis approach to extract Parallel Control Flow Graphs with Synchronization Information (PCFGs-SI) from Java source code. Our PCFGs-SI consist of (i) Control Flow Graphs (CFG) to represent the control flow of statements, and (ii) synchronization edges to highlight points of synchronization between code blocks and methods in the CFG. The extraction is done in two

phases: first, the source code is parsed to gather information on classes, attributes, methods, and synchronization statements. Second, the control flow graphs and synchronization information from a selected set of classes is extracted. Regarding the synchronization information, we currently handle two locking mechanisms, namely locking of code blocks and locking of methods. For both, our approach uses a number of heuristics to detect code blocks protected by the same lock or calls to synchronized methods of the same class or object. For each match it inserts a corresponding edge in the control flow graph denoting that the execution of this piece of code needs to be synchronized. Finally, the PCGFs-SI are visualized using a leveled graph visualization technique.

For the evaluation, we implemented our approach and heuristics in a prototype tool that currently supports the Java programming language. Note that our approach can be extended to other programming languages, such as C#, since they provide similar locking mechanisms. Furthermore, our approach currently supports intrinsic locking via the `synchronized` keyword, since these locks occurred most frequently in our subject systems. We applied our tool to five selected open source Java projects that are multi-threaded and use both locking mechanisms. We manually analyzed a random sample of extracted PCFGs-SI to validate the amount of synchronization edges that were correctly extracted and the amount of edges that were missed by our heuristics. The results show that our approach is capable of extracting synchronization edges between code blocks with a precision of 97.50% and recall of 99.50%. Synchronization edges between calls to synchronized methods are extracted with a precision of 100% and recall of 96.35%. In addition, we demonstrate the usefulness of our approach with two examples, in which PCFGs-SI are used to understand and detect shortcomings in the use of locking in Java.

In summary, the main contributions of this paper are: (i) an approach including a first set of heuristics to extract and visualize parallel control flow graphs with synchronization information; (ii) the prototype implementation of the approach; (iii) the evaluation with five open source Java projects; and (iv) the reference data set used for the evaluation.[1]

The remainder of the paper is organized as follows: Section II presents related work. The description of the approach is given in Section III and Section IV presents the experimental study with five open source Java projects. Two examples of PCFGs-SI are presented in Section V. The results and threats to validity are discussed in Section VI. Section VII concludes the paper and outlines directions for future work.

II. Related Work

Related work concerns approaches that use dynamic and static analysis to analyze and better understand multi-threaded programs. Dynamic analysis is the main approach used to reverse engineer information about synchronization behavior of software systems. For instance, Bhattacharya *et al.* [5] use dynamic analysis to align the schedule of threads allowing the disclosure of inference bug patterns. Howarth *et al.* [6] use the Java virtual machine tool interface to monitor information leakage in single-threaded Java programs. Maheswara *et al.* [7] present a visual debugger for Java threaded applications that is built on top of the Java PathFinder [9] framework.

As mentioned above, dynamic analysis approaches face the challenge of handling a large number and size of traces created by executing the programs multiple times. Several approaches exist that address this issue by first performing a static analysis of the source code. For instance, Yuan and Xie [10] first analyze the branching structure and then run the test suite to collect execution traces. Using the branching structure, they are able to merge traces that cover the same branch. Systä *et al.* [11] use a hybrid approach. First, they ask developers to mark parts of the code that they are interested in. Then, they run the software collecting traces of only those marked parts. Several metrics can be used to reduce the size of the traces. For instance, Yuan *et al.* [12] use information about the usage frequency and data dependency between operation nodes (*e.g.*, method calls) as a guide to their partitioning algorithm that reduces the size of the traces.

Several approaches exist that use static analysis for analyzing multi-threaded programs. Many of them extract a state-based representation of the source code to help developers understand and document it. For instance, Amighi *et al.* [13] present a sound technique to extract a Control Flow Graph (CFG) from Java byte code without losing information. Krinke [14] introduces Threaded-CFG, a technique to represent the start and end points of threads. This representation allows the application of slicing techniques to parallel code. Kester *et al.* [15] provide a good overview of static analysis tools that can be used to detect concurrency bugs in multi-threaded programs. As one of their findings they mention that existing tools only manage to find few errors. Among the evaluated systems, the Chord framework [8] showed the best performance concerning false positives. Chord is presented by Naik *et al.* and is capable of detecting statements that can cause a deadlock, atomicity violation, or race conditions, and for each such situation outputs a warning. More bugs can be detected with the approach of Chen *et al.* [16]. They extract Probably Race Condition Graphs from the source code. Those graphs point out shared resources to which access is not properly synchronized.

Tao and Qian [17] present a set of best practices for writing synchronized code in Java programs. Based on these findings, they also present a technique to refactor classes that violate these practices. Regarding synchronization in Java programs, Prado *et al.* [18] introduce the Parallel Control Flow Graph (PCFG). The graph represents control, data, and synchronization flow of a program. In particular, the synchronization flow of a concurrent program is represented by edges among threads and processes that represent operations of synchronization and communication. Their approach to extract synchronization information is, however, completely based on dynamic analysis of programs. This approach has been extended by Howarth *et*

[1] We will make the prototype tool and dataset available on our web-site.

al. [6], who present an approach to use PCFGs for detecting race conditions using model checking.

In this paper, we present an approach that adapts the ideas of Prado *et al.*'s approach [18] using solely static analysis of source code. We also adapt their definition of synchronization flow to match the features of the Java programming language for implementing synchronization in multi-threaded programs. Moreover, our approach differs from the Chord framework [8] in two aspects: (i) it does not use the Java byte code representation, but directly uses the Java source code; and (ii) it does not only detect race conditions, but instead presents a comprehensive view supporting developers in understanding the synchronization in Java programs and exposing possible shortcomings of it.

III. Approach

In this section, we introduce the Parallel Control Flow Graph with Synchronization Information (PCFG-SI) and present our approach to extract it from Java source code. Our approach consists of two phases. First, we parse the source code to gather the information on classes, attributes, methods, and synchronized statements. In the second phase, the methods of a given set of classes are analyzed to extract the synchronization information on the access of critical code blocks. In contrast to existing approaches that use dynamic analysis, we apply static analysis using a set of heuristics. This way we overcome two main shortcomings of dynamic analysis, namely, first, the program needs to be executable, and second, execution traces typically do not cover all possible scenarios.

A. Parallel Control Flow Graph with Synchronization Information

In the context of this work, a Parallel Control Flow Graph with Synchronization Information (PCFG-SI) consists of a set of sub-graphs each representing the Control Flow Graph (CFG) of the code of a given method. We base our definition of control flow graphs on the one provided by Allen [19] and extend it in several ways. In his definition, a node represents a code instruction and an edge the flow of execution from one instruction to the next one. Our first extension is to add a new type of node that represents a block of instructions that are executed atomically. We refer to this node as *synchronized node*. Furthermore, we add two new edge types to represent the synchronization between different control flow graphs. In the work of Prado et al. [18], synchronization edges represent the access to the same code location by different processes. Since we use static analysis in our work, a synchronization edge shows where two code blocks *potentially* need to schedule the access to a code block or method.

Listing 1 presents an example of a potential race condition between two Java classes. Class `TableRowSWTBase` contains the variable `lock` that can be initialized via its constructor. It is declared as `Object`, therefore can be assigned any type of object. `lock` is used in the method `getTableCellCore` to synchronize the access to parts of its code. Class `BuddyPluginBuddy` defines the method `checkTimeouts` that also contains a block of code that is synchronized using `this`. The current implementation allows developers to initialize the variable `lock` with an object of the class `BuddyPluginBuddy` potentially causing an unwanted race condition between the methods `getTableCellCore` and `checkTimeouts`.

Currently, our approach supports two types of intrinsic locks in Java, namely synchronized code blocks and synchronized methods. They are supported by the Java programming language using the `synchronized` keyword. A synchronized code block is a piece of code enclosed by the `synchronized` keyword. Similarly, a method declared as `synchronized` defines an intrinsic lock to protect the atomic execution of all the instructions in that method. In the PCFG-SI, we introduce a *synchronized block edge* to represent the first type and a *synchronized method edge* to represent the second type of synchronization.

Since version 1.5, Java provides the Concurrency API to support also explicit locking. We focused on intrinsic locking since they occurred more frequently in our subject systems and were also shared across classes while explicit locks were not. Furthermore, we targeted the Java programming language because of the availability of a big variety of case studies as open source. Note that our approach can be extended to take into account explicit locks in Java, and to support other programming languages that provide similar locking mechanisms. For instance, C# uses the annotation `MethodImpl(MethodImplOptions.Synchronized)` to define a method as synchronized and the keyword `lock` to define a code block protected by a locking mechanism.

B. Preprocessing

In the preprocessing phase, we use Eclipse JDT[2] to parse the source code of each Java class and extract the following information:

- package name
- class name
- extended class name
- list of methods with their signature
- list of public attributes
- list of imported classes

Package and class names are used to build the fully qualified name of a class. The extended class name is used to reconstruct the inheritance hierarchy of a class. Lists of methods and public attributes are used to correctly link method calls and attribute accesses to the class that declares them. In addition, we extract the list of imported classes to facilitate the reconstruction of the fully qualified names of the data types used in a class.

C. Extraction of PCFGs-SI

To construct the Parallel Control Flow Graphs with Synchronization Information, the developer needs to select at

[2]http://www.eclipse.org/jdt/

```
1  public abstract class TableRowSWTBase implements
       TableRowSWT {
2      ...
3      protected Object lock;
4      ...
5      public TableRowSWTBase(Object lock, ...) {
6          this.lock = lock;
7          ...
8      }
9      ...
10     public TableCellCore getTableCellCore(String
           name) {
11         synchronized (lock) { ... }
12     }
13     ...
14 }
```

```
1  public class BuddyPluginBuddy {
2      ...
3      protected void checkTimeouts() {
4          ...
5          synchronized(this) {
6              ...
7          }
8          ...
9      }
10     ...
11 }
```

Listing 1: Code example taken from Vuze version 5721-04 showing a potential race condition between the method `getTableCellCore` of class `TableRowSWTBase` and the method `checkTimeouts` of class `BuddyPluginBuddy`.

least two methods. Given this input, the analysis process is performed in three steps. In a fourth step, the PCFGs-SI are visualized using leveled graphs.

I. Control Flow Graph: The first step consists of creating the Control Flow Graph (CFG) of each selected method from the abstract syntax tree representation of the source code created by Eclipse JDT. Each instruction in the source code is represented by a node and the control flow between the instructions is represented by edges. When we encounter an instruction that creates an anonymous class, we add a node for that instruction and a node to represent the anonymous class. Inside the latter node, we create the control flow graphs of each method contained by the anonymous class. When we encounter an instruction in the abstract syntax tree that defines a synchronized code block, we add a synchronized node to the CFG. The synchronized node embeds the instructions of its synchronized code block.

In addition, during the visit of the abstract syntax tree, we build a symbol table that contains all the variables of a class and their scope of visibility, *i.e.*, class, method, statement block. Variables that are inherited from parent classes are also included in the symbol table. The symbol table together with the data collected in the preprocessing phase are used to reconstruct the exact type of each variable referenced in the selected methods. This information is needed to extract the synchronization edges between code blocks and methods as described in the following.

II. Synchronized Method Calls: The goal of this step is to extract the *synchronized method edges* between method invocations. For this, we traverse the control flow graphs of the selected methods, and for each node denoting a method invocation, we determine the corresponding method declaration(s). Since object-oriented programming languages support overriding and extension of methods, multiple method declarations might match. Therefore, we introduce the term *compatible* method declaration. We define a method declaration as *compatible* with a method call if it fulfills the following three requirements:

- the names of the method declaration and the call are equal;
- the number of parameters in the method declaration and the number of arguments in the method call is the same;
- and each type in the signatures of the method call and method declaration has a *compatible type*. The definition of *compatible type* for Java is given in [20].

For determining the class whose method is called, we first check which type is referenced in the method invocation. If the invocation does not reference a variable and respectively a class, we first search a compatible method declaration in the current class. If we cannot find a compatible method declaration, we extend the search to its parent classes. If the invocation references a variable, we first determine the class referenced by the variable. If the class is an instantiable (*i.e.*, concrete) type, we search for a compatible method declaration inside that class. If we cannot find a corresponding method declaration, we extend the search to its parent classes until we find a compatible method declaration or reach the root class.

Following the idea of Hammer *et al.* [21], in the case where the method invocation is referencing a virtual method, we resolve all possible concrete implementations of it. If the class referred by the variable is defined as abstract, we determine all its parent- and sub-classes and collect all the compatible method declarations matching the invoked method. If the type is an interface class, we first determine all the sub-interfaces extending the interface, next all the classes implementing these interfaces, and then all the sub-classes of these classes. We search these classes for compatible method declarations that match the method invocation. From the list of matched compatible method declarations we keep only the methods declared as `synchronized` and record the fully qualified names of classes for those.

In the next step, the list of classes of each pair of method invocations from two different control flow graphs (including the control flow graphs created for methods of anonymous classes) are compared. If the list of classes of two method invocations have at least one class in common, we insert a syn-

chronized method edge between the two corresponding nodes. We distinguish two types of synchronized method edges: *maybe*-synchronized method edge and *mustbe*-synchronized method edge. We create a mustbe-synchronized method edge if both method invocations point to static synchronized methods of the same class. Only then can we guarantee that they share the same lock. In the other cases, we create a maybe-synchronized method edge.

III. Synchronized Code Blocks: In this step, we extract the *synchronized block edges* between synchronized code blocks. A synchronized block edge models the fact that two synchronized code blocks *potentially* share the *same* lock.

Our approach traverses the control flow graphs of the selected methods, and for each synchronized node it determines the data type of the object used for locking the code block. Next, we distinguish two ways how the lock can be initialized: (i) the lock is only set inside the class; and (ii) the lock can also be set from outside the class. We say that a lock can be initialized from outside of the class if one of the following conditions holds:

- the lock is set via a parameter of the method that contains the synchronized code block;
- or the lock is defined as an attribute of the class that appears on the left hand-side of an assignment statement, and the right hand-side of the assignment is not a call to a constructor.

We added the last condition, since a call to a constructor always creates a new object that is usually not overridden from outside the class.

Finally, using this information we pairwise compare the synchronized nodes from two different methods (*i.e.*, control flow graphs). A synchronized block edge between two synchronization nodes is added, if the two locks have a compatible type according to the definition in [20], and if at least one of the following conditions is satisfied:

- both synchronized blocks are contained by methods of the same class and the locks reference the same class attribute;
- both locks are inherited from the same super class;
- one lock can be set from outside of the class and the other lock is performed on the `this` object;
- or both locks can be set from outside of the class.

The first condition handles implementations in which a lock is shared between methods of the same class. The second condition handles implementations in which a lock is inherited from the same parent class. The third condition handles implementations in which a synchronized block uses a lock that is initialized with an object that locks on itself. The last condition handles implementations in which the same object might be used to initialize the locks of two synchronized code blocks.

IV. Visualizing PCFGs-SI: For visualizing PCFGs-SI we use a leveled graph visualization in which nodes of the extracted PCFGs-SI are represented as ellipses and edges between nodes are represented as arcs. Starting nodes are represented as dashed ellipses. Black nodes represent ending nodes of the control flow. Dashed boxes represent synchronized code blocks. Solid arcs represent the control flow between statements. Dashed edges represent synchronized block edges. Maybe-synchronized method edges are drawn as dotted arcs. Strong solid arcs represent mustbe-synchronized method edges. We implemented the visualization using Graphviz.[3]

Figure 1 depicts an example of PCFGs-SI extracted from two Java classes `Thread_1` and `Thread_2` that extend the Java Thread class. We selected the `run()` methods in both classes to create the PCFGs-SI. Both methods use a class attribute `lock` of type `Object` to protect the access to their critical code blokcs. In both classes, the value of this attribute is initialized in the constructor. We add a synchronized block edge between the two synchronized nodes because the locking variables have a compatible type and satisfy the fourth condition - both locks can be initialized from outside of the class. Their value is set in the constructors, therefore there exists the possibility that both locks are initialized with the same object. A maybe-synchronized method edge between the nodes representing the invocation in Line 9 of the `Thread_1` class and the invocation in Line 7 of the `Thread_2` class is added, since they both implement a call to the `synchronized` method `init()` of class `Thread_2`.

IV. Evaluation

In this section, we present the evaluation of our approach to extract parallel control flow graphs with synchronization information from Java programs. For this, we developed a prototype tool that implements our approach and performed an empirical study with five selected open-source Java projects. With the results obtained from the empirical study, we aim to answer the following two research questions:

RQ1 What is the precision P_{SME} and recall R_{SME} of our approach to extract synchronized method edges? P_{SME} is measured by counting the number of correct and erroneous mustbe- and maybe-synchronized method edges extracted by our approach. R_{SME} is measured by counting the number of synchronized method edges missed by our approach.

RQ2 What is the precision P_{SBE} and recall R_{SBE} of our approach to extract synchronized block edges? P_{SBE} is measured by counting the number of correct and erroneous synchronized block edges extracted by our approach. R_{SBE} is measured by counting the number of synchronized block edges missed by our approach.

A. Experimental Setup

For the evaluation, we performed an empirical study with five selected open source Java projects. We chose Java because it offers language features to implement synchronized code blocks and methods. Furthermore, Java is widely used in a large number of open-source projects whose repositories are public available and easy to access. Prominent examples of

[3] http://www.graphviz.org/

```java
import Thread_2;
class Thread_1 extends Thread {
    Object lock;
    Thread_2 var;
    Thread_1(Object lock){
        this.lock = lock;
    }
    public void run(){
        int init = var.init();
        synchronized(lock){
            System.out.println("Thread 1: " + init);
        }
        Thread.sleep(1000);
        System.out.println("End");
    }
}
```

```java
class Thread_2 extends Thread {
    Object lock;
    Thread_2(Object lock){
        this.lock = lock;
    }
    public void run(){
        int init = init();
        synchronized(lock){
            System.out.println("Thread 2: " + init);
        }
        System.out.println("End");
    }
    public synchronized int init(){
        return this.hashCode();
    }
}
```

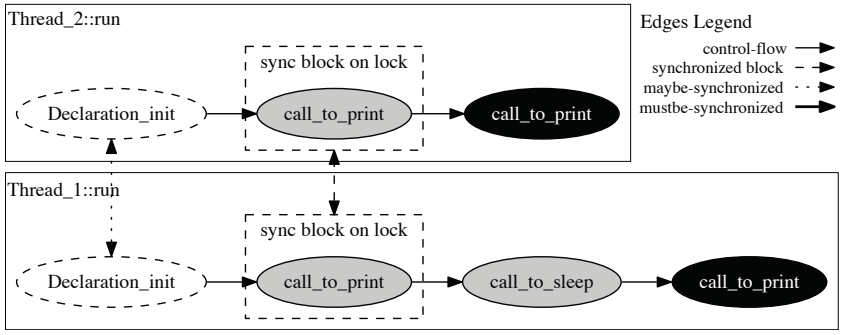

Figure 1: PCFGs-SI extracted from the two `run` methods of class `Thread_1` and class `Thread_2`. Dashed ellipse represent the starting node of the control flow and black nodes represent ending instructions. Dashed boxes represent synchronized nodes containing the set of instructions guarded by a lock.

Table I: Selected Java projects used in the evaluation with number of classes (NOC), number of methods (NOM), number of synchronized code-blocks (NSB), and number of synchronized method declarations (NSM).

Project	Version	NOC	NOM	NSB	NSM
ActiveMQ	5.13.3	2441	23708	717	478
Airavata	0.15	3326	26497	23	64
Jetty	9.3.9v20160517	1793	13935	321	116
Vuze	5721-04	4323	40323	2144	175
Wildfly-Core	3.0.0.Alpha9	3557	21924	346	328
Total		15440	126387	3551	1161

open source Java projects are the Apache projects or the Spring Framework[4]. We selected five open source projects that use the aforementioned Java synchronization mechanisms in their source code. Furthermore, we considered projects from different vendors and of different size. The selected five open-source Java projects are listed in Table I.

ActiveMQ[5] is a message broker and Airavata[6] is a software suite to compose, manage, execute, and monitor large scale applications and workflows on computational resources. Both are Apache projects. Jetty[7] is a web server provided by the Eclipse Foundation. Vuze[8] is a Java implementation of the bittorrent protocol. We decided to include Wildfly-core[9], an application server developed by JBoss, because it relies on multiple external libraries. Table I lists several descriptive statistics computed for the projects. The numbers show that the size of the projects varies from 1793 to 4323 classes, whereas Vuze is the largest project. Only a small fraction of these methods is declared as synchronized, which was expected. Furthermore, synchronized code blocks are used more frequently than synchronized methods except in Airavata. In particular, Vuze contains more than 2140 synchronized code blocks. In total, our sample comprises 3551 synchronized code blocks and 1161 synchronized methods, which we deem as representative for our study to obtain valid results.

Regarding the execution of the study, we first ran our prototype tool on the product code of each selected project and for each class extracted the information described in Section III-B. In addition, for each synchronized code block, we extracted the data type of the lock and its accessibility from inside and outside the class. For each method invocation the tool applies the heuristics presented in Section III-C to find out the list of potential called synchronized methods. Next, we used our tool to create the PCFGs-SI for methods that a) contain a synchronized code block and b) contain a

[4]https://github.com/spring-projects/spring-framework
[5]http://activemq.apache.org/
[6]http://airavata.apache.org
[7]http://www.eclipse.org/jetty
[8]http://dev.vuze.com
[9]https://github.com/wildfly/wildfly-core

potential call to a synchronized method. From these graphs, we output the list of extracted synchronization edges found between all pairs of methods. In total, our tool extracted 6068 synchronized method edges and 912177 synchronized block edges. For each edge, the list contains the fully qualified names of the two methods, the line number of the two synchronized code blocks or calls to the synchronized method(s), plus the type of the synchronized edge. This list and its information served as the basis for our manual analysis to compute the precision and recall of our approach. The manual analysis was performed by two co-authors of the paper using the source code navigation features of IntelliJ.[10]

B. Evaluation of Synchronized Method Edges

We calculated precision and recall of our approach to extract synchronized method edges. Remember, two method invocations need to be synchronized, if they call a method declared as synchronized from the same instance of a class.

First, for evaluating the precision, we randomly selected 400 from the 6068 synchronized method edges. This sample set allows us to obtain valid results with a 95% confidence level and 5% margin of error. For each synchronized method edge we used IntelliJ to manually analyze the source code of the corresponding classes to find out whether the extracted edge is correct. This manual analysis was done as follows: for each of the two invocations of a given edge we first used the "Go To - Declaration" feature to find the declarations of the two invoked methods. For each method declaration, we next obtained a list of all implementations of that method using "Go To - Implementations". From the list of methods we only kept the fully qualified names of the classes of methods that are declared as synchronized. Then, we checked whether the two lists of classes had at least one class in common. In addition, if the edge was a mustbe-synchronized method edge we further checked if the invoked methods are declared as static. The results of our manual analysis showed that our approach can extract both types of synchronized method edges with a precision (P_{SME}) of 100%. This was expected since all the information on these edges can be obtained statically from the source code.

Secondly, for evaluating the recall of our approach we manually investigated all 1161 synchronized method declarations from the five open source Java projects. For each such method, we used the "Find Usages of the base method" feature of IntelliJ to find all the usages, *i.e.*, invocations, of a synchronized method in the source code of a project. We searched for the usages of the base method to cover all usages of the method taking into account dynamic dispatching. This approach, however, also obtained invocations from the base method implemented in other subclasses. Therefore, we checked each returned usage if the synchronized method under analysis can be reached using the "Go To - Declaration" feature and, if not, we filtered it. The result was a list of verified usages for each synchronized method. Using this list, we built pairs of all possible combinations of the verified usages. For each pair, we then computed the PCFGs-SI of the methods that contains the two usages. Finally, we checked whether the two PCFGs-SI contained the expected synchronized method edge counting the number of found and missed edges.

Table II presents the outcome of our manual analysis. The two columns on the left show the number of methods declared as synchronized in each project and the number of calls to them. The largest number of synchronized method calls, namely 390, was found in the ActiveMQ project. The project Airavata has the smallest number of calls (48), but also the smallest number of synchronized methods (64). In the Wildfly-Core project, we observed that many of the 328 synchronized methods are mainly called from methods defined outside of the project, *e.g.*, listener methods or the `run` method of Java threads, therefore we only found 181 calls.

The two columns, *SE Expected* and *SE Extracted*, show the number of expected synchronized method edges and number of edges extracted by our approach which we used to compute the recall presented in the last column. Overall, the values for the recall show that our approach is capable of extracting synchronized method edges with average recall of 96.35%. For instance, ActiveMQ was the project with the highest number of calls to synchronized methods, namely 390. In this project, we manually found 1952 synchronized method edges out of which 1835 were correctly extracted by our approach, giving a recall of 94.01%. For Airavata, our approach extracted 205 out of the 207 expected edges.

Through the manual analysis we also discovered the different situations in which our approach failed to extract a synchronized method edge. Currently, our prototype tool does not handle expressions used in the inline initialization of arrays. These initializations are frequently used in ActiveMQ to initialize arrays of objects as parameter of calls to the logging function. Our prototype tool also does not handle implicit calls to the `toString()` method, for instance in the concatenation of strings. Several synchronization edges are not extracted because our prototype tool fails to resolve the correct types and consequently the methods used in chained method calls. In particular, if one of the types is declared in an external library. If it cannot resolve one method call, it stops to check the successive method calls. This situation most frequently occurs in Jetty, Vuze, and Wildfly-Core.

C. Evaluation of Synchronized Block Edges.

We addressed RQ2 by collecting all possible pairs of methods that contain a synchronized code block which share a compatible type for the lock. We excluded pairs that do not share a compatible type since we can be certain that they do not synchronize on the same object. Over the five open source Java projects, we collected 912177 pairs of methods that potentially synchronize on the same object. From this set of methods we randomly selected 400 allowing us to obtain valid results with a 95% confidence level and 5% margin of error.

[10]https://www.jetbrains.com/idea

Table II: Results of the evaluation of the recall for detecting synchronized method edges in the five open source Java projects.

Project	Sync. Method	Sync. Methods Calls	SE Expected	SE Extracted	Recall
ActiveMQ	478	390	1952	1835	94.006%
Airavata	64	48	207	205	99.034%
Jetty	116	95	551	525	95.281%
Vuze	175	156	817	807	98.776%
Wildfly-Core	328	181	652	617	94.632%
Total	**1161**	**870**	**4179**	**3989**	**96.346% (Avg)**

For each of the 400 pairs we used our approach to create the PCFGs-SI of each method. We then verified each pair by manually checking whether each synchronized block edge extracted in the PCFGs-SI was correct and whether there were no missing synchronization edges. For this, we analyzed the source code of the classes containing the two methods under analysis. Starting from the two synchronized statements, we manually reconstructed the data flow of the two locks and checked whether they could be initialized with the same object. We counted the number of correct and missing edges and based on these numbers computed the precision and recall.

Out of the 400 pairs, *i.e.*, edges, our approach correctly identified 70 edges resulting in a precision P_{SBE} of 97.50%. It missed to extract only 2 edges resulting in a recall R_{SBE} of 99.50%. These numbers clearly indicate that our approach covers the majority of situations in which different synchronized code blocks potentially share the same lock. Note, not all of the 400 pairs required a synchronization between them. In addition, the evaluation also pointed out the situations in which our heuristics did not work. The first situation concerns statements in which the return value of a method call is assigned to the lock. The problem is, that our approach currently does not analyze the implementation of the called method. For instance, if two synchronized code blocks are guarded by locks that have a compatible type and both are assigned a value returned by the call to the same method, our approach creates a synchronized block edge. Each call, however, might return a `new` instance therefore the extracted edge is not correct. The recall of our approach was also affected because it currently does not process the Java ternary operator `x = condition ? v1 : v2` correctly.

V. Applications of PCFGs-SI

PCFGs-SI show which parts of a software system synchronize the access to critical code blocks and methods. One application of our graphs is to help developers understand and document this aspect in multi-threaded programs. Furthermore, developers can analyze the PCFGs-SI extracted from their project and detect classes with shortcomings in the implementation of the synchronization. In this section, we present two examples taken from Vuze and Jetty to showcase how PCFGs-SI can be used to first, detect a potential race condition, and second to understand multiple synchronizations between two classes.

A. Detecting a Race Condition in Synchronized Code Blocks

In our first example, we use PCFGs-SI for detecting situations in the source code in which the same lock can be assigned to two different protected code blocks and thereby cause a race condition. We found an example of such a situation in the Vuze project in the classes `AzureusCoreImpl` and `TableViewImpl`. `AzureusCoreImpl` handles the core status of the whole Vuze application. It contains several methods that contain synchronized blocks using the keyword `this` as lock. The reference of a `AzureusCoreImpl` object is passed to all the plugins to allow the extension of its functionalities. The class `TableViewImpl` is used by developers to create a table for the GUI of a plugin. It contains several methods that contain synchronized blocks to synchronize the processing of table rows, using the attribute `rows_sync` that can be set via the public method `setRowsSync`.

Figure 2 shows an excerpt of the PCFGs-SI extracted for the two methods `canStart` of the class `AzureusCoreImpl` and `runForSelectedRows` of the class `TableViewImpl`. Each method contains a synchronized code block. The code block in `canStart` is guarded by `this` and the code block in `runForSelectedRows` is guarded by `rows_sync`. Since the type of `rows_sync` is defined as `Object` our approach detects a potential synchronization between the two code blocks and inserts a synchronized block edge. Consequently, there is the possibility that the developer uses an instance of the `AzureusCoreImpl` class as lock for the class `TableViewImpl`. This can create a race condition between the GUI of the plugin and the core of the application. A better solution would be to use a specific type for the `rows_sync` lock to prevent such a race condition.

B. Understanding Multiple Synchronized Method Calls

PCFGs-SI can help developers understand classes that share multiple synchronization edges. Figure 3 shows an example found in the Eclipse Jetty project. We present the PCFGs-SI extracted for the two `doStop` methods that share multiple method calls to static synchronized methods of the class `ShutdownThread`. The `ShutdownThread` class is implemented as singleton that maintains a list of socket instances registered with it. It also provides the functionality to close these sockets.

The two classes `ClientContainer` from package `org.eclipse.jetty.websocket.jsr356` and `WebSocketClient` from package `org.eclipse.jetty.websocket.client` use sockets and therefore contain several calls to the synchronized methods of the class `ShutdownThread`. The `ClientContainer` class handles connections from clients that use the `javax.websocket` API. Class `WebSocketClient` provides a means of establishing connections to remote websocket endpoints. In total,

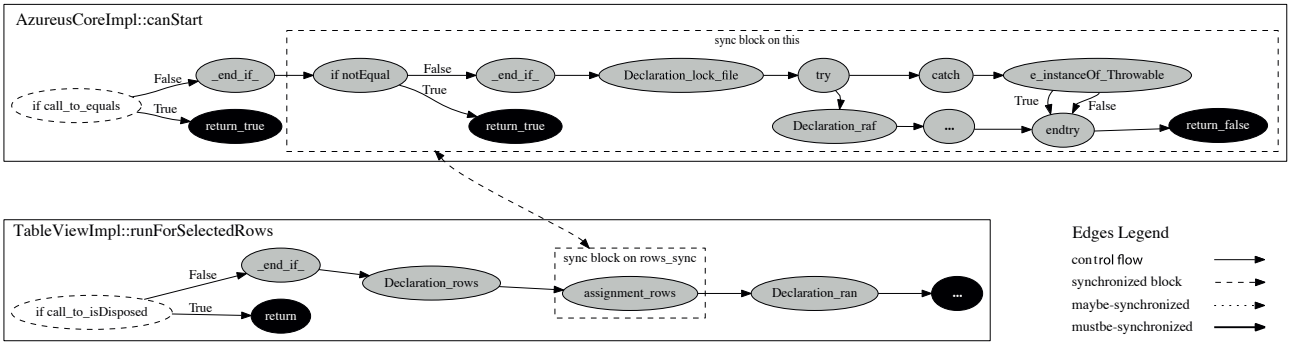

Figure 2: Excerpt of the PCFGs-SI extracted for the `canStart` and `runForSelectedRows` methods of the Vuze project showing an example of a potential race condition caused by using an object of the class `AzureusCoreImpl` as lock in the method `runForSelectedRows`. Nodes labeled with "..." represent groups of statements not relevant for the example.

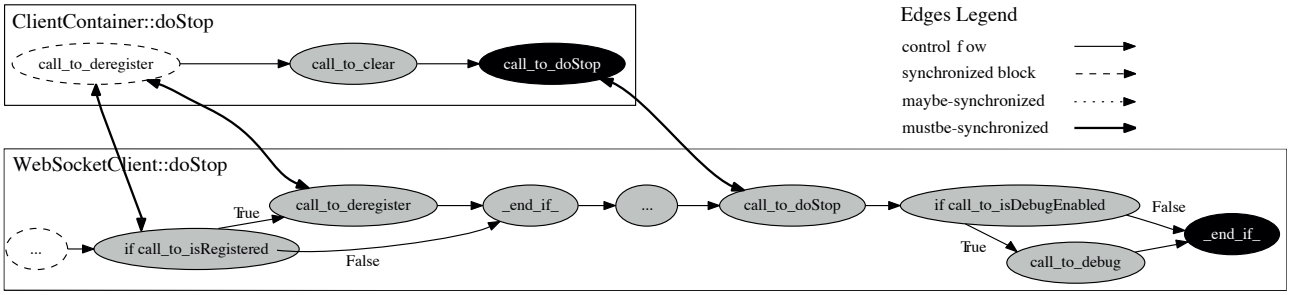

Figure 3: Excerpt of the PCFGs-SI extracted from the Jetty project showing three mustbe-synchronized method edges between the two `doStop` methods of the classes `ClientContainer` and `WebSocketClient`. Nodes labeled with "..." represent groups of statements not relevant for the example.

our approach detected 19 mustbe-synchronized method edges for these two classes. Figure 3 shows the three synchronized edges between the two `doStop` methods. The first two edges on the left hand side are detected since they represent calls to the static methods of the `ShutdownThread` class. The third edge represents the calls to the static `doStop` method of the super class `ContainerLifeCycle`.

VI. Discussion & Threats to Validity

This section discusses the results of our study and limitations of our current approach. Furthermore, we discuss potential threats to validity in our evaluation.

A. Discussion

We introduced a static analysis approach to overcome two relevant shortcomings of dynamic analysis, namely first, the program needs to be executable and second, execution traces typically do not cover all possible scenarios. In contrast to dynamic analysis, our approach can perform the extraction of synchronization information at every stage of the software development cycle. The only requirement of our approach is that the source code needs to be parseable. Since we do not need to handle multiple execution traces, we also argue that our approach is more scalable and easier to use than existing approaches based on dynamic analysis. This is based on the fact, that we need to handle less information.

On the other hand, dynamic analysis approaches might be more accurate for detecting race conditions because the execution traces contain the actual types instantiated and methods executed. However, we would like to point out that our static analysis approach showed to extract synchronized block edges with a precision of 97.50% and a recall of 99.50% and synchronized method edges with a precision of 100% and a recall of 96.35%. Furthermore, dynamic analysis might miss to detect some race conditions since not all possible execution traces that lead to race conditions might have been generated.

Compared to existing static analysis approaches, most prominently the Chord framework [8], our approach provides a comprehensive view on the synchronization aspect of Java programs. This view can be used by developers to detect shortcomings in the implementation, such as demonstrated with the first example in Section V, that can potentially lead to race conditions. The evaluation of the full benefit of our approach for developers, however, needs experiments with developers. This is subject to our future work.

The manual evaluation with the five open source Java projects also showed a number of limitations of our approach. Regarding the implementation, our prototype tool does not consider array initialization expressions and implicit calls to

the method `toString()`, for instance in string concatenations. Moreover, it does not parse libraries used in a project. For instance, if there is a chain of method calls and one call returns a type defined in a library, our tool stops to process the chain because it cannot resolve the specific method call. Furthermore, we currently do not apply a full-fledged data flow analysis. Such an analysis could help to further improve the precision for detecting synchronized block edges. We plan to address these issues in our future work.

B. Threats to Validity

In the following, we discuss several threats to the internal and external validity of our results and show how we addressed them in our experiments.

Internal Validity. The main threat to the internal validity is due to the implementation of our prototype tool. We mitigated this threat by thoroughly testing the prototype tool manually and with unit tests. For the manual analysis, we randomly selected 400 pairs of methods to evaluate the precision and recall of our heuristics to detect synchronized block and synchronized method edges. The size of our sample set is larger than the minimum number required to obtain results at a 95% confidence level with a 5% error. Moreover, we evaluated the recall of synchronized method edges with a manual investigation of every method declared as synchronized in the five open source Java projects.

The manual analysis was performed by two co-authors of the paper with the help of IntelliJ. Both co-authors have profound knowledge in Java and its locking mechanisms. Furthermore, IntelliJ is a well-known development environment that is supported by a large community and heavily used in industry, therefore we can safely assume that it is well tested and obtains correct results.

Another threat to the internal validity is that our approach currently handles only intrinsic locking mechanisms provided by the Java programming language. It does not handle explicit locks of the Concurrency API introduced in Java 1.5. We focused on intrinsic locking because they occur more frequently than explicit locks. In our five subject systems, out of the 15440 classes, only 41 classes (0.25%) use explicit locking while 741 classes (4.8%) use intrinsic locking of code blocks and 371 classes (2.4%) contain synchronized methods.

Finally, our current implementation uses a light-weight data flow analysis. Although this approach was sufficient to reach a high precision and recall, we are convinced that a full-fledged data flow analysis will further help to improve the precision. We plan to add such a data flow analysis in our future work.

External Validity. The results of our approach may not be generalized to other software projects. We mitigated this threat by choosing five open source Java projects of different size and of different communities to improve the validity of our study. Further studies with more systems are, however, needed to further mitigate this threat. Furthermore, we only investigated open source Java projects. Extending the studies to projects implemented with other programming languages, such as C#, is subject to our future work.

VII. Conclusions & Future Work

In this paper, we presented a static analysis approach to extract parallel control flow graphs with synchronization information (PCFGs-SI) from Java source code. While the control flow shows the execution of statements, synchronization edges represent points of synchronization between code blocks and methods. The main goal of our approach is to provide a comprehensive view on how different parts of the source code are using locking to schedule the access to critical code blocks, and to help developers to detect shortcomings in the source code that might lead to race conditions.

We introduced several heuristics for extracting the synchronization edges and manually evaluated them with the source code of five open source Java projects. The results show that our approach is capable of extracting synchronized block edges with a precision of 97.50% and recall of 99.50%. The precision and recall of extracting invocations of synchronized methods is 100% and 96.35%, respectively. Furthermore, we presented two examples of how a graph visualization of PCFGs-SI can support developers in detecting a potential race condition and understand multiple synchronized method calls.

Future work is mainly concerned with further improving the accuracy of our approach and heuristics. For this, we plan to address the limitations that we found in our evaluation. We also plan to add a full-fledged data flow analysis to reduce the number of false positives. Furthermore, we plan to consider explicit locking as provided by the Java Concurrency API and extend our approach to other programming languages, such as C#. Regarding the evaluation, we plan to extend our studies to other open source and industrial Java projects and projects developed in other programming languages. Finally, we plan to evaluate the usefulness of our approach in experiments with software developers.

References

[1] P. Dano and A. Bollin, "Down to hades and back - experiences gained in comprehending a distributed legacy system," in *Proceedings of the Interational Scientific Conference on Informatics*. IEEE, 2015, pp. 85–90.

[2] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering (FOSE)*. ACM, 2000, pp. 73–87.

[3] J. Joyce, G. Lomow, K. Slind, and B. Unger, "Monitoring distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 2, pp. 121–150, 1987.

[4] H. Garcia-Molina, F. Germano Jr, and W. H. Kohler, "Debugging a distributed computing system," *Transactions on Software Engineering (TSE)*, vol. 10, no. 2, pp. 210–219, 1984.

[5] N. Bhattacharya, O. El-Mahi, E. Duclos, G. Beltrame, G. Antoniol, S. Le Digabel, and Y.-G. Guéhéneuc, "Optimizing threads schedule alignments to expose the interference bug pattern," in *Proceedings of the International Symposium on Search Based Software Engineering (SBSE)*. Springer, 2012, pp. 90–104.

[6] J. Howarth, I. Altas, and B. Dalgarno, "Information flow control using the java virtual machine tool interface (jvmti)," in *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2010, pp. 689–695.

[7] G. Maheswara, J. S. Bradbury, and C. Collins, "Tie: An interactive visualization of thread interleavings," in *Proceedings of the International Symposium on Software Visualization (SOFTVIS)*. ACM, 2010, pp. 215–216.

[8] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2006, pp. 308–319.

[9] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer (STTT)*, pp. 366–381, 2000.

[10] H. Yuan and T. Xie, "Automatic extraction of abstract-object-state machines based on branch coverage," in *Proceedings of the International Workshop on Reverse Engineering To Requirements (RETR)*, 2005, pp. 5–11.

[11] T. Systä and K. Koskimies, "Extracting state diagrams from legacy systems," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer, 1997, pp. 272–273.

[12] Z. Yuan, Y. Ma, J. Bian, and K. Zhao, "Automatic enhanced cdfg generation based on runtime instrumentation," in *Proceedings of the International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, 2013, pp. 92–97.

[13] A. Amighi, P. d. C. Gomes, D. Gurov, and M. Huisman, "Sound control-flow graph extraction for java programs with exceptions," in *Proceedings of the International Conference on Software Engineering and Formal Methods (SEFM)*. Springer, 2012, pp. 33–47.

[14] J. Krinke, "Static slicing of threaded programs," *ACM Sigplan Notices*, vol. 33, no. 7, pp. 35–42, 1998.

[15] D. Kester, M. Mwebesa, and J. S. Bradbury, "How good is static analysis at finding concurrency bugs?" in *Proceedings of the Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2010, pp. 115–124.

[16] Y. Chen, Y.-H. Lee, W. E. Wong, and D. Guo, "A race condition graph for concurrent program behavior," in *Proceedings of the International Conference on Intelligent System and Knowledge Engineering (ISKE)*, vol. 1. IEEE, 2008, pp. 662–667.

[17] B. Tao and J. Qian, "Refactoring java concurrent programs based on synchronization requirement analysis," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 361–370.

[18] R. R. Prado, P. S. Souza, G. G. Dourado, S. R. Souza, J. C. Estrella, S. M. Bruschi, and J. Lourenco, "Extracting static and dynamic structural information from java concurrent programs for coverage testing," in *Proceedings of the Latin American Computing Conference (CLEI)*. IEEE, 2015, pp. 1–8.

[19] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[20] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification: Java SE 8 Edition*. Pearson Education, 2014.

[21] C. Hammer, R. Schaade, and G. Snelting, "Static path conditions for java," in *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 2008, pp. 57–66.

AAU-SERG-2017-001
ISSN 1872-5392