Static Analysis of Concurrent Java Programs

Toke Jansen Hansen & Bjarne Ørum Wahlgreen

Kongens Lyngby 2007
IMM-B.Sc-2007-11

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

In this thesis we introduce an approach to static analysis of concurrent Java programs. We analyze individual classes, to find any use of a class that breaks any thread-safety conditions within the class. We present properties for class-wise thread-safety and describe analyses capable of collecting adequate information to be able to detect violations of these properties.

The result achieved, is the ability to analyse a class for thread-safety as an over-approximation of how multiple threads may use the class and present the user with warnings corresponding to violations. The tool developed is able to run as stand-alone or integrated with Eclipse, where it generates markers for thread-safety violations in the editor.

Acknowledgements

First of all, we would like to thank our supervisor, Christian Probst, for agreeing to supervise this project, suggested by ourselves. Christian has great knowledge of program analysis and has guided us in choosing the right approaches and analyses to be able to achieve our results. We also thank Christian for his great help in reading and commenting the drafts of the thesis, which has been a great help in improving the final version.

The project also relies on concurrency theory, that is a foundation to be able to analyze a class for thread-safety properties. For this subject, Hans Henrik Løvengreen, has been kind to help us, when questions regarding concurrency and thread safety has come about, and we thank him for helping and pointing us in the right direction.

We would also like to thank the authors and developers of FindBugs, an analysis framework which our analyses are based on, for making this tool available along with an API and documentation. Without this tool, we could not have achieved the results we do.

Finally, we thank our fellow student Nikolaj Dalgaard Tørring for good company in the last couple of weeks before handing in this thesis.

Contents

Summary **i**

Acknowledgements **iii**

1 Introduction **1**

1.1 Motivation and Purpose 1

1.2 Structure and Overview 3

1.3 Delimitation 4

2 Background **5**

2.1 Program Analysis 5

2.2 Concurrency Theory 10

2.3 Java Analysis Frameworks 16

2.4 The Java Execution Model 21

2.5 Java and Synchronization Primitives 25

2.6 Class-wise Thread Safety 29

3 The Analyses 39

3.1 Our Approach 39

3.2 General Definitions 43

3.3 Points-To Analysis 45

3.4 Lock Analysis 59

3.5 Dominator Analysis 68

3.6 Concurrent Points-To Analysis 73

3.7 Applying the Analyses 76

4 Implementation 81

4.1 The Analyses 81

4.2 Detecting Bugs 87

4.3 Testing 88

5 Conclusion 91

5.1 Achievements 91

5.2 Applications 93

5.3 Future work 93

List of Notation 94

Bibliography 99

A README 103

B FindBugs XML 105

B.1 messages.xml 105

B.2 findbugs.xml 107

C Test cases 109

C.1 LockTryFinally.java 109

C.2 ReaderWriterLocks.java 111

C.3 ExposedStateVariables.java 113

C.4 ReaderWriterDeadLock.java 114

C.5 PublicNonFinalDispatch.java 116

C.6 DispatchTest.java 118

C.7 AssignmentCycles.java 119

C.8 ForLoopTest.java 121

C.9 PhiResolveTest.java 124

C.10 SynchronizedTests.java 125

C.11 ThisDeadlock.java 127

C.12 FieldAccess.java 128

C.13 LockOnLocalVariable.java 129

C.14 PublicFinalDispatch.java 130

C.15 GuardedVariable.java 131

C.16 SynchronizedMethod.java 132

C.17 LockOnNullReference.java 133

Chapter 1

Introduction

More and more focus is nowadays put into multi-threaded programming. However, caution must be made to avoid, e.g., race conditions in concurrent software. This project will focus on the Java programming language and aims at developing analyses capable of statically verifying thread-safety of Java classes individually, so they may be used safely in a multi-threaded environment.

1.1 Motivation and Purpose

The use of concurrency in applications is an increasing factor, and will be so for the forseeable future. Personal computers have moved rapidly from one processing core to currently dual- and quad-cores, and the chip-producers are agreeing that a sustained growth in processing cores will be the main factor in keeping up with Moores law in the near future of processors. To benefit from the increase in the number of processing cores, software must keep up with the pace and focus on concurrency, to spread the workload on multiple processors. However, the discipline of writing concurrent software requires special skills, that developers must learn and improve, at least until new language technologies arise that may be able to infer automatic concurrency to sequential programs. The latter has undertaken research for ages without reaching usable results.

In Java among other languages, concurrency is supported, e.g., through the use of threads. However, introducing threads in a program drastically increases develop-

ment complexity, as synchronization issues must also be accounted for. These issues have been known and researched extensively for many decades, introducing several tools and analyses capable of verifying concurrent applications or models. Although few tools have made it to the compile-time analyses performed by the Java compiler; these tools are often too time consuming and increases too fast in computation complexity with the number of concurrent threads. In the research projects presented in [17, 19, 23], the general approach is based on identifying threads from the program context. Other approaches have also been applied, e.g., the paper [22] discusses a method for component-based concurrency testing of the readers-writer problem, and combines static analysis with code inspection and dynamic testing, to do so. In other research work [14], a model specification of concurrency is applied the program source by annotating code with temporal logic expressions. This, however, rely on the developers' ability to express correct temporal logic representing the desired concurrent behavior and has to address the state-explosion problem, that model-based exploration tools suffer from.

We take a different approach, by not identifying concurrent threads, but instead to statically analyze a class from the point of view of a Strongest Possible Attacker (onward referred to as SPA), that is, a person that will use the class in as many threads he likes, using all visible constructs the class may provide, in any way. We identify different unwanted synchronization scenarios found by our tool, e.g., deadlocks, raise warnings and present them to the developer, just like many integrated development environments (IDE's) notify the developer on errors while writing programs. In our case, the warnings are visualized in the widely adopted open source IDE, Eclipse.

A main purpose of our work, has been to develop the tool such that it proves as useful and easily integratable in existing environments, like Eclipse, as a stand-alone tool or even as extensions to a compiler. We develop the tool capable of analyzing a large set of Java programs, using `synchronized` and `java.util.concurrent.locks.Lock` as synchronization primitives, whereas `java.util.concurrent.Semaphore` and other means of synchronization is left for future work.

One important aspect of our approach and a deviation from many projects on this subject, is that we do not attempt to verify programs in their completely accurate behavior. Instead we over-approximate on the program behavior, according to the SPA approach, which will analyze on every possible (mis)use of the class being analyzed to reveal potential unwanted behavior. Our tool will warn the developer, which may or may not react to the warning, according to his conviction about the actual program behavior. The advantage of this approach is, that if our tool does not generate any warnings, the analyzed class is thread-safe according to our specification of thread safety, which is introduced later in section 2.6 page 29.

It is our conviction, that too few concurrent analyses have made it from theory to applied tools, that developers may benefit from. Therefore we shall not only concentrate on describing theory and abstract descriptions of the analyses, but emphasize the implementation of the analyses.

1.2 Structure and Overview

In Chapter 2, we introduce main concepts that is the foundation of the work in this project. This involves some insight into the disciplines of *program analysis*, more specifically *dataflow analysis* and *lattice theory*. Then follows some basic *concurrency theory*, which introduces the concepts of *threads*, *processes*, and how parallel and concurrent execution may interleave according to the *interleaving model*, which is the main model of concurrency applied in the analyses.

The chapter proceeds by describing a number of different analysis frameworks for Java and continues with a description of the *Java Execution Model*. This introduces Java *bytecode* and aspects of the *Java Virtual Machine*, which are technologies our analyses depend on. Ongoing, the chapter then introduces the *synchronization primitives* in Java, which are basically the constructs for achieving synchronization between threads in Java and are constructs that our analyses must be aware of, to give proper approximations of program behavior.

Finally, the chapter introduces the main guidelines for class-wise thread-safety in Java, which defines the properties that our analyses target to investigate.

In Chapter 3, the analyses we have developed, are described in detail. To start with, we document the approach that our tool follows in determining thread-safety of Java classes. This leads to a derivation of the analyses we shall describe and the interdependencies of these. First of all, we identify the need of a *points-to analysis*, that is *flow-sensitive* and *intra-procedural*. This analysis first of all has the purpose of making a *lock analysis* possible. The lock analysis shall determine which locks are held at a given location, both with certainty and possibly. The points-to- and lock analyses are the foundation of a *concurrent points-to analysis*. The concurrent points-to analysis has the task of collecting points-to information from the points-to analysis, which is merged into a set representing the information about what objects may point to, when accounted for concurrent use of the class of interest. The merging of information into the resulting *concurrent points-to set* is conditional on the locks held at program points and a fourth analysis, the *dominator analysis*, which is a prerequisite of the concurrent points-to analysis and depends on the points-to and lock analyses. In Figure 1.1, the interdependency and order of the analyses applied is illustrated.

Chapter 4 describes some details of how the analyses have been implemented in Java, using *FindBugs* as analysis framework. The structure of the implementation is sketched and the main challenges are mentioned and discussed. The chapter also introduces the *detectors* that utilize the information our analyses compute to reveal thread-safety violations on a class-wise level. The detectors are the actual instances that reveal thread-safety violations and report them, however, they are small, simple programs that collect and compare the information from the analyses to detect violations. The reporting of violations found by the detectors are integrated into Eclipse,

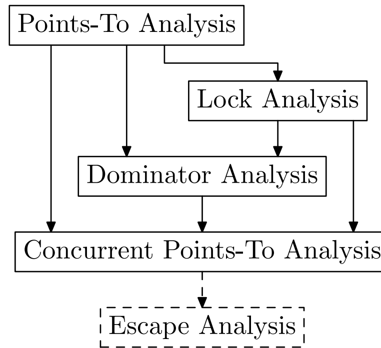

Figure 1.1: The illustration shows the dependencies of the analyses we will apply. The escape analysis is left for future work.

such that markers appear in the classes marked as target of the analyses describing the kind of thread-safety violation.

To verify the implemented analyses, we have developed a testing framework, which is described in the end of Chapter 4. The testing framework is based on JUnit, that functionally tests a number of classes and compare output from the detectors to expected output. The format outputted from the detectors is XML which allows DOM-based comparison of the output and expected output.

Finally, our achievements and suggestions to future work is presented in Chapter 5.

1.3 Delimitation

Our tool will support analyses of classes using the synchronization primitives offered by the `synchronized` construct, and classes extending `java.util.concurrent.locks.Lock`, including readers-writer locks. However, we delimit this project not to include the `java.util.concurrent.locks.Semaphore` synchronization primitive in the analyses. Although, this mechanism is easy to integrate in future work, as it does not introduce any differences to what regions of code that would be mutually exclusive, based on the semaphores held. It requires to count the number of locks held, which we already do take account for in the analyses.

We focus this project on the use of instance methods and instance variables, and do not analyze static methods or variables. However, future work can extend the functionality of our analyses to also support static methods and variables. The properties that we state for class-wise thread-safety then also have to take static methods and instance variables into account.

Chapter 2

Background

In this chapter we establish the foundation of theory concerning the analyses we develop. The background theory is widespread from program analysis, lattice theory, and concurrency theory to specific technologies in the Java Virtual Machine and language constructs and mechanisms in the Java programming language. In the end of this chapter, a foundation of principles concerning thread-safety regarding Java classes is established. These properties are the main targets to be investigated by the tool.

2.1 Program Analysis

In this project, our approach is to analyze classes statically, that means analyzing the program without running it, in opposition to dynamic analysis, where the program is executed and the change of values evaluated at runtime. Here we outline the static analysis technique of dataflow analysis, which is the main technique of static analysis we apply. Our outline is an deduction from the concepts covered thoroughly in [24].

2.1.1 Dataflow Analysis

Dataflow analysis is a static analysis technique which collects an approximation of information of interest present at a given program point in a program. We shall

call a unique program point a *location* in the following. It does so by traversing a ***control flow graph*** (**CFG**), a graph representation of the program. A control flow graph is a directed graph, where the nodes, called vertices and referred to as $\boldsymbol{V}$, are usually *basic blocks* - linear sequences of code without jumps and with jump targets as the first instruction and jump instructions as the last. Edges, referred to as $\boldsymbol{E}$, represent the control flow - they connect the jump instructions to the jump targets with conditions on the edges. The mathematical definition of such a graph can be expressed:

$$\langle V, E \rangle \text{ where } E \in V \times V$$

As information may propagate differently through various parts of the CFG, the information collected at a given program point may be undecidable at compile-time. Therefore dataflow analyses are approximations on what information may or must reach specific locations at runtime.
Dataflow analyses are often formulated as a set of dataflow equations for each node in the CFG and calculating the output for each node, based on its input. An iterative algorithm is then usually applied, to recalculate the dataflow equations as long as information change. Consequently, the dataflow equations must be guaranteed to reach a point where the information no longer changes, such that the dataflow analysis eventually terminates. How this can be achieved, follows from concepts of *lattice theory*, which dataflow analyses are based on.

2.1.1.1 Lattice Theory

A *partially ordered set* $L = (S, \sqsubseteq)$, consists of a set, S, and a *partial ordering*, $\sqsubseteq$, a binary relation over a set S, that respects the following conditions:

$$\begin{aligned} &\forall x \in S : x \sqsubseteq x && (Reflectivity) \\ &\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z && (Transitivity) \\ &\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y && (Antisymmetry) \end{aligned}$$

Now, for the set $X \subseteq S$, we say that $y \in S$ is an *upper bound* for X, written $X \sqsubseteq y$, if $\forall x \in X : x \sqsubseteq y$. Similarly, $y \in S$ is a *lower bound* for X, written $y \sqsubseteq X$, if $\forall x \in X : y \sqsubseteq x$. A *least* upper bound of a set X, written $\sqcup X$, is defined as:

$$X \sqsubseteq \sqcup X \wedge \forall y \in S : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$$

Likewise, the *greatest* lower bound for X, written $\sqcap X$, is defined as:

$$\sqcap X \sqsubseteq X \wedge \forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$$

When $\sqcup X$ and $\sqcap X$ exist for all $X \subseteq S$, they must be unique (follows from the antisymmetry of $\sqsubseteq$) and we call $\boldsymbol{L}$ a *lattice*. For a lattice $L = (S, \sqsubseteq)$, a *greatest*

element can always be introduced as $\top = \sqcup S$ (a.k.a. *top*) and equivalently the *least* element as $\bot = \sqcap S$ (a.k.a. *bottom*). It is common in program analysis that $\sqcup$ is called the *join operator* and $\sqcap$ the *meet operator*. We call a lattice containing unique top and bottom elements a *complete lattice*, and write it as $L = (S, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$.

Monotone Functions. A function $f : L_1 \rightarrow L_2$ between partially ordered sets $L_1 = (S_1, \sqsubseteq_1)$ and $L_2 = (S_2, \sqsubseteq_2)$ is monotone if $\forall x, y \in L_1 : x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y)$. Notice that the operations $\sqcup$ and $\sqcap$ are monotone. The result of compositions of monotone functions yields another monotone function.

Fixed Points. As we mentioned about dataflow analyses, we must ensure that computations will terminate at some point, as a result of all information eventually stabilizing. In practice, a result from lattice theory helps us achieve that this can be accomplished. A *fixed point* of a function $f \rightarrow L \times L$ on a complete lattice $L = (S, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is an element $x \in S$ such that $f(x) = x$. *Tarski's Fixed Point Theorem* states that this set is lower and upper bounded, given the function f is monotone. The proof of this theorem can be reviewed in [24].

2.1.1.2 Lattices in Dataflow Analysis

In dataflow analysis we consider the information at a specific point in the CFG a lattice. The information is calculated from the information of other nodes in the CFG, usually either the information from input or output edges. When the information calculated on a specific point in the CFG is dependent on the information from the input edges, we say that the dataflow analysis is a *forward* dataflow analysis and, when it depends on the information from the output edges, a *backward* dataflow analysis. Depending on the combine operator, sometimes analyses are identified as a *must* analysis when $\sqcup$ is defined as the $\cap$ operator, or a *may* analysis when $\sqcup$ is defined as the $\cup$ operator. The characteristics of each is:

- **May** analyses calculate the information that may reach a certain destination.
- **Must** analyses calculate only the information that will definitely reach a certain destination.

The functions that calculate the information flowing to and from a node, v_i, can be expressed as:

$$
\begin{aligned}
IN(v_i) &= \bigsqcup OUT(v_n), \text{ where } v_n \text{ is a neighbor of } v_i \\
OUT(v_i) &= f_\ell(IN(v_i))
\end{aligned}
$$

$\bigsqcup$ is defined according to the type of the dataflow, e.g., $\sqcup = \cup$ for a may analysis. However, the operator $\bigsqcup$ need not necessarily to be either $\bigcup$ or $\bigcap$ (and thus, $\sqcup = \cap$ and $\sqcup = \cup$, respectively), as the type of analysis may require a custom operator, which then will have to be defined specifically. The neighbors are either the immediate predecessors for a forward analysis, or the immediate successors for a backward analysis. The function f_ℓ, called the *transfer function*, is based on the actual information from the node.

These functions are monotone, and as a result of the fixed point theorem we know the existence of a *least* fixed point. That means we can apply an iterative approach that terminates when the output functions do no longer change on recomputations.

2.1.1.3 Interprocedural Analysis

Until now we have stated that dataflow analyses traverse a CFG. As already mentioned, CFGs are abstract representations of programs. In a language with procedure calls, like methods in Java, CFGs are usually constructed for the body of each procedure, where the linear sequences of code pieces without jumps make up the nodes, and control flow edges are the conditions for jump instructions, pointing to jump targets (or simply fall through). The dataflow analysis targets are then the CFGs for each method in a program or class. Meanwhile, within one CFG, another procedure might be invoked, and the analysis will have to analyze the CFG of the called method to be able to compute the approximation of flow of information. This type of dataflow analysis is called *interprocedural* analyses, in contrast to *intraprocedural* analysis, where procedures either do not exist or do not effect the information the analysis is computing. In the following, we present some of the concepts for interprocedural analysis.

Control Flow and Procedures. To be able to approximate the information flow after a given location in a CFG where a procedure invocation is performed, either a *context-insensitive* analysis or a *context-sensitive* analysis may be carried out. A context-insensitive approach collects all information possible from a CFG of a procedure, independent on the calling context, e.g., which parameters are passed as arguments. Of course this must approximate the arguments by using some abstraction, as there is no way to analyze all possible arguments passed. The advantage is that the dataflow analysis only has to be performed once for each procedure. Computation results can then be combined into the calling context on procedure invocations. The disadvantage is, that it does not approximate program behavior very well. Therefore, another concept, *context-sensitive* analysis, comes handy. In context-sensitive analysis, a set representing context information of some sort is computed through the CFG, and is then passed on as initial analysis information, when a procedure invocation requires another CFG to be analyzed. Thus a better approximation on the program behavior is achieved, but at the cost of potentially recomputing information flow in

the same method over and over again, with different context information parsed as parameter. So a trade-off must be considered to balance efficiency and precision, when choosing an appropriate approximation.

Flow-sensitivity versus Flow-insensitivity Up to now we have only considered dataflow analyses *flow-sensitive*, meaning the computed information of interest has been dependent on the exact order of the locations being analyzed. Sometimes, a *flow-insensitive* approach can be a sufficient approximation for the information that is the target of investigation. That means, the order in which locations are being analyzed does not influence on the information being computed.

2.1.1.4 Intraprocedural Control Flow Analysis

The Java compiler transforms Java source code into bytecode, which is a low-level intermediate representation of Java programs that serves as instructions for the Java Virtual Machine. To be able to perform dataflow analysis on such a representation, it is necessary to initially run an *intraprocedural control flow analysis*. It computes the CFG of each procedure by grouping linear sequences of instructions without jumps in nodes and create the flow relations between the nodes from the jump targets.

2.1.1.5 Static Single Assignment Form

For program analysis it can sometimes be convenient and more accurate to transform the analyzed context into an intermediate representation called *static single assignment* (**SSA**) form. The outcome of this transformation is that each variable is, at any program location, assigned at most once. This reflects, that at runtime, variables will have at most one definition at any location, despite different definitions on different flows.

For dataflow analyses, such a representation comes in handy, e.g., if the information of interest (or context information) at a certain location should represent what a variable definition is at that location. Without some sort of abstraction in the dataflow analysis, different flows may reveal that the variable potentially points to several definitions. Using SSA form, we can be sure that the variable is at most assigned one of the definitions at runtime, and for analyses were this information improves efficiency or simplicity, we can abstract the dataflow information, such that it represents that the variable definition is at most one amongst several. This is usually done by introducing so-called φ-functions, where each argument position of the function represents a definition as the outcome of a specific program flow. Information can be brought in the φ-function to represent the exact flow a certain definition is the result from, by identifying flows and inserting definitions, such that the ith definition corresponds to the flow identified by i. The φ-functions need not necessarily be implemented as

functions, but may just be simple types mapping to arrays holding the possible set of definitions.

2.1.1.6 Dominator Theory

For a directed graph $G = \langle V, E \rangle$, such as a CFG, we say that a node v_d dominates a node v_i, if all paths to v_i leads through the node v_d. Mathematically written:

$$Dom(v_o) = (v_o)$$
$$Dom(v_i) = \left(\bigcap_{v_n \in predecessors(v_i)} Dom(v_n) \right) \bigcup (v_i)$$

where v_0 is the root node and $v_i \neq v_n$. A few definitions are suitable:

- A node v_d *strictly dominates* a node v_n if v_d dominates v_n and v_d does not equal v_n.
- The *immediate dominator* of a node v_n is the unique node that strictly dominates v_n, but does not strictly dominate any other node that strictly dominates v_n.
- The *dominance frontier* of a node v_n is the set of all nodes v_i such that v_n dominates a predecessor of v_i, but v_n does not strictly dominate v_i.

The latter definition, dominance frontiers, can be utilized to compute the exact locations where φ-nodes should be inserted, when transforming into SSA form. The reason why, is that from v_i's point of view, the set of all nodes v_i in the dominance frontier are the nodes at which other control flow paths that do not go through v_i make their earliest appearance, and thereby the dominance frontiers are the exact locations, where different definitions may reach, thus the candidate locations for creation of φ-functions.

2.2 Concurrency Theory

A concurrent program can be defined as a program where multiple interacting computational tasks execute simultaneously. These tasks may be implemented as separate programs, processes, or threads within a single program.

On many computer systems the tasks may execute in parallel, however true parallelism is difficult to realize because that would require as many processors as the number of running tasks. Parallelism is therefore often obtained by time-slicing one

or more processors, where the operating system is in charge of scheduling the execution of tasks. Tasks may also be distributed across a network and thereby be executed on a remote host. A consequence of the way that tasks may be executed is that one can never know when a task is scheduled for execution, which leads to a general rule in concurrent programming:

The speed of execution of each task is unknown.

In non-trivial programs tasks may need to communicate with each other to share information. The way that tasks communicate can be divided into the categories:

- **Shared memory**
  The concept of shared memory means that tasks may access and share the same memory. The communication between tasks is therefore done by altering some shared variable that will become visible to other tasks at some later point. This style of concurrent programming is often achieved through the use of threads running within a single program. Because variables are shared, applications that utilize threads often apply some form of locking to coordinate access to shared variables and thereby to preserve program invariants. In Section 2.5 we cover the synchronization primitives provided by the Java platform.

- **Message parsing**
  The concept of message parsing allows tasks to communicate by exchanging messages, where the exchange may be done both synchronously (blocking) or asynchronously (non-blocking). There exists a number of models for modeling the behavior of systems using message parsing, e.g., rendezvous may be used to model blocking implementations, in which the sender blocks until the message is received. Implementing concurrency based on message parsing has the advantage of being easier to reason about than shared memory, because such implementations do not share address spaces. However, these suffer from a higher overhead than implementations based on shared memory. Message parsing is for example used by Unix processes that communicate using pipes.

What we call a “task” is in classic concurrency theory denoted by a “process”. A process is defined as a distinguished program part that is to be executed independently. In the sequel we will use the classical notation of a process.
We will now turn our attention to the modeling of concurrent systems.

2.2.1 Modeling Concurrent Behavior

Concurrent programs are said to be *reactive* because they express some activity rather than the computation of a final result. Properties of a concurrent program can be divided into two informal categories:

- **Safety properties**
  Properties that ensure that the program does nothing wrong.
- **Liveness properties**
  Properties that ensure that the program makes progress. Parring Liveness properties with the safety properties imply that the program does something good.

In order to make it easier to model and prove various properties of concurrent systems, different models have been developed. In this section we introduce two models usefull for modelling concurrent behavior, namely the Petri Nets and the interleaving model.

Petri Nets. A Petri Net is a bipartite, directed graph that can be used to model discrete distributed systems. This means that the model is not limited to concurrency in computer systems, but can be used to model any kind of system where things may happen simultaneously. A Petri Net consists of *places*, *transitions*, and *arcs*, where the arcs connect places and transitions. Places in the net may contain zero or more tokens, and a distribution of tokens over the places is called a marking.
A transition acts on input tokens by "firing", meaning that the transition consumes the token from its input places, performs some processing task, and places a specified number of tokens into each of the output places belonging to the given transition. The firing process is performed in a single, non-preemptible step, thus atomically.
A transition is enabled if it can fire, which is only possible if there are enough tokens in every input place. Enabled transitions can fire at any time, and happens in a non-deterministic manner, meaning that multiple transitions may fire simultaneously, making Petri Nets usable for modeling concurrent behavior.

A Petri Net can be represented in mathematical terms of the tuple $\langle P, T, F, M_0, W \rangle$, where:

$$\begin{aligned}
P &: \text{Is a set of nodes called } \textit{places}. \\
T &: \text{Is a set of nodes called } \textit{transitions}\text{, where } P \cap T = \emptyset. \\
F &: \text{Is a relation called a } \textit{flow}\text{, where } F \subseteq (P \times T) \cup (T \times P). \\
M_0 &: \text{Is a set of initial } \textit{markings}\text{, with } M_0 : P \to \mathbb{N} \\
&\quad \text{and } \forall p \in P \text{ there are } n_p \in \mathbb{N} \text{ tokens.} \\
W &: \text{Is a set of } \textit{arc weights}\text{, with } W : F \to \mathbb{N}^+ \text{ which assigns each arc } f \in F \\
&\quad \text{some } n \in \mathbb{N}^+ \text{that denotes how many tokens are consumed from a} \\
&\quad \text{place by a transition, or alternatively, how many tokens are produced} \\
&\quad \text{by a transition and put into a place.}
\end{aligned}$$

The state of a Petri Net is represented as a vector M, where the initial state is given by M_0. If an enabled transition t is fired, the marking M evolves into M', where the

ability for t to fire is denoted by $M \xrightarrow{t} M'$. Figure 2.1 shows an example of a Petri Net where the state is given by $M^T = (1\ 1\ 0\ 0)$.

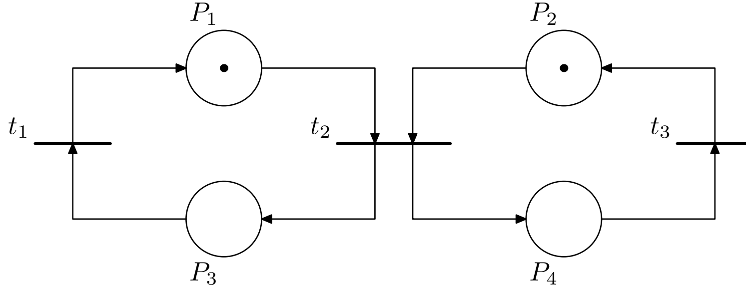

Figure 2.1: An example of a Petri Net with initial state given by $M^T = (1\ 1\ 0\ 0)$. Note that t_2 may only fire if there is a token in both P_1 and P_2.

As already described, a transition $t \in T$ can only fire if there exist enough tokens in the input places. This we formalize as:

$$\forall p \in P,\ f = (p, t) \in F\ :\ W(f) \leq M(p)$$

Because of the conditions that are necessary for a transition to fire, many synchronization mechanisms can easily be modelled by using Petri Nets. One of the strengths of the Petri Net model is that many people find the graphical representation appealing.

Petri Nets were and are widely used to model concurrent systems. However they suffer from an important limitation; they can model control flow but not data flow.

The interleaving model. By using the classical notion of a process, we can model the execution of a process as a sequence of states that a program must go through. For a given process we can model the program flow by using a finite or infinite sequence of the form:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots$$

Here s_0 is the initial state and the $a_i's$ are actions. The execution of a_0 will change the state of the process from s_0 to s_1, etc. If the actions of a process are always executed without any overlapping in time, the process is said to be a *sequential process*.

In the interleaving model, a concurrent program is considered to be composed of two or more sequential processes. Because the actions in each of the sequential processes may be executed overlapping in time, we introduce the concept of an *atomic action* meaning that an action seen by other processes appears to be performed

indivisibly, thus no other process will be able to detect any intermediary states during its execution. This implies that if two or more actions are executed overlapping in time, the resulting state would be the same as if the actions were executed in some sequential order.
We now define the term *mutually atomic*, meaning a group of actions that overlap in time has the same effect as if they were executed in some sequential order. A concurrent program is said to have atomic actions if any selection of actions from different processes are mutually atomic. In the interleaving model we assume that all actions in processes have been chosen so that they are atomic, which is a nice property because it allows us to model all executions of a program simply by taking all possible interleavings of all possible sequences of actions of the processes. Even though the property enables us to calculate all possible program executions, the computation explodes as the number of interleavings in the processes increase.
By letting i_n denote the number of atomic actions in the $n'th$ process, we can express the total number of interleavings by:

$$\frac{(i_1 \cdot i_2 \cdot i_3 \cdot ... \cdot i_n)!}{i_1! \cdot i_2! \cdot i_3! \cdot ... \cdot i_n!}$$

In most cases there are simply to many interleavings in a concurrent program to make a complete analysis based on these. Therefore when analyzing a concurrent program using the interleaving model, some abstraction is usually needed in order to reduce the state space.
In Java even a simple operation like `i++` is not atomic, therefore the developer must be aware of how the compiler transforms code in order to know what he can assert being atomic. The `i++` operation actually consists of three actions, namely reading the value of `i`, increasing the value by one and finally storing the result in `i`. Figure 2.2 shows an example of a possible interleaving where processes, P_1 and P_2 increment the same variable.

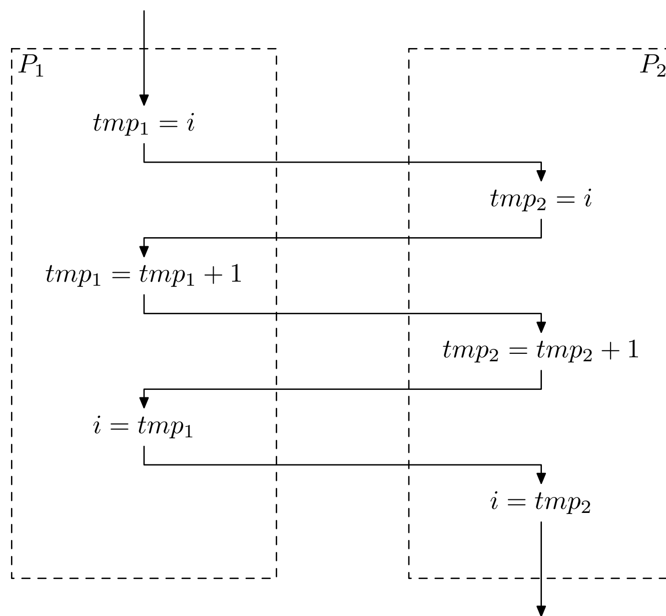

Figure 2.2: Illustrates two processes, P_1 and P_2 incrementing a shared variable i, without the necessary synchronization. If the initial value is $i = 0$, then the outcome for the above interleaving will be $i = 1$, whereas other interleavings may result in $i = 2$ alse.

2.3 Java Analysis Frameworks

To be able to apply the analyses developed in this project, we shall benefit from the existence of several analysis tools that are currently available for Java. We have investigated several frameworks to understand their possibilities and how these could support the analyses we shall develop. Rather early in our work we had to choose among the tools, to be able to develop the analyses within the available time, and therefore we were highly depending on making the right choice. Although we have chosen only one framework, our analyses would probably have been possible to apply based on one or more of the other tools presented. Common to all the frameworks is that they are all implemented in Java. For our analyses we need to develop dataflow analyses, and therefore we emphasize on the different frameworks' ability to support the development of custom dataflow analyses.

2.3.1 Soot

Soot[8, 17, 27, 26, 28] is a free analysis framework for Java that can be used to analyze, optimize and transform Java source code and Java bytecode. The user can choose between four intermediate representations of Java programs that the tool can operate on:

- **Baf.** A streamlined representation of bytecode which is simple to manipulate.
- **Jimple.** A typed 3-address intermediate representation suitable for optimization.
- **Shimple.** An SSA variation of Jimple.
- **Grimp.** An aggregated version of Jimple suitable for decompilation and code inspection.

For more information on the workings of Soot in general you may inspect [17], where Soot is applied with Jimple. Further information on the other intermediate representations can be found in [27, 26] and of course in the documentation of Soot [8].

Soot comes with a number of built-in analyses and an Eclipse plugin. Furthermore, an external points-to analysis PADDLE [28] can be downloaded and installed, for use with Soot. As already mentioned, Soot can operate on both Java source code and Java bytecode. However, operating on Java source code only fully supports Java 1.4. As we strive to be able to apply the analyses on all current Java platforms, we therefore decides not to use Soot on Java source code. But it still leaves the option of operating on Java bytecode, as newer platforms of Java use the same bytecode instructions as the previous platforms.

The documentation of Soot is mainly based on a collection of papers and a poor API documentation, and we thought it was hard to get familiarized with. Also, Soot's source code is still based on Java 1.4 and therefore does not use generics, which we would very much prefer - it diminishes a great deal of type casting and makes common behaviors easier to generalize under common types.

Soot has a little awkward approach to raise warnings upon identifying violations during an analysis. The analyses in Soot analyze Jimple code and upon violations add tags to the Jimple code to represent the type of violation. Warnings are then not raised until traversing the tagged Jimple code after the analysis finishes. To create dataflow analyses in Soot, one must initially *jimplify* binary Java class files, that means translate bytecode into the Jimple representation. Basically the jimplification is an intraprocedural control flow analysis that comes built-in with Soot. Afterwards the developer should take the following steps:

1. Create a class derived from `Transformer`. This class uses the dataflow analysis to add tags to Jimple.
2. Create a class derived from `FlowAnalysis`. This class provides the flow functions and provides the lattice functions.
3. Instantiate a `FlowSet`. This class is solely data for nodes in the lattice and does not include any functionality to merge or copy data.

This abstraction somewhat resembles the theoretical dataflow abstractions, however it is split up slightly different. Also Soot does not use the visitor pattern, so the developer must do iterations over the AST abstractions on the respective levels.

As we were in the process of selecting a framework for our analyses, Soot did not support metadata, such as runtime-visible annotations in the bytecode. However, this functionality has been added in the recent (and long-anticipated) release of Soot.

Soot is distributed under the GNU Lesser General Public License[7] and can be downloaded from [8].

2.3.2 BCEL

BCEL[3], the Byte Code Engineering Library, is, as the name suggests, a bytecode engineering library. Basically, that means it operates on compiled Java classes (.class) by inspecting bytecode instructions. The BCEL API can be divided into the following categories:

1. Classes that describe "static" functionality of a Java class file, i.e., constraints that reflect the class file format and are not intended for bytecode modifications. The classes enable to read and write class files from or to a file, which

is especially useful for static analysis of Java classes from bytecode. One can obtain methods, fields, etc. from the main datastructure called `JavaClass`.

2. Classes that enable modification of such `JavaClass` or `Method` objects, another common datatype representing methods in a Java class. These classes can be used for code injection or optimizations, e.g., stripping unnecessary instructions.

3. Examples and utilities.

Basically, what BCEL offers is datatypes for inspection of binary Java classes. It does not come with analyses, such as dataflow, control flow or points-to analyses, which makes it very little helpful for our purpose, as we would like to benefit from a framework that offers such functionality.

BCEL is fairly well documented, but there has not been a lot of development for the past few years, and a more recent project ASM has come to life, matching and surpassing the functionality of BCEL.

For the purpose of dataflow analyses, BCEL does not come with any built-in abstractions easing the process. That means one would have to create the necessary abstractions, like an intraprocedural control flow analysis to create the CFGs, and implement a visitor pattern for traversal.

BCEL is distributed under the Apache Software License[1] (open source) and can be downloaded from [3].

2.3.3 ASM

ASM[2, 9] is a bytecode engineering library suited for static and dynamic optimizations and transformations of Java programs, operating on bytecode level. The static analysis capabilities also suit it for static analysis of Java bytecode. The framework is highly optimized and is rather small and fast, e.g., compared to BCEL, while offering similar functionalities. ASM analyses compiled class files directly, which means arrays of bytes as classes are stored on disk and loaded in the Java Virtual Machine. ASM is able to read, write, transform, and analyze compiled classes and does so by using the visitor design pattern. In many ways ASM resembles BCEL, but focuses more on compact size and speed, which is a core requirement for performing runtime code transformations.

ASM comes with a number of basic built-in analyses, though fewer than Soot. For the purpose of dataflow and control flow analyses it provides classes and interfaces that can be implemented and extended to the desired behavior; a clear advantage over BCEL, were one would have to implement the visitor pattern and the flow analysis. ASM also comes with an Eclipse plugin that renders the bytecode generated from your Java source files automatically while editing in Eclipse.

ASM is very well documented, via the API available at [2] and also trough the thorough guide [9], which also explains the structure and workings of ASM under the hood. Furthermore, ASM visits annotations [11] in the compiled classes and makes these metadata available for the analyses, which is either a feature left undocumented or (most likely) not present in BCEL.

To built up dataflow analyses with ASM, parts of the visitor patterns have to be customized by the developer. First of all, ASM is primarily intended for bytecode transformations, and it does not include abstractions for the flow of data - it simply applies transformations or basic analyses independent of the program state. That means it does not even have a datastructure representation of a CFG, which would have to be implemented by the developer. Though, ASM does support this to be implemented in a rather easy approach, as the basic type for dataflow analyses `Analysis` is basically an intraprocedural control flow analysis. During the analysis, it calls the methods `newControlFlowEdge` and `newControlFlowExceptionEdge`, which however are left empty by default. To build up a CFG, one would extend the `Analysis` class and override these methods, and a CFG could be constructed in whatever datastructure desired.

Comparing this to what we have seen Soot offers, leaves ASM lacking behind. The next framework presented, FindBugs, has overcome this obstacle and implements these higher level representations, but founded on both BCEL and ASM.

ASM is distributed under an open source license, specific for the tool, which can be reviewed in [2].

2.3.4 FindBugs

FindBugs [4, 10, 13, 5, 6, 16, 22] is the last framework we have considered. FindBugs is a tool that searches for bug patterns in Java bytecode, resembling ASM a lot in the way it operates. As a matter of fact FindBugs uses both BCEL and ASM as foundation for its analyses. FindBugs uses the visitor design pattern in the same way ASM does, and the detectors are basically state machines, driven by the visited instructions, that recognizes particular bugs.

The framework comes with many analyses built-in and classes and interfaces that can be extended to build custom dataflow analyses, amongst others. Apart from that, the framework contains a suite of detectors, that use the analyses to implement the before mentioned state machines that make up bug detectors. The framework operates on bytecode and comes with an intraprocedural control flow analysis that transforms the analyzed bytecode into CFGs.

FindBugs has very good documentation, especially the API documentation stands out. Although, it is not as well documented as ASM concerning the details of its basic workings. As it uses the datatypes of both ASM and BCEL, the APIs of these

tools have to be used in addition. Lots of recent projects have been using FindBugs and guides of usage are easy to find.

Findbugs also comes with an Eclipse plugin, that based on the analyses chosen from FindBugs notifies the user with bug descriptions on program locations where a bug was detected. FindBugs does, consequently by using the ASM framework, support metadata like annotations, so our intentions to use annotations for our analyses, can be fulfilled with FindBugs as a framework.

For implementing of custom dataflow analyses the developer should take the following steps:

1. Extend the interface `DataflowAnalysis` or any of its subclasses. This class is responsible for all the flow functions and the block order, i.e. forward or backward.
2. Create a class representing the fact passed through the flow functions and updated appropriately to represent the information that may be desired at the specific program locations. This class does not have to conform to any parent type, which offers the developer great freedom of what is desired to represent at individual program locations.

After specifying these classes, other analyses or detectors can be developed that instantiate and run the particular dataflow analysis, which can then be queried about the analysis results at specific program locations. These abstractions are in accordance with theoretical dataflow abstractions, and allows for easy implementation of custom operations for combining analysis information from different control flows.

FindBugs is distributed under the GNU Lesser General Public License [7] and can be downloaded from [4].

2.3.5 Summary

In this project we develop different dataflow analyses, which we will ultimately implement in Java. Amongst the frameworks here presented, Soot and FindBugs stand out as the most feature-rich, meaning that they come with built-in dataflow analyses and have good possibilities for extending classes to the desired behavior of custom dataflow analyses. While ASM also offers some of these options, FindBugs is already using ASM in its core, and is superior as it has dataflow analysis abstractions built-in. BCEL is more or less ruled out, except for the fact that it is also the foundation for FindBugs.

Our choice of framework has been very influenced by the documentation and our ability to familiarize with the framework. Here Soot really fell behind, as it seems very poorly documented and help and examples were not easy to find. The framework

seems very complex in its structure, although it does seem to come with lots of useful built-in analyses. Another disadvantage of Soot is that it introduces a new language, Jimple, on which the analyses are run, and the developer will have to get familiarized with this language. On the other hand, in FindBugs the developer will have to get familiarized with bytecode, which has more than 200 instructions, compared to Jimples approximately 15 instructions. However, we think that we could benefit more from introducing ourselves to bytecode, than to learn a language only specific to Soot and with no further applications. FindBugs also has a greater flexibility in the choice of fact representation in a custom dataflow analysis, than Soot, which dictates manners of the facts as it must derive from a certain fact interface.

All in all we have set our decision on the FindBugs analysis framework, which we shall utilize to built our analyses upon.

2.4 The Java Execution Model

Because many aspects of concurrent programming are closely related to the way that the Java Virtual Machine (JVM) executes code and interacts with the native platform, a good understanding of the execution model is necessary in order to perform a correct analysis.

The JVM is a stack based virtual machine that is one of the cornerstones in the Java platform. The JVM provides the developer with an instruction set common on all platform which makes the "code once, run anywhere" philosophy possible. The JVM in it self does not know anything about the Java language because all it needs to do, is to provide an architecture capable of executing programs that can be expressed within the Java language. This means that the JVM can be used as a platform for many other languages as long as the semantics of a program in the given language can be expressed in bytecode.

In Java code is executed inside threads, where each thread has its own execution stack which is composed of *frames*. A frame represents a method invocation and every time a method is invoked, a new frame is pushed onto the stack. When a thread exits a method, either by returning or as a consequence of an unhandled exception, the frame on top of the stack is popped, revealing the frame belonging to the calling method where program execution should continue.

Each frame consists of two parts: a local variable part and an operand stack part. Local variables within a frame can be accessed in any order, whereas the operand stack, as the name implies, is a stack of values that are used as operands by bytecode instructions. This means that values in the stack can only be accessed in a LIFO[1] order. One should not confuse the operand stack and the threads execution stack: Each frame in the execution stack has its own operand stack.

[1] LIFO is the abbreviation of "last in first out".

The size of the local variables and the operand stack depends on the method that the given frame belongs to. These sizes are computed at compile time, and are stored along with the bytecode instructions in the compiled classes. As a consequence, at run time, all frames belonging to a given method will have a fixed size.
When a frame is created, it is initialized with an empty stack, and its local variables are initialized with the target object `this` (for non-static methods) and the method's arguments. The operand stack and the local variables can hold any Java value, except `long` and `double`, these values are 64 bit and therefore require two 32 bit slots. This will in many cases complicate the management of local variables because one cannot be sure that the $i'th$ argument is stored in the $i'th$ locale variable.

As stated earlier the JVM executes bytecode instructions. Each instruction is made of an opcode that identifies the instruction and a fixed number of arguments.

- The *opcode* is an unsigned byte value which limits the instruction set of JVM to a maximum of 256 different instructions. At the time of writing not all opcodes are used, meaning that there is room for adding new instructions to the JVM. Valid opcodes can be identified by a mnemonic symbol making the instruction easier to remember. For example the opcode `0xC2` is identified by the mnemonic symbol `MONITORENTER`.

- The *arguments* are static values that define the precise behavior of the instruction. Instruction arguments are given just after the opcode and should not be confused with instruction operands: argument values are statically known at compile time and are stored in the compiled code, whereas the operand values come from the operand stack and are therefore first known at runtime.

Instructions can be divided into two categories: A small set of instructions which are used for transferring values between the local variables and the operand stack. The other instructions only act on the operand stack as they pop some values from the stack, compute a result based on these values, and push the result back on to the stack.

The bytecode instructions `ILOAD`, `LLOAD`, `FLOAD`, `DLOAD` and `ALOAD` are used to read a local variable and push its value on the operand stack. All these instructions take an index i as an argument which is the local variable index. The `ILOAD` instruction is used to load a `boolean`, `byte`, `char`, `short` or `int` local variable. The `LLOAD`, `FLOAD` and `DLOAD` instructions are used to load a `long`, `float` and `double`, respectively, where the `LLOAD` and `DLOAD` loads the value at index i and $i + 1$, as they consume 64 bit. Finally the `ALOAD` instruction is used for loading a non-primitive value, namely object and array references.
For each of these `LOAD` instructions there exists a matched `STORE` instruction used to pop a value from the operand stack and store it in a local variable designated by its index i.

The `LOAD` and `STORE` instructions are typed to ensure that no illegal conversion is done. An `ISTORE 1` followed by a `ALOAD 1` is illegal because the stored value is loaded using a different type. If such conversion was allowed, e.g., which is is in C, it would be possible to store an arbitrary memory address in a local variable, and then turn it into an object reference, which makes encapsulation impossible. It is however perfectly legal to overwrite a local value with a given type with a value of another type. Note that this means that the type of a local variable may change at runtime.

The other instructions than the ones described above, work on the operand stack only. Below we have categorized these remaining instructions:

- **Stack**
  These instructions are used to manipulate the values on the stack. The `POP` instruction pops the value on top of the stack. The `DUP` instruction duplicates the top value on the stack by pushing the top value on to the stack. Finally, the `SWAP` instruction pop the two upper values and push then back on the operand stack in reverse order.

- **Constants**
  The constant instructions are used to push a constant value on the operand stack. `ACONST_NULL` pushes the `null` value, `ICONST_0` pushes the `int` value 0, `FCONST_0` pushes the `float` value 0 and `DCONST_0` pushes the `double` value 0. The `BIPUSH b` pushes the `byte` with value `b`, `SIPUSH s` pushes the `short` value `s` and `LDC c` pushes an arbitrary `int`, `float`, `long`, `double`, `String` or `class` constant `c` on the operand stack.

- **Arithmetic and logic**
  These instructions are used to pop numeric values from the operand stack, combine them, and push a result back on to the stack. None of the instructions take any arguments but work purely on the operand stack. The instructions: `xADD`, `xSUB`, `xMUL`, `xDIV` and `xREM` correspond to `+`, `-`, `*`, `/` and `%`, where `x` is either `I`, `L`, `F` or `D`. Furthermore there exist instructions corresponding to `<<`, `>>`, `>>>`, `|`, `&` and `^`, for `int` and `long` values.

- **Casts**
  These instructions are used to cast a value with a given type to another type, which is done by popping a value from the stack, converting the type, and pushing the result back on the stack. There exists instructions corresponding to the cast expressions found in Java. `I2F`, `F2D`, `L2D`, etc. convert numeric values from one numeric type to another. The `CHECKCAST t` instruction converts a reference value to the type `t`.

- **Objects**
  This category deals with the creation of objects, locking them, testing their type, etc.

The NEW type instruction is used to push a new object with the given type on the operand stack. The MONITORENTER objectref and MONITOREXIT objectref instructions both pop an object from the operand stack, and respectively requests and releases the lock on the object. Note that if the objectref is null a NullPointerException will be thrown.

- **Fields**
  Field instructions are used to read or write the value of a field.
  GETFIELD owner name desc pops an object reference from the operand stack, and pushes the value of its name field. PUTFIELD owner name desc pops a value and an object reference, and stores the value in its name field. In both cases the object must be type owner and its field must be type desc. The GETSTATIC and PUTSTATIC are instructions that work in a similar way but for static fields.

- **Methods**
  The instructions INVOKEINTERFACE, INVOKESPECIAL, INVOKESTATIC, INVOKEVIRTUAL are used for invoking a method or a constructor. Common for all these instructions are that they pop as many values as there are method arguments, plus the value for the target object, and they push the result of the method invocation on to the operand stack.

- **Arrays**
  These instructions are used to read and write values in arrays. The xALOAD instruction pop an index and an array, and pushes the value of the array element at this index on the operand stack. The xASTORE instruction pop a value, an index and an array, and store the value at that index in the array. For both instructions x can be I, L, F, D, A, B, C or S.

- **Jumps**
  Jump instructions are used to jump to a arbitrary instructions if some condition is true, or simply unconditionally. The Java primitives: if, for, do, while , break and continue are represented in bytecode using jump instructions. For example the instruction IFEG label pops an int value from the operand stack, and jumps to the instruction with the given label and the popped value is 0, otherwise execution continues normally to the next instruction. There exist many variations of jump instructions, but their mnemonic symbols makes it easy to reason about the behavior, like IFNE and IFGE. The switch primitive in Java is however represented in bytecode using the TABLESWITCH and LOOKUPSWITCH instructions.

- **Return**
  Finally the xRETURN and RETURN instructions are used to terminate execution within a given method and to return its result to the caller. The RETURN instruction are used in case where a method return void, and xRETURN are used in all other cases, where x can be A, D, F, I or L.

Not all the instructions that the JVM support have been described in the above items, but the reader should by now have gained enough knowledge about the JVM instruction set to understand the following chapters. For more information consult the JVM specification [21].

2.5 Java and Synchronization Primitives

In a multi-threaded program, several threads execute simultaneously in a shared address space, which means that state variables may be shared between threads. In order to classify a piece of code as being thread-safe, it must function correctly during simultaneous execution by multiple threads. This means that no matter how the threads interleave, the semantics of the program should be deterministic from the developers' point of view, and therefore concurrent programs must be correctly synchronized to avoid counterintuitive behaviors.

The target of our analyses will be the Java programming language. As already mentioned, we shall not identify threads in our analyses. Instead we will identify synchronization primitives in the class being analyzed and compute the necessary information for our analyses based on the synchronization primitives. Java provides multiple synchronization primitives, but in this project we will mainly focus on the following:

2.5.1 Monitors

In the early seventies Edsger Dijkstra, Per Brinch-Hansen, and C.A.R. Hoare developed the idea of a monitor, an object that would protect data by enforcing single threaded access, meaning that only one thread would be allowed to execute code within the monitor at one time.

Java provides the `synchronized` primitive which reassembles the monitor idea in many ways[2], and we will in the following denote the `synchronized` primitive as a monitor. In the classical monitor approach, the monitor object contains a lock which is taken by the given thread that enters the monitor. If a thread tries to enter the monitor and the lock is already taken the thread is simply blocked. The semantics of the monitor pattern ensures that the given thread exits the monitor before returning the lock to the monitor object, so that no to threads can never be within the monitor object at once. Java monitors are at bit more flexible, as they allow to use any object as a monitor lock, and like the classical monitor approach only one thread at a time can execute code within the monitor.

[2] In bytecode the instructions used when entering/exiting a synchronized region is even called "monitorenter" and "monitorexit".

A thread in the classical monitor approach can exit a monitor either by returning from the given method or by waiting on a condition queue. Waiting threads can then be woken up, by another thread notifying the condition queue. In early style monitor implementations, notifying a condition queue caused a waiting thread to receive the lock and run immediately. Because implementations that guarantee such semantics suffer from a high overhead, newer implementations first hand over the lock to a waiting thread, when the notifying thread exits the monitor. Java supports the later kind of semantics with a few twists, through the use of the `wait`, `notify` and `notifyAll` primitives. One interesting thing introduced in Java 1.5, is the ability for threads to wake up without being notified, interrupted or timed out, a so-called spurious wakeup. Therefore one should always test if a thread that reenter the monitor, is allowed to proceed execution, otherwise it should continue waiting. Listing 2.1 illustrates how one can accommodate spurious wakeups.

```
1 synchronized (obj) {
2   while (<condition does not hold>) obj.wait(timeout);
3   ... // Perform action appropriate to condition.
4 }
```

Listing 2.1: Illustrates how one can accommodate spurious wakeups, by surrounding the `wait` method call.

In Java a thread may exit the scope of a monitor in the same ways as a thread may exit the classical monitor, furthermore an exception can be thrown within the scope of the monitor, which may or may not make the thread exit the monitor, depending on whether the exception is caught within the monitor. The semantics will in all cases ensure that a given thread cannot leave the scope of a monitor without releasing the lock.

Finally one should note that monitors are reentrant, meaning that a thread holding a given lock belonging to a given monitor, can enter the same monitor over and over again, this kind of semantics therefore allow recursive calls to be performed while holding a specific lock. If monitors were not reentrant the example in Listing 2.2 would deadlock, assuming the boolean expression evaluates `true`.

```
1 synchronized method() {
2   // Determine if recursion should continue
3   if (<condition does hold>) method();
4 }
```

Listing 2.2: Illustrates the advantage of reentrant monitors. If Java monitors were not reentrant, a recursive method call like the one above, would deadlock, assuming that the boolean expression evaluates `true`.

2.5.2 Locks

Another mechanism that Java provides for communicating between threads is the so called `Lock`. The concept of the `Lock` was introduced in Java 1.5 as a part of the `java.util.concurrent` package, which provides the developer with a nice toolbox for constructing multi-threaded programs. Probably, the most interesting thing about the `lock` is that it offers far greater flexibility than the monitor in terms of design of a critical region. This is due to the fact that the `synchronized` primitive is a part of the Java grammar which enforces the developer to define the scope of the monitor, whereas the `lock` is implemented as an object and therefore does not suffer from these grammar constraints. This enables the developer to construct locked regions that intersect, whereas the `synchronized` primitive only allows locked regions to be contained in each other. Therefore a mutual exclusive region like the one in Listing 2.3 can never be constructed using the `synchronized` primitive.

```
1 method() {
2   A.lock();
3   B.lock();
4   A.unlock();
5   B.unlock();
6 }
```

Listing 2.3: Illustrates that locks may be used to construct locked regions that intersect, and not only contained in each other.

In many applications, an object is shared between threads where some of them only read the state of the object, whereas other threads alter the state of the object[3]. By using the monitor approach, the developer has no other choice than making exclusive access to the state variables encapsulated within the object in order to avoid race conditions. Because a race condition first occur when a thread reads a variable changed by another thread, no race condition occur if multiple threads only read the object state simultaneously. Therefore the monitor approach has quite an overhead because mutual exclusion is enforced on both reads and writes. In order to increase performance in such cases, Java provides the `ReentrantReadWriteLock`[4] implementation which is a synchronization primitive that can be used to solve the readers-writer problem. The `ReentrantReadWriteLock` combines two locks, namely a `ReentrantReadLock` and a `ReentrantWriteLock` used for respectively reading from and writing to a shared state. The semantics of the locks allow multiple reader threads to read from the shared state concurrently, while a writer thread requires exclusive access.

Java provides other `Lock` implementations, but common for all of them are that they extend from the `java.util.concurrent.locks.Lock`.

[3]This is known as the classical readers-writer problem.
[4]Note that the lock is reentrant.

Finally one should always accommodate exception handling when applying locks, because the semantics of locks does not require a one-to-one relation between the number of calls to `lock` and `unlock`, this is often done by using the `try` - `finally` semantics to ensure that `unlock` is always called when a thread leaves the intended scope of the lock. Listing 2.4 illustrates how `try` - `finally` may be used to guarantee that the lock is released, even in the case of an exception being thrown.

```
1 method() {
2   l.lock();
3   try {
4     // Perform actions
5   }
6   finally {
7     l.unlock();
8   }
9 }
```

Listing 2.4: Illustrates how `try` - `finally` can be used to guaretee that a lock is released. Note that if the `lock` method call was inside the `try`-scope, there would exist the potential risk of throwing an exception before acquiring the lock, thereby calling `unlock` without owing the lock.

Both monitors and locks guarantee “visibility”, a property that is closely related to the Java memory model which will be described in section 2.4.

2.5.3 Semaphores

Edsger Dijkstra invented the semaphore, a classic concurrency control construct. The classical semaphore only has two methods, namely `V()` and `P()`, where `V` stands for “verhoog”, or “increase’ and `P` for “probeer te verlagen”, or “try-and-decrease”. Note that Edsger Dijkstra was Dutch.
Listing 2.5 illustrates an implementation of a semaphore, where both the `V()` and `P()` methods are synchronized because parts of the operations within these methods must be done atomic.

Because semaphores are very simple they are often used in environments where resources are few, e.g., they are the primitive synchronization mechanism in many operating systems.

2.5.4 Some Notions of Locks

For the benefit of later topics, we denote the following notions about the different synchronization mechanisms:

```
1  public class Semaphore {
2    private volatile int s = 0;
3    public Semaphore(int s0) {
4      s = s0;
5    }
6    public synchronized void P() throws InterruptedException {
7      while(s == 0) wait(); /* must be atomic once s > 0 is detected */
8      s--;
9    }
10   public synchronized void V() {
11     s++; /* must be atomic */
12     notify();
13   }
14 }
```

Listing 2.5: An implementation of a simple semaphore. Note that the implementation depends on the `synchronized` primitive, thus the semaphore would not work without.

- **Reader- and writer-locks**
  We refer to types of `ReentrantReadLock` (or types extended from) as *reader-locks* and similar to types of `ReentrantWriteLock` (or types extended from) as *writer-locks*.
- **Mutual exclusive locks**
  We will use this phrase about all monitors and locks, that are not reader- or writer-locks (or derived from any of these). The reason is, that synchronization performed by these mechanisms are performed with mutual exclusion between the blocks of code surrounded by a synchronization primitive of these kinds, allowing us to generalize their common behavior in one term.
- M denotes a mutual exclusive lock.
- R denotes a reader-lock.
- W denotes a writer-lock.

The latter three notions may be followed by indices allowing to distinguish between locks. E.g. the reader- and writer-locks from the same `ReentrantReadWriteLock` will be addressed R_i and W_i, respectively.

Finally, let $ls(\ell_i)$ denote the set of locks held *at* the location ℓ_i and $ls(\ell_i)'$ the set of locks held *after* ℓ_i.

2.6 Class-wise Thread Safety

To be able to define analyses capable of detecting non thread-safe behavior in a class, we must initially give some meaning to what thread-safe behavior is. In Java a class

consists of methods and fields, where the fields are used to maintain the state of the object. A class without field variables have no state because the outcome of a method call can only depend on the input, namely the arguments.

In this section, we look upon some common rules and policies for writing thread-safe programs and generalize this to a set of rules that we can apply to our analyses. We start by summarizing some usefull policies for using and sharing objects in a concurrent program. These concepts are adopted from [12].

- **Thread-confined**
  A thread-confined object is owned exclusively by and confined to one thread, and can be modified by its owning thread.
  This policy is known from the Java Swing framework, where all access to GUI components must be done through the event-dispatch thread.

- **Shared read-only**
  A shared read-only object can be accessed concurrently by multiple threads without additional synchronization, but cannot be modified by any thread. Shared read-only objects include immutable and effectively immutable objects.

- **Shared thread-safe**
  A thread-safe object performs synchronization internally, so multiple threads can freely access it through its public interface without further synchronization.

- **Guarded**
  A guarded object can be accessed only with a specific lock held. Guarded objects include those that are encapsulated within other thread-safe objects and published objects that are known to be guarded by a specific lock.
  A policy like this thereby assumes that the client conforms to the given lock-protocol.

Because the "Thread-confined" and "Guarded" policies are program-wise, a complete analysis of a program would be required in order determine if a program comply to one of these policies.

Regarding the "Shared read-only" and "Shared thread-safe" policies a class-wise analysis is possible because only the class invariants must be maintained. When an object enters a state where one or more class invariants may be violated thread-safety might fail. In order to analyze a program for thread-safety without knowing the invariants the developer had in mind, one has no other choice than making an over-approximation when reasoning about class invariants. Some analysis tools enable the developer to express invariants in a formal way within the code, so that the analysis can benefit from knowing the exact invariants, assuming that the developer is able to express them correctly. Even though an analysis enabling invariants to be described by the developer has the advantage of being potentially more precise, it also has the

drawback of being dependent on the developer to write correct invariants, which may not be a trivial task.

As stated in the introduction we have decided to use the concept of the SPA when trying to reason about thread-safety, because this reminds us that we should always over-approximate our analysis in order to be sure that the SPA cannot break any invariants of the class.

When saying that a class is threadsafe we mean that neither safety- or liveness-properties can be violated within the context of the class. Below we have defined the set of rules that make up our definition of class-wise thread-safety.

2.6.1 Encapsulation

The state of an object is represented by the field variables in the object. For the class-wise approach we apply, that means field variables must not be directly accessible from outside the class or in classes extending the class of interest, because then the SPA may be able to break class invariants by altering the state of the class of interest. This leads to a general property of thread-safety for field variables, namely that field values may not be declared `public` or `protected`, unless they are declared `final`.

Field variables of a class may be objects that have an internal state themselves. If the state of such objects is not properly encapsulated and the fields are accessible by the SPA, their state may be altered and thereby also the state of the class of interest is altered, potentially breaking class invariants. A property that ensures that the state of the class of interest is not altered by SPA is to require all reference-type field variables are declared `private`.

However, a less conservative property can be applied. We say that an object with a state that can be altered from outside the object is *mutable* and otherwise it is called *immutable*. If the field variables of the class of interest contains mutable objects, then these must be declared `private`. If the field variables contain immutable objects, then it is only required that they are not declared `public` or `protected`, unless they also are declared `final`. Then the SPA may not unexpectedly change the state of the class of interest.

Listing 2.6 shows examples of bad encapsulation.

```
1 public class BadEncapsulationExample {
2   public String a = "";                   /*BAD: Public field*/
3   protected String b = "";                /*BAD: Protected field*/
4   public final List l = new ArrayList(); /*BAD: Mutable object in a public field*/
5   /* etc. */
6 }
```

Listing 2.6: Example of encapsulation violations

A final note regarding field variables, is that if a field is used as a lock by a Java synchronization primitive within the class of interest, then that field must be `private` , because otherwise it may be accessed by the SPA, raising the possibility of deadlock within the class of interest (if SPA provokes, e.g., a deadlock or infinite loop outside the class). This will further be described in the next section.

2.6.2 Absence of Deadlock

If a program enters a state where at least two threads may be waiting for the locks held by the other and vice versa, we classify this as being a potential deadlock. In reality this may not be a deadlock because a third thread may release some resource, so that one of the two stuck threads are able to continue. As described earlier we will have to settle for an over-approximation, and therefore we will classify a potential deadlock as deadlock in the context of the analysis.

We will in the following take a closer look at the properties of deadlock and derive a more formal definition of how potential deadlocks can be revealed. The Listing

```
1  public class DeadlockExample {
2    private final Object m1 = new Object();
3    private final Object m2 = new Object();
4
5    public void doStuff1() {
6      synchronized(m1) {
7        /* Thread T1 here => deadlock with T2 */
8        synchronized(m2) {
9          /* etc. */
10       }
11     }
12   }
13
14   public void doStuff2() {
15     synchronized(m2) {
16       /* Thread T2 here => deadlock with T1 */
17       synchronized(m1) {
18         /* etc. */
19       }
20     }
21   }
22 }
```

Listing 2.7: Examples of deadlocks between mutual exclusive locks

2.7 shows some examples of potential deadlock situations, when using mutually exclusive locking mechanisms. The characteristics can easily be described formally: Consider two different locations, l_1 and l_2. Then a potential deadlock exists in case the following condition holds:

$$ls(l_1) \cap ls(l_2) = \emptyset \wedge ls(l_1)' \cap ls(l_2) \neq \emptyset \wedge ls(l_2)' \cap ls(l_1) \neq \emptyset \qquad (2.1)$$

For the case of reader- and writer-locks, deadlocks can occur in another manner: If a location locked by the reader-lock, R_i, attempts to acquire the corresponding writer-lock, W_i, then a deadlock will occur if that particular location is reached. Similarly, a deadlock will occur if a location locked by W_i attempts to acquire R_i (or W_i again) and that location is ever reached. We may formalize this property also. We will refer to the readlock of lock L as $rl(L)$ and the writelock of lock L as $wl(L)$, allowing us to formalize the desired property:

$$\begin{aligned}
&(rl(L), > 0) \in ls(\ell) \wedge (wl(L), 1) \in ls(\ell)' && \vee \\
&(wl(L), 1) \in ls(\ell) \wedge (rl(L), > 0) \in ls(\ell)' && \vee \\
&(wl(L), > 1) \in ls(\ell)
\end{aligned} \tag{2.2}$$

where (l, n) denotes the lock l acquired n times.

Another case of deadlocking regarding writer-locks, is due to the fact that writer-locks behave as mutually exclusive locks do, thus 2.1 also applies if one or more writer-locks are present in the sets of locks causing potential deadlocks.

Another constraint is that all objects used as a lock mechanism, must be declared `private`. The reason is that SPA may introduce a deadlock using visible immutable state objects. Thus an interesting consequence is that the `this` reference should never be used as a synchronization primitive, because the `this` reference can never be encapsulated within the object itself. Listing 2.8 exemplifies this particular case.

2.6.3 Escaped Objects

All objects that enter through non-private methods have the potential to be shared between threads, therefore all objects that enter the class being analyzed are escaped. This means that field variables can never be substituted with objects that are passed as parameters to a non-private method, because the class must be able to synchronize all access to its own state. This problem can for example be related to the `Collections.SynchronizedList`, which is an object that encapsulates a regular non-thread-safe `List`-type while synchronizing all access to it and thereby making it thread-safe. The problem is that the `List` has to be given as a parameter, making it impossible for the wrapper class to be sure that the encapsulated `List` objects are not accessed directly which would break thread-safety.

It is allowed for an escaped value to be passed as an argument to a method call on a state variable, otherwise the state of the object could never be modified as a result of a method call. A consequence of this is that return values from method calls on a state variable are escaped, if an escaped object can be used as a parameter in any method invocation on the given state variable. If no escaped object can escape to a state variable, the state of that state variable cannot be escaped, because the class must encapsulate all muteable state variables.

```java
public class ThisDeadlockExample {
  public void doStuff1() {
    /* Threads T1 and T2 executed by SPA prevents entering forever. */
    synchronized (this) {
      /* Do important stuff ... */
    }
  }
}

public class SPACausingDeadlock {
  private final ThisDeadlockExample e = new ThisDeadlockExample();

  public void doStuff3() {
    synchronized (e) {
      /* Thread T1 here => deadlock with T2 */
      synchronized (this) {
        /* etc. */
      }
    }
  }

  public void doStuff3() {
    synchronized (this) {
      /* Thread T2 here => deadlock with T1 */
      synchronized (e) {
        /* etc. */
      }
    }
  }
}
```

Listing 2.8: An example illustrating how SPA can cause a deadlock when `this` is used for locking.

This also implies that a state variable is escaped if it is passed as an argument to a method call on an escaped object. Furthermore all values that are returned from within the context of the class being analyzed will be categorized as escaped.

A final constraint for escaped objects, is that if they are used as a lock by a synchronization mechanism, that particular lock may be changed outside the class of interest. Thus the regions of code the lock surrounds will be not to be mutually exclusive or sound with respect to reader- and writer-locks, depending of which kind of lock it is. Therefore locking with escaped objects can be ignored.

2.6.4 Locking

To maintain the class of interest in a state without violations of liveness properties, all locks acquired within a non-private method in the class of interest, must be released again before all exit locations of that method and similar more locks may not be released than acquired, within non-private methods. Otherwise the SPA may leave the class of interest in a state where liveness properties are not satisfied.

At the point where a lock is acquired, the object that is locked on should usually not point to several values. In case a lock potentially points to several values, the region within the locked scope cannot be assumed to be mutual exclusive with any other locked regions within the class. We therefore classify this as unwanted locking behavior.

2.6.5 Thread-Safe Field Access

All access to non thread-safe fields must be mutual exclusive, meaning that only one thread at a time may use the given object. Though, with the exception of synchronization with the means of reader- and writer-locks, where only interleavings that allow both reader- and writer-locks or multiple writer-locks to access a non thread-safe field simultaneously, will violate thread-safety. If these constraints are not fulfilled, the object being used concurrently might enter an unsafe state where its class invariants are not satisfied. Regarding escaped objects synchronization is of no use, because we can never be sure that we are in complete control of the escaped object. The synchronization primitives used to construct mutual exclusive regions within a class, must be stored in field variables in order to be visible in every method. If one for some reason wants to substitute an object used as a synchronization primitive with another, all access to that field must be synchronized in order to maintain thread-safety in the regions that the object is used to protect. In listing 2.9 we see that a `ReentrantReadWriteLock` can be used to safely change the variable `o`, which is used as a synchronization primitive. The problem is a typical readers-writer problem. Listing 2.9 illustrates that multiple readers will perform mutual exclusive actions,

```
public class NonFinalLockPrimitive {
  private ReentrantReadWriteLock rwl =
      new ReentrantReadWriteLock();
  private Object o = new Object();
  public void doMutexAction() {
    rwl.readLock().lock();
    try {
      synchronized(o) {
        ... // do mutex action
      }
    }
    finally {
      rwl.readLock().unlock();
    }
  }
  public void newLockPrimitive() {
    rwl.writeLock().lock();
    try {
      o = new Object();
    }
    finally {
      rwl.writeLock().unlock();
    }
  }
}
```

Listing 2.9: Example of the use of `ReentrantReadWriteLock`, where a synchronization primitive is safely changed.

and as the lock `o` changes, it is performed under a write-lock, such that no readers will be performing the mutual exclusive action.

In section 3.1 we will see that the ability to change references to objects that is used as a synchronization primitive, introduces some challenges in the design of a desired analysis, capable of rendering visible the values a given variable is possible assigned, when concurrent access is taken into account.

In general, ensuring safe field accesses regarding the potential misuse caused by SPA, must rely on that the developer intends to be able to access state variables with deterministic values. If that is somehow not the case, we are not capable of properly approximating the class invariant the developer may have in mind. Thereof we come to a general conclusion about safe access to state variables and the general approximation of our analysis will be:

> *All writes to a field variable may not give occasion to non-deterministic results possibly being read elsewhere in the class. Neither must multiple writes to the same field variable take place at any time.*

2.6.6 Stale data

Modern computers utilize multiple processors and caches to speedup program execution. In a shared-memory multiprocessor architecture, each processor has its own cache that is periodically reconciled with the shared main memory.
Ensuring that every processor knows exactly what the other processors in the system is doing at all time is to expensive, because most of the time this information is not needed. Therefore processors relax their memory-coherency guarantees to improve performance. An architectures memory-model tells what guarantees can expected from the memory-system, and specifies a set of instructions necessary to coordinate access to shared data. Because not all architectures provide the same set of guarantees concerning the memory-model, Java provides its own memory model, where the JVM deals with the differences between the common JMM and the underlaying architectures memory-model, inserting the necessary memory barriers.
The JMM defines a partial ordering called *happens-before* on all actions within a program. To guarantee that a thread executing an action B, can see the results of an action A(whether or not A and B occur in different threads), there must exists a *happen-before* relationship between A and B. In order for the JVM to optimize performance, e.g., by trying to reduce the number of cache misses, the JVM is is free to reorder instructions where no *happens-before* relationship exists. This makes the reasoning about ordering in the absence of synchronization complicated. The rules for *happen-before* are:

- **Program order rule**
  Each action within a thread *happens-before* every action in that thread that comes later in the program order.

- **Lock rule**
  An unlock on a monitor lock *happens-before* every subsequent lock on that same monitor lock. Furthermore does locks and unlocks on subclasses of `java.util.concurrent.Lock` have the same memory semantics as monitor locks.

- **Volatile variable rule**
  A write to a volatile field *happens-before* every subsequent read of that same field. Also reads and writes of atomic variables have the same memory semantics as volatile variables.

- **Thread start rule**
  A call to `Thread.start` on a thread *happens-before* every action in the started thread.

- **Thread termination rule**
  Any action in a thread *happens-before* any other thread detects that thread has terminated, either by successfully return from `Thread.join` or by `Thread.isAlive` returning `false`.

- **Interruption rule**
  A thread calling interrupt on another thread *happens-before* the interrupted thread detects the interrupt, with is either done by having a `InterruptedException` thrown, or by invoking `isInterrupted` or `interrupted`.

- **Finalizer rule**
  The end of a constructor for an object *happens-before* the start of the finalizer for the object.

- **Transitivity**
  If A *happens-before* B, and B *happens-before* C, then A *happens-before* C.

Listing 2.10 illustrates the concept of stale data, where a value `n` is given as an argument in the constructor and assigned to a field. Because no *happens-before* relationship exists, the value of `n` may not be visible to other threads after the constructor has finished. The guarantee of *initialization safety* ensures that for properly constructed

```
1  public class Holder {
2    private int n;
3
4    public Holder(int n) { this.n = n; }
5
6    public void assertSanity() {
7      if(n != n) {
8        throw new AssertionError("Expression is true!");
9      }
10   }
11 }
```

Listing 2.10: A class where no *happens-before* relationship exists between the assignment to `this.n` in the constructor and the `if`-branch, thus `n` is not properly published and therefore may the boolean expression in line 7 be `true`.

objects, all threads will see correct values of `final` fields, regardless of how the object is published. The `Holder` class may be fixed by declaring `n` `final` or by establishing a *happens-before* relationship, e.g., by declaring `n` `volatile`.

Chapter 3

The Analyses

Java support for concurrency is like the first rule: Don't spill grape juice on the carpet.

– Unknown

In this chapter, we present the analyses that we have developed and implemented to facilitate a tool capable of detecting certain synchronization pitfalls. First of all, we need to narrow down our overall approach and identify the functionality of the analyses we apply.

3.1 Our Approach

First of all, the classes our analyses operates on are binary bytecode class files compiled from Java source programs, meaning that the targets of the analyses are required to be verified bytecode classes. Thus, we do not perform any initial verification of the input classes, but assume them to be verified already.

Section 1.1 on page 1 describes the main concept of our approach. We view the target of the analysis from the point-of-view of a strongest possible attacker, SPA, to reveal where thread-safety may fail. This resembles the concept of *unit tests*, where

a unit is somehow identified as target of a test suite, to make sure this unit will perform as expected. After such a test suite has been performed, the unit can be "plugged in" as a component of a larger system, where it can be viewed as a simple black box, that given some particular input, produces the correct output. The target unit or component of our analysis will be a specific Java class, which will then be analyzed from the view of SPA, with regards to non thread-safe behavior. In Section 2.6 on page 29 this led us to a set of well defined properties that apply to class-wise thread-safety and these definitions are the properties we must analyze. A small recap follows; refer to Section 2.6 for full descriptions:

- **Encapsulation**
  Mutable state variables of a class must be kept private. Immutable state variables may be non-private if they are declared `final`, unless they are used as lock mechanism by Java's synchronization primitives in which case they may be cause for deadlocks.

- **Absence of Deadlock**
  No interleaving may exist, where two threads wait for the lock that is held by the other and vice versa. This eliminates the existence of deadlocks.

- **Escaped objects**
  Basically, all objects either not properly encapsulated or entering from outside the current target class of the analysis, are escaped. In case escaped objects enter the state of any object inside the class, all return values from this object, will also be considered escaped.

- **Locking**
  All non-private methods within the class must have released all locks held at return point.

- **Thread-safe Field Access**
  All accesses to non thread-safe fields must be mutual exclusive or sound with respect to multiple readers and writers accessing the non thread-safe field.

- **Stale Data**
  The existence of caches with write-back memory models reveal the possibility that non synchronized accesses to fields can result in inconsistent values and must be analyzed.

To be able to document how we have identified and developed analyses cabable of revealing violations in the above conditions, we will look at each of the conditions in turn and identify what information they will require. Based on that information, we will present analyses capable of collecting that information.

3.1.1 Encapsulation

To analyze whether the state of a class is properly encapsulated, we need to traverse and notice if fields are mutable or immutable. After this, we can check if they conform to the requirement stated about encapsulation, by testing against their visibility modifier and the `final` modifier. Although we cannot yet determine whether fields are used by synchronization primitives, which reveals a dependency of an analyses capable of telling which variables are used by locking mechanisms.

3.1.2 Absence of Deadlock

To identify potential deadlocks we must be able to know which variables have been used as a lock by synchronization primitives. For this to be possible, we first of all need a points-to analysis, that is, an analysis that keeps track of the possible set of values that may have been assigned to variables. With this information, another analysis can be performed, keeping track of which locks are currently held. This is easier said than done, but we shall not describe the complexity at this point.

3.1.3 Escaped objects

Analyzing which objects can have escaped, again relies on the existence of a points-to analysis, because that information is indeed required to be able to track the flow of escaped objects. The information about where escaped objects may enter the class of interest can in fact be collected by the points-to analysis and made available by somehow flagging the points-to values if they are potentially escaped objects. An individual analysis for escaped objects could then collect this information and generate violations where applicable.

3.1.4 Thread-Safe Field Access

The information required to collect this information both relies on the ability to be able to determine which locks are currently held and the ability to track the flow of possible values a variable can be assigned. As before mentioned, this does introduce some complexity which we shall soon address.

3.1.5 Stale Data

An analysis of thread-safe field access will reveal possible stale data also.

3.1.6 Summary

Until this point, we have seen the necessity of a points-to analysis and an analysis capable of identifying which locks are currently held. We will denote the latter analysis as the *lock analysis*. Inter-procedural points-to analyses are well researched and some suggestions of implementations for Java exist [25, 15, 18, 20, 29]. However, we have decided to specify an inter-procedural points-to analysis specific to our approach ourselves, because we would like the analysis to analyze and take mutually exclusive regions into account, which is a specialized behavior for our approach. We denote this particular analysis a *concurrent points-to analysis*. This analysis takes the information about locks at the current location into account and therefrom determine what other locations can be accessed concurrently and merge points-to information from these locations into the information currently present. This introduces an interdependency between the lock-analysis and the concurrent points-to analysis: The lock-analysis requires to be able to tell what values a given variable may be assigned at the location where a lock is taken on that variable, whereas the concurrent points-to analysis, requires information about which locks are held at a given location.

To address this problem, we present a general assumption that allows us to avoid this interdependence. It simple says that:

> *The variables used as locking mechanisms by synchronization primitives may not be changed anywhere in the class of interest.*

Reasoning about the limitation introduced from this assumption, we now present an example where a variable used for locking is changed.

Listing 3.1 illustrates that a variable, `m2`, used for locking, is changed, although properly synchronized by another lock, `m1`, and thereby does not ensure mutual exclusion between the scopes synchronized with `m2`: *Action 1* can very well be performed concurrently with *Action 2*. The location at *Action 2* illustrate, that if one wraps synchronizations by `m2` with synchronizations by `m1`, the only effect of the synchronizations, is that of `m1`. Therefore we might as well ignore the synchronizations by `m2`, which justifies the assumption we will apply. The example is of similar character regarding changing either a read- or write-lock.

However, we must not forget the example presented in listing 2.9. Here a lock is actually changed, while maintaining the mutual exclusive access. We have chosen to make the assumption anyways, because it allows us to avoid the interdependency mentioned. That means that examples such as that of listing 2.9, where actions are accomplished mutual exclusive, although a lock changes, may give rise to violations in our analysis. Though we believe the example stands almost for itself, and we consider the assumption a fair over-approximation of actual runtime program behavior.

Now we focus on how we use the assumption to avoid the interdependency between the lock analysis and the concurrent points-to analysis. With the assumption applied,

```java
public class BadMutualExclusion {
  private final Object m1 = new Object();
  private Object m2 = new Object();

  public void changeLock() {
    synchronized(m1) {
      m2 = new Object();
    }
  }

  public void doStuff() {
    synchronized(m2) {
      /* Action 1 */
    }
    /* etc. */
    synchronized(m1) {
      synchronized(m2) {
        /* Action 2 */
      }
    }
  }
}
```

Listing 3.1: A variable used by a synchronization primitive, is changed, resulting in non mutually exclusive actions.

this allows us to make an initial points-to analysis that is intra-procedural and thereby only reveals what variables may point in a sequential manner. As locks cannot be changed, that means that the lock analysis can very well build on top of this analysis alone, because then it is known at the time a lock is taken, what the locking variable points to. With the lock analysis in place, the concurrent points-to analysis can then be applied, as it can now be queried what a variable points to at any location and which locks are held at any location. Moreover The concurrent points-to analysis requires another analysis to come into play, namely a special *dominator analysis* that will be described in detail later.

We have now summed up the analyses we shall depend on, to be able to analyze for violations of the conditions we apply regarding class-wise thread-safety. Figure 1.1 on page 4 illustrates the order of which the analyses must be performed. What remains is detailed information about each of the analyses and the precision of the approximations they will apply. Generally, we desire the analyses to be as accurate as possible, which may be at the cost of performance.

3.2 General Definitions

We shall now describe the analyses that our tool consists of. The order of which they will be introduced, corresponds to the dependencies of the analyses, which was introduces in the previous section. Initially we will introduce some notions and functionality that will be used to describe the analyses.

In the following we will use the notion of a *location*, denoted $\boldsymbol{\ell_{C,M,n}}$, where C is the class of interest, M is the method of the CFG being analyzed and n is a linenumber ($n \in \mathbb{N}$) in the bytecode instruction sequence for M. Our analyses work on classes, and the class of interest is often implicit from the context, so we will use $\boldsymbol{\ell_{M,n}}$ or just $\boldsymbol{\ell_n}$ as the notation of a location, where we do not need to distinct between different class contexts or both the class and method is implicit from the context. In case we discuss an arbitrary location, we will not use a subscript at all, we just write $\boldsymbol{\ell}$.

For the analyses, we want to abstract an actual program element to a *value*, that uniquely identifies that program element. By program element, we could mean, e.g., a variable or a certain instantiation value, which all would be mapped in a one-to-one relationship from the value to the program element it actually describes. The program elements that an analysis operates with will be introduced in the contexts of the analyses.

Because the analyses operates on bytecode, initially we will introduce functions that can abstract the actual mapping into the stack to a representation that eases in describing and formalizing the analyses. At this point, we will assume the existence of a function $stack(\ell) = S_\ell$ which yields the contents of the stack at the location ℓ. We abstract the stack representation to an array indexable by a number i, where the bottom element has the index 0. Based on the representation of the stack as an indexable array, we now assume the existence of a function $top(S_\ell)$ that yields the top element from the stack. The top element is represented as a value which, in this context, is a one-to-one relationship between an actual variable reference, instance reference or value and the abstracted value yielded. The abstraction of this function is such that, i.e., a double on the stack (which occupies two slots, as it is 64-bit long) is represented by one value, which is the value that can be accessed by $top(S_\ell)$. In the following we shall represent values as integers. Furthermore, we introduce the function *new*, which yields a new, unique value.

The bytecode instruction operands are elements of the stack, and on the execution of instructions, initially the operands are popped off the stack and if a result is computed, the result pushed back on the stack. We shall abstract the operands of instructions by assuming the following functions to be able to index into the stack and receive the correct values, representing the operands.

$target(ins)$	The object that is the *target* of an instruction. In general this is the object into which a value is saved or an invocation is performed on.
$cpindex(ins)$	The index in the constant pool to which the instruction refers (for instructions with an index in the constant pool as operand).
$lindex(ins)$	The index in the local variable frame into where the instruction indexes (for instructions with an index in the local variable frame as operand).
$method(ins)$	Is the method of the instruction given as parameter (for instructions that has a dispatch target).

With these general definitions, we now describe the analyses one by one.

3.3 Points-To Analysis

The points-to analysis is the main cornerstone of our tool. The task of the points-to analysis is to collect information about what variables may point to at any given location. As already mentioned, this analysis is intra-procedural and the main purpose of it is to offer the lock analysis the ability to know which variable is used for locking mechanism by a synchronization primitive at a given point. Furthermore, this analysis will be used to collect information for the concurrent points-to analysis, which bases its results on the points-to and lock analyses.

We define the information that the points-to analysis must compute as a function, the points-to set denoted $PTS(\ell)$, which we will refer to as the *fact* computed by the points-to analysis, also referred to as the points-to set:

$$PTS(\ell) = \langle S, D \rangle$$

where S is a set and for each $s_n \in S$, $d_n \in D$ is a destination that s points to. The points-to set is a multiset of pairs, representing that s_i points to several destinations. The mapping in $PTS(\ell)$ expresses a directed graph, with source and intermediate nodes as values $s \in S$ and leaf nodes representing specific value or instance assignments. Note that destinations, D, may either be an intermediate node or a leaf node; that is, either a source (another variable) or a specific value or instance assignment respectively. We use the points-to set for a specific location as a function of a value, s_1, such that it yields a set, D_i, containing all destinations, d_{s_i}, s_i points to:

$$PTS(\ell)(s_i) = D_i \qquad \forall d_{s_i} \in D_i : (s_i, d_{s_i}) \in PTS(\ell)$$

In the following, we shall describe the computation of $PTS(\ell)$ for all ℓ in a given class and method. The computation is performed on the CFG for the method of interest. We now introduce some notation for computing $PTS(\ell)$ CFG-wise: Let

CFG	The CFG of interest,
(ℓ, ℓ')	represents an edge from ℓ to ℓ' in the CFG,
$\ell_\circ$	is the first location in the entry basic-block,
ι	specifies the initial analysis information,
$ins(\ell)$	yields the bytecode instruction at the location ℓ,
f_ℓ	is the transfer function at ℓ.

The transfer function, f_ℓ, depends on the information of interest at ℓ, in this case $ins(\ell)$. We now define expressions for computing the incoming and outgoing points-to information at a location ℓ, $PTS_\circ(\ell)$ and $PTS_\bullet(\ell)$ respectively.

$$\begin{aligned} PTS_\circ(\ell) &= \begin{cases} \iota & \text{if } \ell = \ell_\circ \\ \bigsqcup_{PTA} \{ PTS_\bullet(\ell') \mid (\ell', \ell) \in CFG \} & \text{otherwise} \end{cases} \\ PTS_\bullet(\ell) &= f_\ell(PTS_\circ(\ell)) \end{aligned} \tag{3.1}$$

The points-to analysis is a forward dataflow analysis and should act almost similar to a may analysis, such that the points-to sets are over-approximations representing that a variable may be assigned one of several possible values. We will define the combine operator $\bigsqcup_{PTA}$ later.

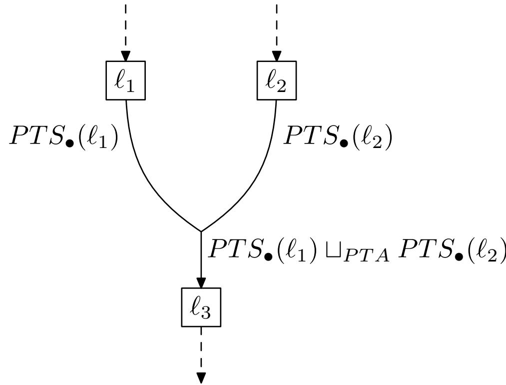

Figure 3.1: The combining of different facts according to (3.1).

In the following, we present examples that step-by-step describe the workings of the points-to analysis, what choices of precision we have considered, and which have

been chosen for the points-to analysis. We describe the behavior from Java source code examples, although the analysis traverses bytecode instructions. Java source code, however, is easier to understand. After presenting the behavior of the points-to analysis, we introduce the transfer functions that maintain this behavior formally, based on bytecode instructions. For the examples, we use the notations for PTS with a line number from the Java source code, instead of a location. This is to make the reasoning easier to express from Java source code, however caution should be made, not to confuse a line number in Java with a line number in bytecode. Typically one line of code in Java will be several instructions in bytecode.

3.3.1 Conversion to SSA

For the lock analysis to be described later in Section 3.4, the points-to set at a location where a lock is taken must be able to express what the object that is locked on points to at that location. However, in case the object locked on, points to several different values as a result of different flows that have been combined by the analysis, the lock analysis should not approximate this by assuming all possible values for an object being locked on, as this would be too imprecise. The reason is, that at the location a lock is taken, the lock will only point to one of possibly several values at runtime, but it is undecidable which one at compile-time. The lock analysis then must not consider a region locked on an object undecidable at compile-time to be mutual exclusive with any other region, but the information of potential locks should be present anyways, as they may cause potential deadlocks. The points-to analysis handles this, by representing facts in accordance with SSA form, as described in 2.1. Therefore we introduce φ-functions.

In our analysis, we introduce so-called φ-*values*, which are single values, $\varphi_i = v$, for which the points-to analysis has a mapping, such that the φ-function, $\varphi(v)$, yields all destinations that v points to. We introduce the φ-values when combining facts from different control flows, that has different assignment values for the same source value. That functionality is introduced by defining the combine operator, $\bigsqcup_{PTA}$, such that it is not the component wise extension of union for tuples, but a slight modification. We define $\sqcup_{PTA} : PTS \times PTS \rightarrow PTS$ as:

$$
\begin{aligned}
PTS_1 \sqcup_{PTA} PTS_2 = \langle S_1, D_1 \rangle \sqcup_{PTA} \langle S_2, D_2 \rangle \\
= \left\{ (s,d) \;\middle|\; \begin{array}{l} PTS_1(s) = PTS_2(s) \vee \\ (d \in PTS_1(s) \wedge s \in S_1 \wedge s \notin S_2) \vee \\ (d \in PTS_2(s) \wedge s \in S_2 \wedge s \notin S_1) \end{array} \right\} \\
\bigcup \left\{ \begin{array}{l} \langle (s,d), (d,d_1), (d,d_2), ..., (d,d_n) \rangle \\ \;\middle|\; \begin{array}{l} PTS_1(s) \neq PTS_2(s) \wedge \\ (d_i \in PTS_1(s) \vee d_i \in PTS_2(s)) \wedge \\ s \in S_1 \wedge s \in S_2 \wedge \varphi(d) := \{d_1, d_2, ..., d_n\} \end{array} \end{array} \right\}
\end{aligned}
$$

The consequence of introducing SSA form, is that the facts are added intermediate nodes, that has been given a new, unique value, corresponding to a φ-value, and these values then points to several other values in the points-to set. Thereby the lock analysis can represent an undecideable lock taken with a φ-value.

3.3.2 Variable Assignments

The analysis needs to be able to track that a given variable has been assigned a value or instance reference (depending on if the variable is value-type or reference-type, respectively). In Listing 3.2, the variable `o1` is assigned a new instance of type `Object`.

```
1 public class PointsToExample {
2   public void assignNewInstance() {
3     Object o1 = new Object(); /* o1=2, new Object=3 */
4     Object o2 = o1;           /* o2=4 */
5     /* etc. */
6   }
7 }
```

Listing 3.2: An example of an assignment of one local variable to another.

At line 3 the information computed from the points-to analysis must represent the assignment to `o1` in all facts for locations succeeding line 3 in the CFG. Therefore the fact $PTS_{\circ}(3) = \emptyset$ reaching line 3 will be extended with the points-to information for `o1` - a fresh value, that represents the source variable `o1`, say the number 2, and a fresh value that represents the instance created by `new Object()`, say 3. The outgoing fact then becomes $PTS_{\bullet}(3) = \{\langle 2, \{3\}\rangle\}$.

Other than just tracking what values or instance references variables may be assigned to, the points-to analysis must also be aware what other variables a given variable may point to. In Listing 3.2, line 4, variable `o1` is assigned to `o2`, and we take a look at what then should happen. At location l_3 the output fact is $PTS_{\bullet}(3) = \{\langle 2, \{3\}\rangle\}$. So the input fact in line 4 is $PTS_{\circ}(4) = \{\langle 2, \{3\}\rangle\}$. In line 4, the assignment to `o2` must be added to the fact. We are now left with two options: The destination can either be the value assigned to `o1`, namely 2, or the analysis can query $PTS_{\circ}(4)$ to reveal that `o1` points to 3. The analysis should not loose information required for the concurrent points-to analysis, which takes into account that multiple threads may execute anywhere in the class simultaneously. However, the variable pointed to, `o1`, is a local variable and cannot be shared between threads, as threads each have their own execution stack. Therefore the analysis can safely use 3 as the points-to destination. Assuming the local variable `o2` is represented by the unique value 4, the output fact from line 4 then becomes $PTS_{\bullet}(4) = \{\langle 2, \{3\}\rangle\,;\langle 4, \{3\}\rangle\}$.

With global variables, on the other hand, the points-to analysis must do otherwise. The example in Listing 3.3 shows a local variable `o4` that is assigned to a global variable `o3`.

```
1  public class PointsToGlobalExample {
2    private final Object o1 = new Object(); /* o1=2, new Object()=3; */
3    private final Object o2 = new Object(); /* o2=4, new Object()=5; */
4    private Object o3 = o1;                 /* o3=6 */
5
6    public void assignVariable() {
7      Object o4 = o3;                       /* o4=7 */
8    }
9    public void changeGlobal() {
10     o3 = o2;
11   }
12 }
```

Listing 3.3: In the example, `o3` may point to both `o1` and `o2`, when multiple threads execute within the class. Therefore `o4` may not map the points-to destination all the way back to the instantiation of `o1`, or we will lose information for the concurrent points-to analysis.

In Listing 3.3 we assume that the analysis assigns values to the variables and instantiations according to the comments. As initializations of `final` fields during the initializations of objects are guaranteed to be performed before an instance of `PointsToGlobalExample` may be visible to any thread, it might seem reasonable to assume the input fact at ℓ_7 such that `o3` maps directly to the instantiation with value 3. However, as soon as the object is published after the initialization, several threads may access the object and that means `o3` may point to either of `o1` and `o2`. It might not even have been initialized yet, because `o3` is not declared final! The points-to analysis may therefore not map `o4` to point to other than `o3`. At a later time the concurrent points-to analysis is responsible of collecting all possible points-to destinations of `o3`, and not until then can it be determined which instantiations `o4` may point to.

Another example must be given attention, that illustrates further behavior that must be put in the points-to analysis in order not to lose precision before the concurrent points-to analysis.

In Listing 3.4 line 4, a local variable `o2` is assigned the value of the global variable `o1`. In line 5 the global variable `o1` is changed, however, that does not influence the value `o2` points to. As `o1` is then assigned the value of `o2` in line 6, the previous behavior described, would traverse what `o2` points to at that location, which would yield the new value `o1` was assigned in line 5. This, however is not correct, and to compensate, the points-to analysis must do more, when assigning a variable the value of a global variable. What it does, is that if a variable is assigned to a global variable, then the points-to analysis must add both the value of that global variable and all leaf node destinations that global variable has in the points-to set. The com-

```
1  public class PointsToLimitationExample {
2    private Object o1 = new Object(); /* o1=2, new Object()=3 */
3    public void doAssign() {
4      Object o2 = o1;          /* o2=4 */
5      o1 = new Object();       /* new Object()=5 */
6      o1 = o2;
7    }
8  }
```

Listing 3.4: An example demonstrating a tricky self-assignment to a global variable, which the points-to analysis must also handle right.

putation of points-to information would then compute as follows within the method `doAssign()`: $PTS(3) = \langle(2,3)\rangle \rightarrow PTS(4) = \langle(2,3),(4,2),(4,3)\rangle \rightarrow PTS(5) = \langle(2,5),(4,2),(4,3)\rangle \rightarrow PTS(6) = \langle(2,5),(2,3),(4,2),(4,3)\rangle$. This behavior is an approximation that yields the points-to information about `o1`, that it may have in a multi-threaded environment accessing the class, thus the points-to information is correct, with respect to the concurrent points-to analysis described later, but a necessary over-approximation regarding the intra-procedural points-to analysis.

```
1  public class LoopAssignmentExample {
2    private Object o1;
3    public void loopAssignment() {
4      int i = 0;
5      do {
6        o1 = new Object();
7        i++;
8      }
9      while (i<100);
10     /* etc */
11   }
12 }
```

Listing 3.5: A variable `o1` is assigned 100 times in a loop.

Looping structures, such as `for`, `while`, and `do...while`, all result in a cycle in the control flow. For such control flow, the analysis only traverses until it reaches a node already seen before. Thereby it will collect all assignments within the looping structure once, which is enough for the following analyses. E.g., if a field is not safely assigned to (with respect to thread-safety) within a looping control flow, the unsafety introduced by that assignment is present first time the loop construct evaluates, as well as all other iterations. In Figure 3.2 a cycle is present in the control flow, here illustrating the `do...while` construct from Listing 3.5. The fact after the cycle is unaffected by the cycle; the points-to information within is only added once.

The general approach of the points-to analysis regarding variable scope of visibility, is that it safely can map in depth as long as the points-to destinations are local variables, but as soon it is a global variable, it must append both that global variable and all leaf-nodes that variable points to, to the destinations.

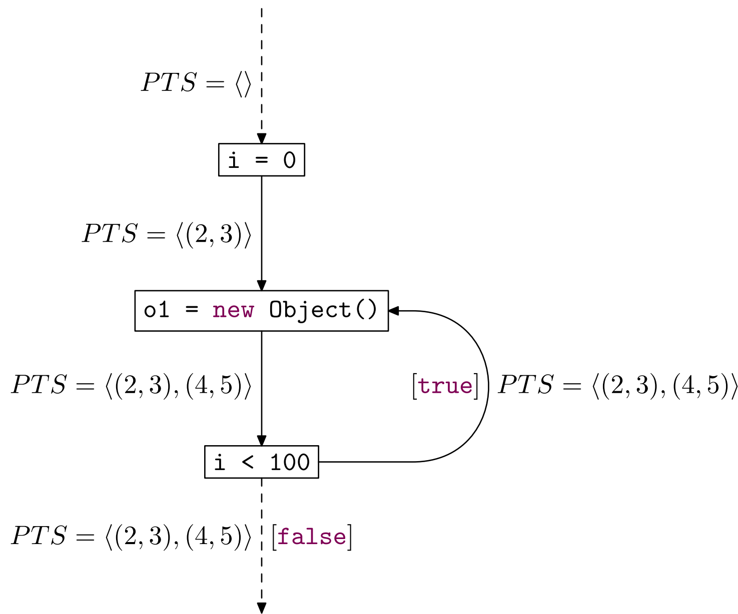

Figure 3.2: Control flow with a cycle, corresponding to listing 3.5

3.3.2.1 Transfer Function

We now turn to describing the transfer functions for the bytecode instructions, that come in play for assignments. The instructions that store a value are `PUTFIELD` for global variables, and `ISTORE`, `LSTORE`, `FSTORE`, `DSTORE`, and `ASTORE` for local variables. The transfer functions rely on the stack-function $stack(\ell) = S_\ell$, which we assume available and yielding the top element with $top(S_\ell)$ corresponding to the element that is the value of the assignment. The definitions of the functions *cpindex* and *lindex* can be found in Section 3.2. For convenience, we define a function $traverse(PTS(\ell), s)$ that traverses the points-to set for all points-to information for the value s. The function yields a set containing all possible leaf-nodes of the subtree with s as root and in addition, all global variable values in the subtree, that are not dominated by another global variable value in the subtree with s as root. This set accomodates the special handling required for assigning a variable the value of a global variable. The Table 3.1 shows the transfer functions for assignments.

3.3.3 Constructors

In the previous section, neither of the examples contain constructors. When introducing these, more information can be added the points-to information. If a class

$ins(\ell)$	$PTS_\bullet(\ell) = f_\ell(PTS_\circ(\ell))$:
PUTFIELD	$[Given : s = cpindex(ins(\ell))]$ $target(ins(\ell)) =$ this : $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell) \setminus \langle(s, PTS_\circ(\ell)(s))\rangle \sqcup_{PTA}$ $\bigsqcup_{PTA} \{(s,d) \mid \forall d \in traverse(top(S_\ell))\}$ $otherwise$: $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell)$
ISTORE, LSTORE, FSTORE, DSTORE or ASTORE	$[Given : s = lindex(ins(\ell))]$ $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell) \setminus (s \times PTS_\circ(\ell)(s)) \sqcup_{PTA}$ $\bigsqcup_{PTA} \{(s,d) \mid \forall d \in traverse(top(S_\ell))\}$

Table 3.1: Transfer functions for variable assignments.

has a single constructor, this constructor is known to have been executed, before the object constructed publishes to the threads that may use the instance. If multiple constructors are available, the class-wise approach our tool takes cannot determine which constructor has instantiated the class, however one constructor has been run to instantiate the object. The matter of constructors does not apply to static methods however, so the following only applies to non-static methods.

In general, when constructors are available, our analysis should be capable of knowing the values that variables have been assigned by a constructor, in advance of analyzing any non-static method. To do this, we make sure that the points-to analysis has analyzed constructors and built facts for these, before analyzing any methods. However, if several constructors exist, we are left with two options for the behavior of the analysis. Either methods have to be analyzed with as many initial facts as there are constructors, or the facts after each constructor should be combined using $\bigsqcup_{PTA}$ prior to analyzing methods. The first option is the most precise, as it may reveal errors specific to a certain constructor, but it also requires $n-1$ more traversals of each method, compared to the second approach, if there are n constructors. The second approach however, is not able to reveal which constructor may cause a thread-safety violation, but apart from that, it can be made equally precise. The critical aspect to get right in the second approach, is that although a field variable may have been initialized with different values by different constructors, it will only have one of these values and not potentially several of them, because a constructor is only run by one thread amongst those sharing the access to the object[1], implying no thread interleavings in constructors.

Our analysis uses the second approach. To be able to distinct the initial values a variable may have been assigned by constructors, we introduce a function $initial(v)$ that

[1] A constructor invoked by two different threads, yields two different instances, and can not be shared amongst the two threads, until the constructors finishes and the instance becomes visible. However the `this` reference may escape if a new thread is staring in the constructor, which in general is bad code practice.

a value v can be tested with, yielding `true` if that value is assigned by a constructor and `false` if not.

3.3.4 Method Invocation

Apart from instructions that directly assign a value or instance, as the ones that we have described previously, other instructions may contribute with points-to information. These are instructions that invoke a method, inside or outside the class of interest. Commonly, two approaches can be applied on dispatches, either context insensitivity or context sensitivity.

A context sensitive points-to analysis is the most accurate approximation. It is costly though, as it requires to traverse a dispatch from every context it occurs in, meaning that the same method may be analyzed several times, but with different initial analysis information, ι. In our class-wise approach, we strive for good accuracy and will therefore apply context sensitivity, although not to all dispatches.

3.3.4.1 Context Sensitivity Inside Class

Our analysis will handle dispatches to private and final methods within the class of interest sensitive to context, as we have the adequate context information immediately available here. The reason is, that private methods and non-final methods may not be overwritten by a SPA in classes inherited from the class of interest. Thus, the SPA cannot alter the behavior of these methods, and therefore the information they may add to the points-to set, is determinied by the context they are called in.

The context information required is simply the points-to set at the location where a dispatch occurs. Here, a mapping to the arguments of the dispatch is done, such that the values of the parameters correspond to what the arguments point to in the current points-to set. Then the analysis is performed on the target of the dispatch, with the mapped arguments and the current points-to set as initial analysis information. The outcome of the analysis of the dispatch method is then combined into the current points-to set, such that the following locations have the information from the dispatch also. We denote the set of return facts $\mathcal{R}_{PTA}(m, \iota)$ for the method m with the initial analysis information ι. Thereby the outcome of the analysis of an invoked instruction becomes $\bigsqcup_{PTA} \mathcal{R}_{PTA}(m, PTS_\circ(\ell))$. Note that exit points may be unhandled exception edges as well as return points, such that all possible facts on exit points are combined into the resulting points-to set.

With this approach comes a problem similar to that of looping constructs, namely recursive method invocations. This does not introduce cycles in the CFG, but instead in the *call graph*, that is the information of where invocations occur and which method they invoke. Our analysis does not compute a call graph in advance whereon cycle

detection could be performed. Instead, the analysis is aware of which locations there has already been analyzed a dispatch from and what return fact that yielded. The analysis will then only be performed again from that location, if the return fact changes compared to the last return fact. For the points-to analysis this clearly terminates, as the same assignments in the dispatched method will yield the same values after the first traversal by the analysis only. Notice that this behavior also applies to analysis of the constructors, such that one constructor that invokes a method, performs an analysis to get the return fact of that invocation, in order to propagate accurate points-to information in the following context.

3.3.4.2 Context Insensitivity Outside Class

Concerning non-final, non-private dispatches within the class of interest and dispatches that do not target a method within the class of interest, context sensitivity gets harder to realize. For non-final, non-private dispatches within the class, the class may be extended and the SPA can then introduce all sorts of behaviors that will change the class-invariant. For dispatches to methods outside the class of interest, first of all, the instance that the dispatch is performed on, may have been given as parameter to some method in the class of interest. Thereby, we cannot determine anything about its internal state. Though, for objects created within the context of the class of interest that are not escaped, an analysis of it self could approximate the state of that object. However, a precise context-sensitive analysis to determine the state of objects within the class, would require flow-insensitivity, as the class of interest can be concurrently accessed, and it would require the information about which locks are held at what location. Therefore we decide not to create such a costly analysis and instead choose a context-insensitive approach to dispatches with targets that are not within the class of interest and non-final, non-private dispatches within the class of interest.

The context-insensitive approach the analysis performs, then will not be able to determine what can happen to variables passed as parameters to these dispatches. Therefore, if a field from the class of interest is passed a method invocations either extendable or outside the class context as argument, the safe over-approximation on the behavior will be that that field has escaped the class of interest, which violates thread-safe behavior, as the state then may not be preserved. Similarly, the return value of such a dispatch must necessarily also be over-approximated with an escaped value, as the context insensitive approach cannot determine if that returned instance is used outside the class or not. These issues should be solved by the escape analysis, which is, however, left for future work.

In bytecode, the two kinds of dispatches that are analyzed context insensitive, need a little attention to get right. First of all, the method that is invoked by an invoke instruction (see section 2.4) is identified by an index in the constant pool. However, this does not distinct the object on which the method is invoked, which is information

we will need in the lock analysis to be able to properly identify reader- and writer-locks, as the invocations of either `readLock()` or `writeLock()` on the same object of type `ReentrantReadWriteLock`, yields the same reader- or writer-lock respectively at each invocation. The object that the dispatch is on, is on top of the stack when the dispatch occurs. The analysis then requests the value of the object with $top(\ell)$ and creates a mapping from that value and the particular method index in the local variable frame to a fresh, unique value. Points-to information is then inserted, such that the return value of the dispatch points to the new fresh value, which in turn points to the value of the object the dispatch was performed on, that is $top(\ell)$. The points-to set is then able to map from a return value to a unique value, representing the object the invocation was performed on and the index of the dispatch in the constant pool, which is a requirement for the lock analysis later. Furthermore, the points-to set maps from the unique value to the value of the object that the invocation was performed on, which becomes necessary in the concurrent points-to analysis. For an example of the behavior, review Listing 3.6 and the corresponding bytecode part in Listing 3.7.

```
1  public class InvokeReadLockExample {
2    private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();  /* rwl=2,
       new Ree..=3 */
3    public void doReaderAction1() {
4      Lock rl = rwl.readLock();  /* rl=4, readLock()=5 */
5      rl.lock();
6      try {
7      /* Do reader action */
8      }
9      finally {
10       rl.unlock();
11     }
12   }
13   public void doReaderAction2() {
14     Lock rl = rwl.readLock();  /* rl=7, readLock()=8 */
15     rl.lock();
16     try {
17     /* Do reader action */
18     }
19     finally {
20       rl.unlock();
21     }
22   }
23 }
```

Listing 3.6: An example where the points-to set must be able to map the instances of ReadLocks to the same value, when an instance is returned in line 4 and 14.

In Listing 3.7, location 0 pushes `this` on the stack (local variable index 0). Then `this` is popped from the stack by `GETFIELD` and the `ReentrantReadWriteLock` field is fetched from index 4 in `this` and pushed to the stack. At location 4, the method `readLock()` that returns the `ReadLock` of the `ReentrantReadWriteLock` is invoked, the `ReentrantReadWriteLock` is popped from the stack and the `ReadLock` pushed. That object is then popped from the stack and stored in the local variable index 1,

```
1  0:  aload_0;
2  1:  getfield   #4; //Field rwl:Ljava/util/concurrent/locks/ReentrantReadWriteLock;
3  4:  invokevirtual #5; //Method java/util/concurrent/locks/ReentrantReadWriteLock.
       readLock:()Ljava/util/concurrent/locks/ReentrantReadWriteLock\$ReadLock;
4  7:  astore_1
5  8:  aload_1
6  9:  invokeinterface #6,  1; //InterfaceMethod java/util/concurrent/locks/Lock.lock
       :()V
```

Listing 3.7: Bytecode instructions for the lines 4-5 in Listing 3.6 in method `doReaderAction1`.

corresponding to the `Lock` l from the example Java source. The actions until now, that is from location 0 to 7, are the ones performing the assignment in line 4 in the `doReaderAction1()` method of the Java source. Then the `ReadLock` is fetched from local variable index 1 and pushed to the stack, followed by the invocation of the `lock()` method, that also pops the `ReadLock` again.

Figure 3.3 shows how the points-to analysis computes the points-to set for the bytecode in Listing 3.7. The method `doReaderAction1()` is analyzed with the initial analysis information that `rwl` points to `new ReentrantReadWriteLock()`, that is $\langle 2, 3 \rangle$.

Now return to the example in Listing 3.6. From the behavior of the points-to analysis demonstrated in Figure 3.3, the points-to analysis has, when first seen the invocation of `rwl.readLock()`, stored a value for this particular method invocation with the `ReentrantReadWriteLock` in `rwl`. This value, 6, is identified by the value of `rwl`, 2, and the constant pool index where the method `readLock()` is allocated, 5. Thereby the Java source line 14 (which has the same bytecode instructions as in listing 3.7) will result in that `rl` will point to the return value 7, which in turn then points to the same value, 6, introduced for the invocation earlier. Then 6 again points to 2, the value for the field rwl.

3.3.4.3 Transfer Function

In bytecode, the instructions `INVOKEINTERFACE`, `INVOKESPECIAL`, and `INVOKEVIRTUAL` are the instructions that cause dispatches. We utilize functions that are defined in Section 3.2, which may be inspected for further explanation. We denote the set of return facts for a method m with initial analysis information ι as $\mathcal{R}_{PTA}(m, \iota)$. For a method invocation on m, with $\iota = PTA_\circ(\ell)$ the transfer functions should compute the points-to set given by $PTA_\bullet(\ell) = f_\ell(PTA_\circ(\ell)) = \bigsqcup_{PTA} \mathcal{R}_{PTA}(m, \iota)$. Said in an informal way we combine all the return facts according to the binary operator $\sqcup_{PTA}$. The function $retval(t, i)$ is a function that yields a unique value corresponding to a pair of target object t and constant pool index i.

The transfer functions for dispatches are shown in Table 3.2.

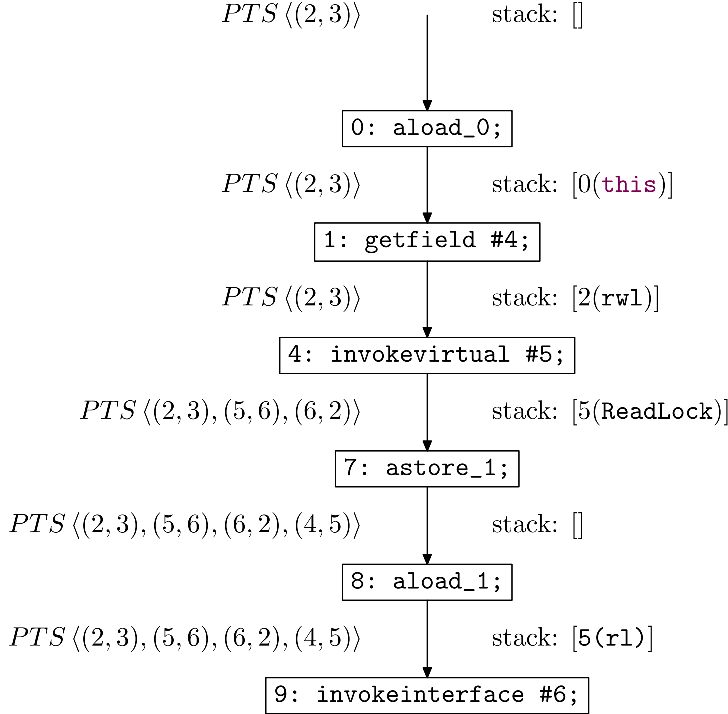

Figure 3.3: Illustration of how the points-to analysis computes the points-to set for the bytecode in listing 3.7. On the left, the points-to set is built up and on the right the stack is shown along the traversal. The values correspond to the values in the comments in listing 3.6. The dispatch at ℓ_4 introduces new points-to information in the points-to set, such that the return value 5 points to a fresh value 6, representing the pair of object and index in the constant pool $((2, 5))$, and that the fresh value 6 points to the object 2 on which the invocation was performed.

$ins(\ell)$	$PTS_\bullet(\ell) = f_\ell(PTS_\circ(\ell)):$
`INVOKEINTERFACE`, `INVOKESPECIAL`, `INVOKEVIRTUAL`	$\left[Given: \begin{array}{l} m = method(ins(\ell)), t = target(ins(\ell)), \\ i = cpindex(ins(\ell)), r = retval(t, i) \end{array} \right]$ $t = \texttt{this} \wedge m$ is $\texttt{final} \vee \texttt{private}:$ $f_\ell(PTS_\circ(\ell)) = \bigsqcup_{PTA} \mathcal{R}_{PTA}(m, PTS_\circ(\ell))$ $otherwise:$ $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell) \sqcup_{PTA} \langle (r, new), (new, t) \rangle$

Table 3.2: Transfer functions for dispatch instructions.

3.3.5 Accessing Field Variables Outside Class

Now, one last thing remains for the transfer functions of the points-to analysis. Namely, that GETFIELD instructions may push the value of a field variable that is not a member of this. This somewhat resembles how we handle dispatches context insensitive.

```
1 public class GetFieldExample {
2   public void getField(A a) {
3     b = a.b;
4     /* etc. */
5   }
6 }
```

Listing 3.8: A field variable with type b from the object A is fetched.

The example in Listing 3.8 references a field from another object than this. This implies a GETFIELD instruction in bytecode, which gets the field from the object on top of the stack; in this case A. Again our analysis is not capable of determining any information about that field, because it may be accessed by a SPA and therefore the state is unknownm, thus it is treated just like a return value as in the case of context insensitive analysis of dispatches.

3.3.5.1 Transfer Function

That then leads us to a transfer function for GETFIELD almost similar to that of dispatches analyzed context insensitive and is defined in table 3.3. The function $retval(t, i)$ is described previously for method invocations to methods outside the class of interest. The *new* function yields a fresh value.

$ins(\ell)$	$PTS_\bullet(\ell) = f_\ell(PTS_\circ(\ell))$:
GETFIELD	$\left[Given: \begin{array}{l} m = method(ins(\ell)), t = target(ins(\ell)), \\ i = cpindex(ins(\ell)), r = retval(t, i) \end{array} \right]$ $t \neq$ this : $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell) \sqcup_{PTA} \langle (r, new), (new, t) \rangle$ $otherwise$: $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell)$

Table 3.3: Transfer function for the GETFIELD instruction.

3.3.6 Summary

The points-to analysis has now been introduced, both by example and formalized in transfer functions. We sum up with a table containing an overview of the transfer

functions used for the points-to analysis in Table 3.4.

$ins(\ell)$	$PTS_\bullet(\ell) = f_\ell(PTS_\circ(\ell))$:
PUTFIELD	$[Given : s = cpindex(ins(\ell))]$ $target(ins(\ell)) = \texttt{this}$: $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell) \setminus \langle(s, PTS_\circ(\ell)(s))\rangle \sqcup_{PTA}$ $\bigsqcup_{PTA} \{(s,d) \mid \forall d \in traverse(top(S_\ell))\}$ $otherwise$: $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell)$
ISTORE, LSTORE, FSTORE, DSTORE or ASTORE	$[Given : s = lindex(ins(\ell))]$ $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell) \setminus (s \times PTS_\circ(\ell)(s)) \sqcup_{PTA}$ $\bigsqcup_{PTA} \{(s,d) \mid \forall d \in traverse(top(S_\ell))\}$
INVOKEINTERFACE, INVOKESPECIAL, INVOKEVIRTUAL	$\left[Given : \begin{array}{l} m = method(ins(\ell)), t = target(ins(\ell)), \\ i = cpindex(ins(\ell)), r = retval(t,i) \end{array}\right]$ $t = \texttt{this} \wedge m$ is $\texttt{final} \vee \texttt{private}$: $f_\ell(PTS_\circ(\ell)) = \bigsqcup_{PTA} \mathcal{R}_{PTA}(m, PTS_\circ(\ell))$ $otherwise$: $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell) \sqcup_{PTA} \langle(r, new), (new, t)\rangle$
GETFIELD	$\left[Given : \begin{array}{l} m = method(ins(\ell)), t = target(ins(\ell)), \\ i = cpindex(ins(\ell)), r = retval(t,i) \end{array}\right]$ $t \neq \texttt{this}$: $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell) \sqcup_{PTA} \langle(r, new), (new, t)\rangle$ $otherwise$: $f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell)$
otherwise	$f_\ell(PTS_\circ(\ell)) = PTS_\circ(\ell)$

Table 3.4: Transfer functions for the points-to analysis.

3.4 Lock Analysis

The goal of the lock analysis is to provide lock informations at all reachable locations within a given class context. The analysis depend on the points-to analysis in order to map the target(s) of an instruction to the correct object(s). Without the points-to analysis it would be impossible to know what object a lock or unlock operation is performed on, so by utilizing the points-to analysis we can at a given location resolve the object that is used as a synchronization primitive, and thereby lock on the correct object. As already mentioned, we have decided to support the `synchronized` primitive and subclass of `java.util.concurrent.Lock` including readers writer lock semantics.

As for the points-to analysis the lock analysis is an intra-procedural analysis and context sensitive for private or final methods within the context of the class.

At a given location ℓ, the lock information is represented as a "lock set" which is defined as the power set:

$$LS(\ell) = \mathcal{P}(L \times N)$$

where $L = \mathbb{N}$ and $N = \mathbb{N} \cup \bot$. For a specific pair (l, n), the value l represents the lock and n represents the lock count. The lock analysis is like the points-to analysis a forward dataflow analysis, and below we define the expressions used for computing the incoming and outgoing lock information at location ℓ.
We will use the notion of a lock set before and after a location as $LS_\circ(\ell)$ and $LS_\bullet(\ell)$, respectively:

$$\begin{aligned} LS_\circ(\ell) &= \begin{cases} \iota & \text{if } \ell = \ell_\circ \\ \bigsqcup_{LS} \{ \; LS_\bullet(\ell') \mid (\ell', \ell) \in CFG \; \} & \text{otherwise} \end{cases} \\ LS_\bullet(\ell) &= f_\ell(LS_\circ(\ell)) \end{aligned} \tag{3.2}$$

The transfer function f_ℓ is responsible for making the necessary modifications on the lock set $LS_\circ(\ell)$ yielding the modified lock set $LS_\bullet(\ell)$. Only in locations of interest will the transfer function modify the lock set, namely if a lock is taken or a lock is released. In the case that neither a lock or unlock operation is performed the transfer function simply yields: $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell)$. A summary of the transfer function is found in Section 3.4.3.
The lock analysis is neither a may or a must analysis, because combining lock sets is neither done by using union or intersection. The lock analysis however resembles a must analysis because the information it provides is guaranteed to hold. We therefore explicitly define the combine operator for the lock analysis, $\bigsqcup_{LS} : \mathcal{P}(LS) \to LS$. We assume the existence of a function $locks : LS \to L$ that yields the set of locks held at a given location ℓ. We then define $\sqcup_{LA} : LS \times LS \to LS$ by:

$$\begin{aligned} ls_1 \sqcup_{LA} ls_2 = & \left\{ (l, n) \;\middle|\; \begin{array}{c} l \in locks(ls_1) \\ \wedge \\ l \in locks(ls_2) \end{array}, n = \left\{ \begin{array}{ll} n & (l, n) \in ls_1 \wedge (l, n) \in ls_2 \\ \bot & otherwise \end{array} \right\} \right\} \\ & \bigcup \{ (l, \bot) \mid l \in locks(ls_1) \setminus locks(ls_2) \} \\ & \bigcup \{ (l, \bot) \mid l \in locks(ls_2) \setminus locks(ls_1) \} \end{aligned} \tag{3.3}$$

The reason why the lock analysis is designed to resemble a must analysis is because we need to be able to determine what locks are guaranteed to be held at a given location ℓ. However if the analysis was a may analysis, we could not use it to determine if a critical region is guarded by a specific lock at all time, thereby making it worthless for our purpose.

3.4.1 Locking and Unlocking

The lock analysis must compute the necessary modifications to the lock set at locations where lock and unlock operations occur. A lock operation will always target some object, meaning that a lock cannot just be taken without anyone owning the lock.
Listing 3.9 illustrates the need for a function that returns distinct locks for `MONITOR` and `INVOKEINTERFACE` instructions when invoked on the same object. This is because all subclasses of `Object` can be used as a monitor lock, thus subclasses of `java.util.concurrent.Lock` may also be used as a monitor lock. Because the lock taken by invoking lock on, e.g., a `ReentrantLock` is not the same as taking the monitor lock on that particular instance, a mutual exclusive region cannot be implemented by using a mixture of these.

```
1  public class DeterministicLock {
2    private final Lock rl1 = new ReentrantLock();
3    public void useLocking() {
4      rl1.lock();
5      try {
6        /*Make actions depending on that no thread
7        will be in the region guarded by the monitor*/
8      }
9      finally{
10       rl1.unlock();
11     }
12   }
13   public void useMonitor() {
14     synchronized(rl1)
15       /*Make actions depending on that no thread
16       will be in the region guarded by the lock*/
17     }
18   }
19 }
```

Listing 3.9: A failed attempt to construct a critical region, using a common object to lock on and use as a monitor lock.

From now on, we assume the existence of a function $lock_{id}(ins(\ell), target(ins(\ell))) = l$ where $l \in L$ that given the same target, returns distinct locks for `MONITOR` and `INVOKEINTERFACE` instructions.
Before defining the transfer functions, we describe a number of examples illustrating the functionality of the transfer function. Listing 3.10 shows an example where a `ReentrantLock` is used to protect a region within the class.

As already mentioned only the synchronization primitives will modify the lock set, meaning that the lock set will only be modified at line 4 and 9 where respectively a `lock` and `unlock` operation is done. The entry lock set ι for the method `doLocking()` is the empty set, and therefore no locks will be held before returning from the `lock()` method call in line 4.

The compiled bytecode for the Listing 3.10 is shown in Listing 3.11.

```
1  public class DeterministicLock {
2    private final Lock rl1 = new ReentrantLock(); /* rl1=2, new ReentrantLock()=3 */
3    public void doLocking() {
4      rl1.lock();
5      try {
6        //Make computation
7      }
8      finally{
9        rl1.unlock();
10     }
11   }
12 }
```

Listing 3.10: A critical region that is protected by a lock that is deterministic at all time.

```
1  0:   aload_0
2  1:   getfield        #15; //Field rl1:Ljava/util/concurrent/locks/Lock;
3  4:   invokeinterface #22,  1; //InterfaceMethod java/util/concurrent/locks/Lock.
        lock:()V
4  9:   aload_0
5  10:  getfield        #15; //Field rl1:Ljava/util/concurrent/locks/Lock;
6  13:  invokeinterface #27,  1; //InterfaceMethod java/util/concurrent/locks/Lock.
        unlock:()V
7  18:  return
```

Listing 3.11: The bytecode for the Java code in example 3.10.

At location ℓ_4 in the bytecode a lock is being acquired, and before invoking the `lock` method the lock set is given by $LS_\circ(\ell_4) = \{\}$, whereas afterwards $LS_\bullet(\ell_4) = \{(5,1)\}$ assuming $lock_{id}(ins(\ell_4), target(ins(\ell_4))) = 5$. In this example $target(ins(\ell_4))$ will resolve to the object that `rl1` points to, namely the instance of `ReentrantLock`. At the unlock call at ℓ_9 the incoming lock set is given by $LS_\circ(\ell_9) = \{(5,1)\}$, thus $LS_\bullet(\ell_9) = \{\}$.

As seen in definition 3.3, the combine operator will assign a specific lock, l, the lock count $\bot$, when combining locksets where the lock count for l differs. In example 3.12 a branch is used to determine if the lock, `rl1`, should be taken.

```
1  public class BottomLock {
2    private Lock rl1 = new ReentrantLock();
3    public void doLocking(boolean b) {
4      if(b) rl1.lock();
5      else //Other action
6    }
7  }
```

Listing 3.12: Locking is performed on a non-deterministic lock, because the value of the boolean b is unknown at compile-time.

Because only one branch results in a method call to `lock`, the lock sets after the `if` and `else` branches will contain a different number of lock counts, thus an empty lock set is combined with a lock set containing one lock with count 1. Expressed formally combining the two lock sets will yield: $LS_\bullet(4) \sqcup_{LA} LS_\bullet(5) = \{(2,1)\} \sqcup_{LS} \{\} = \{(2,\bot)\}$ assuming $lock_{id}(ins(4), target(ins(4))) = 2$.

Listing 3.13 shows a less trivial example which illustrates locking on a undecidable lock, thus the boolean `b` might not be known at compile-time.

```
1  public class NonDeterministicLock {
2    private Lock rl1 = new ReentrantLock();
3    public void doLocking(boolean b) {
4      Lock rl = b ? rl1 : rl2;
5      rl.lock();
6      try {
7        //Make computation
8      }
9      finally{
10       rl.unlock();
11     }
12   }
13 }
```

Listing 3.13: Locking is performed on a lock undecidable at compile-time, because the value of the boolean b is unknown.

Because it would be a under approximation (in the case of mutual exclusion) to take the lock on both object instances that `rl` might point to, another solution had to be found. Fortunately the points-to analysis guarantees that there exists a intermediate φ-value, φ_ℓ, in the points-to set if a variable might point to different destinations, thus we have solved this problem by acquiring the lock, $lock_{id}(ins(\ell), target(ins(\ell)))$, assuming that $target(ins(\ell)) = \varphi_\ell$. Listing 3.14 illustrates how the reader lock belonging to a `ReentrantReadWriteLock` is used to protect a region.

```
1  public class DispatchAndAcquireLock {
2    private final Lock rwl = new ReentrantReadWriteLock();
3    public void useLocking() {
4      rwl.readLock().lock();
5      try {
6        /*Make read operation*/
7      }
8      finally{
9        rwl.readLock().unlock();
10     }
11   }
12 }
```

Listing 3.14: Illustrates a lock being acquired though the use of a private dispatch.

As described in Section 3.3 the $PTA(\ell)$ contains information about the relation between reader and writer locks belonging to the same `ReentrantReadWriteLock` in-

stance. Therefore the lock analysis does not need to make any special treatments for these kinds of locks, because the $target(ins(\ell))$ simply yields the correct target.

3.4.1.1 Transfer functions

We now turn our attention to the transfer functions for the lock analysis and formalize these. We start by defining the transfer functions for the `synchronized` primitive, namely the `monitor` instructions, which is shown in Table 3.5.

$ins(\ell)$	$LS_\bullet(\ell) = f_\ell(LS_\circ(\ell))$
MONITORENTER	$\left[Given : \begin{array}{l} l = lock_{id}(ins(\ell), target(ins(\ell))) \\ n = LS_\circ(\ell)(l) \end{array} \right]$ $(l, n) \notin LS_\circ(\ell)$: $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell) \cup (l, 1)$ $n \neq \perp \wedge n \geq 0$: $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l, n)) \cup (l, n + 1)$ $otherwise$: $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell)$
MONITOREXIT	$\left[Given : \begin{array}{l} l = lock_{id}(ins(\ell), target(ins(\ell))) \\ n = LS_\circ(\ell)(l) \end{array} \right]$ $n \neq \perp \wedge n > 0$: $f_\ell(LS_\circ(\ell))) = (LS_\circ(\ell) \setminus (l, n)) \cup (l, n - 1)$ $otherwise$: $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l, n)) \cup (l, \perp)$

Table 3.5: Transfer function entering and exiting a monitor.

Even though the `wait()`, `notify()` and `notifyAll()` methods all may affect concurrent behavior, we do not need to take these calls into account, because none of these methods will break mutual exclusion within a critical region, due to the semantics of the `MONITOR` instructions.

The transfer function for `java.util.concurrent.Lock` is almost identical, where the only difference is the instructions that will result in a modified lock set. See Table 3.6.

3.4.2 Method Invocation

As already described, the lock analysis is context sensitive for final and private methods within the context of the class being analyzed. Only context sensitive dispatches may change the lock set LS, thus a context insensitive dispatch yields

$ins(\ell)$	$LS_\bullet(\ell) = f_\ell(LS_\circ(\ell))$
INVOKEINTERFACE on java.util.concurrent.locks.lock.lock()	$\left[Given: \begin{array}{l} l = lock_{id}(ins(\ell), target(ins(\ell))) \\ n = LS_\circ(\ell)(l) \end{array}\right]$ $(l,n) \notin LS_\circ(\ell)$: $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell) \cup (l,1)$ $n \neq \bot \wedge n \geq 0$: $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l,n)) \cup (l, n+1)$ $otherwise$: $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell)$
INVOKEINTERFACE on java.util.concurrent.locks.lock.unlock()	$\left[Given: \begin{array}{l} l = lock_{id}(ins(\ell), target(ins(\ell))) \\ n = LS_\circ(\ell)(l) \end{array}\right]$ $n \neq \bot \wedge n > 0$: $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l,n)) \cup (l, n-1)$ $otherwise$: $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l,n)) \cup (l, \bot)$

Table 3.6: Transfer function for invoking `lock` and `unlock` on a subclass of `java.util.concurrent.locks.Lock`.

$LS_\circ(\ell) = f_\ell(LS_\circ(\ell))$. In this section we will describe the functionality of the transfer function regarding context sensitive dispatches.

Listing 3.15 illustrates a lock being acquired by invoking the private method, `acquireLock`. By invoking `acquireLock` the transfer function should transfer $LS_\circ(4) = \{\}$ into $LS_\bullet(4) = \{(2,1)\}$ assuming that $lock_{id}(ins(14), target(ins(14))) = 2$.

```
1  public class DispatchAndAcquireLock {
2    private final Lock rl1 = new ReentrantLock();
3    public void useLocking() {
4      acquireLock();
5      try {
6        /*Make actions depending on that no thread
7        will be in the region guarded by the monitor*/
8      }
9      finally{
10       rl1.unlock();
11     }
12   }
13   private void acquireLock() {
14     rl.lock();
15   }
16 }
```

Listing 3.15: Illustrates a lock being acquire though the use of a private dispatch.

The example illustrates that the lock set held when returning from `acquireLock` must be transferred to the caller and used as lock information at $LS_\bullet(4)$. When

analyzing the `acquireLock` method, the lock set $LS_\circ(4)$ must be transferred to the called method, thus $\iota = LS_\circ(4)$. Because a method may return in more than one place, e.g., due to multiple `return` statements or as a side effect of one or more unhandled exceptions, multiple return lock sets may exist. Therefore the transfer function must combine these return lock sets combining them into one. In the next section we formalize the transfer function for `final` and `private` method invocations.

3.4.2.1 Transfer functions

We denote the set of return facts for a method m as $\mathcal{R}_{LA}(m,\iota)$. For a method invocation on m, with $\iota = LS_\circ(\ell)$ the transfer functions should compute the return lock set given by $LS_\bullet(\ell) = f_\ell(LS_\circ(\ell)) = \bigsqcup_{LA} \mathcal{R}_{LA_m}$. Said in a informal way we combine all the return facts according to the binary operator $\sqcup_{LA}$. See Table 3.7.

$ins(\ell)$	$LS_\bullet(\ell) = f_\ell(LS_\circ(\ell))$
INVOKEINTERFACE INVOKESPECIAL INVOKEVIRTUAL	$\left[Given: \begin{array}{l} m = method(ins(\ell)) \\ \iota = LS_\circ(\ell) \end{array} \right]$ $target(ins(\ell)) = $ `this` $\wedge\, m$ is `final` $\vee$ `private` : $f_\ell(LS_\circ(\ell)) = \bigsqcup_{LA} \mathcal{R}_{LA}(m,\iota)$ $otherwise:$ $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell)$

Table 3.7: Transfer function for local `private` and `final` method invocations.

3.4.3 Summary

The lock analysis has now been introduced. A number of examples has illustrated the purpose of the lock analysis and transfer functions has been formalized. We sum up with a table containing an overview of the transfer functions that make up the lock analysis. See Table 3.8.

$ins(\ell)$	$LS_\bullet(\ell) = f_\ell(LS_\circ(\ell))$
MONITORENTER	$\left[Given: \begin{array}{l} l = lock_{id}(ins(\ell), target(ins(\ell))) \\ n = LS_\circ(\ell)(l) \end{array}\right]$ $(l, n) \notin LS_\circ(\ell):$ $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell) \cup (l, 1)$ $n \neq \bot \wedge n \geq 0:$ $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l, n)) \cup (l, n+1)$ $otherwise:$ $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell)$
MONITOREXIT	$\left[Given: \begin{array}{l} l = lock_{id}(ins(\ell), target(ins(\ell))) \\ n = LS_\circ(\ell)(l) \end{array}\right]$ $n \neq \bot \wedge n > 0:$ $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l, n)) \cup (l, n-1)$ $otherwise:$ $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l, n)) \cup (l, \bot)$
INVOKEINTERFACE on java.util.concurrent.locks.lock.lock()	$\left[Given: \begin{array}{l} l = lock_{id}(ins(\ell), target(ins(\ell))) \\ n = LS_\circ(\ell)(l) \end{array}\right]$ $(l, n) \notin LS_\circ(\ell):$ $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell) \cup (l, 1)$ $n \neq \bot \wedge n \geq 0:$ $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l, n)) \cup (l, n+1)$ $otherwise:$ $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell)$
INVOKEINTERFACE on java.util.concurrent.locks.lock.unlock()	$\left[Given: \begin{array}{l} l = lock_{id}(ins(\ell), target(ins(\ell))) \\ n = LS_\circ(\ell)(l) \end{array}\right]$ $n \neq \bot \wedge n > 0:$ $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l, n)) \cup (l, n-1)$ $otherwise:$ $f_\ell(LS_\circ(\ell)) = (LS_\circ(\ell) \setminus (l, n)) \cup (l, \bot)$
INVOKEINTERFACE INVOKESPECIAL INVOKEVIRTUAL	$\left[Given: \begin{array}{l} m = method(ins(\ell)) \\ \iota = LS_\circ(\ell) \end{array}\right]$ $target(ins(\ell)) = \texttt{this} \wedge m \text{ is } \texttt{final} \vee \texttt{private}:$ $f_\ell(LS_\circ(\ell)) = \bigsqcup_{LA} \mathcal{R}_{LA}(m, \iota)$ $otherwise:$ $f_\ell(LS_\circ(\ell)) = LS_\circ(\ell)$
otherwise	$f_\ell(LS_\circ(\ell)) = LS_\circ(\ell)$

Table 3.8: Transfer functions for the lock analysis.

3.5 Dominator Analysis

The dominator analysis is a prerequisite for the concurrent points-to analysis; the reason will be enlightened in the next section. The dominator analysis task is to reveal what variable assignments have been performed previous to the current location, but within the same locked region as the current location.

We initially define the fact the dominator analysis computes, called the dominator set or $DS(\ell)$, as:

$$DS(\ell) = \mathcal{P}(S \times \mathcal{P}(L))$$

where $S = \mathbb{N}$ and $L = \mathbb{N}$. Thereby an element (s, ls) represents that the assignment to the variable with value s is dominating under the set of locks in ls. We define the function on the dominator set $DS(\ell)(s) = ls$, which yields ls in the pair $(s, ls) \in DS(\ell)$.

The dominator analysis computes the dominator set in a forward manner on the CFG, such that we may express the input $DS_\circ(\ell)$ and output $DS_\bullet(\ell)$, which corresponds to the facts flowing forward in the control flow, as:

$$DS_\circ(\ell) = \begin{cases} \iota & \text{if } \ell = \ell_\circ \\ \bigsqcup_{DA} \{ \; DS_\bullet(\ell') \mid (\ell', \ell) \in CFG \; \} & \text{otherwise} \end{cases}$$

$$DS_\bullet(\ell) = f_\ell(DS_\circ(\ell))$$

The dominator analysis resembles a must analysis, as it will combine facts, such that the combining of sets results in that variable values not present on all paths will not be part of the resulting dominator set. However, the combining of assignments to the same variable under different lockset, that do not intersect, should also not be part of the result, which cannot be expressed by $\cap$ solely and therefore the combine operator for the dominator analysis, $\bigsqcup_{DA} : \mathcal{P}(DS) \to DS$, must be defined specifically. We define $\sqcup_{DA} : DS \times DS \to DS$:

$$ds_1 \sqcup_{DA} ds_2 = \left\{ (s, ls) \;\middle|\; \begin{array}{l} s \in S_1 \wedge s \in S_2, \\ ls = ds_1(s) \cap ds_2(s) \neq \emptyset \end{array} \right\}$$

Assuming that $ds_1 = \mathcal{P}(S_1 \times \mathcal{P}(L))$ and $ds_2 = \mathcal{P}(S_2 \times \mathcal{P}(L))$. An example of this behavior can be that $\{(1, \{8\}), (2, \{8, 9\}), (3, \{8\}), (4, \{8, 10\})\}$ combines with $\{(1, \{8\}), (2, \{9\}), (4, \{9\})\}$ which results in the dominator set $\{(1, \{8\}), (2, \{9\})\}$.

The transfer functions for the dominator analysis vary and either are a function of the instruction and the top of the stack at the current location, $f_\ell(ins(\ell), top(S_\ell))$ or only a function of the current location, implicitly already denoted by f_ℓ.

In the following we demonstrate the dominator analysis behavior from examples and later formalize the transfer functions that define the analysis behavior.

3.5.1 Variable Assignments

First of all, the dominator analysis must compute the dominator sets based on the locations where variables are assigned.

```
1 public void DominatorAssignmentExample {
2   public void doAssign() {
3     Object o3 = new Object();               /* o3 = 2, new Object()=3 */
4     synchronized(this) {                    /* this=1 */
5       Object o2 = new Object();             /* o2 = 4, new Object()=5 */
6       /* etc */
7     }
8   }
9 }
```

Listing 3.16: An example used to demonstrate the dominator analysis.

Listing 3.16, shows an example Java class for which the dominator analysis will be demonstrated. We describe how the dominator set is computed for line 6, by looking at how the dominator analysis must react upon variable assignments during control flow.

In the above example, the CFG of the method **doAssign** is the target of the analysis. The initial analysis information to **doAssign** is empty, because no assignments precede from constructors or initializations. The input fact at line 3 is then empty, $DS_\circ(3) = \{\}$. The assignment in line 3 could therefore potentially add the value of `o3` to the dominator set. However, $LS(3) = \{\}$ and nothing is added to the dominator set. The reason is that the dominator set shall only express the values of variables that have been assigned within a region protected by at least one lock. The analysis then proceeds with the empty dominator set, $DS_\bullet(3) = \{\}$, which is unaffected by line 4, thus $DS_\bullet(4) = \{\}$. At line 5 however, $LS(5) = \{(1,1)\}$ and therefore the assignment to `o2` must now be added to the dominator set, such that it becomes $DS_\bullet(5) = \{(4, \{1\})\}$, which represents that after line 5, the variable `o1` has been assigned to under the lock `this`. The dominator set present at line 6 is then $DS_\circ(6) = \{(4, \{1\})\}$, which ends this simple example.

3.5.1.1 Transfer Function

We will formalize the transfer function of variable assignments according to the behavior presented in the previous section. This can be reviewed in table 3.9 below.

3.5.2 Method Invocation

The dominator analysis must also be aware of method invocations. For invocations to non-final, non-private methods within the class of interest, the over-approximation

$ins(\ell)$	$DS_\bullet(\ell) = f_\ell(DS_\circ(\ell))$
PUTFIELD	$[Given : s = cpindex(ins(\ell))]$ $t =$ this : $f_\ell(DS_\circ(\ell)) = DS_\circ(\ell) \cup \{(s, l) \mid l = dlocks(LS(\ell)) \neq \emptyset\}$ $otherwise$: $f_\ell(DS_\circ(\ell)) = DS_\circ(\ell)$
ISTORE LSTORE FSTORE DSTORE ASTORE	$[Given : s = lindex(ins(\ell))]$ $t =$ this : $f_\ell(DS_\circ(\ell)) = DS_\circ(\ell) \cup \{(s, l) \mid l = dlocks(LS(\ell)) \neq \emptyset\}$ $otherwise$: $f_\ell(DS_\circ(\ell)) = DS_\circ(\ell)$

Table 3.9: Transfer functions for variable assignments in the dominator analysis.

of the behavior regarding the dominator analysis, is that nothing will be added the dominator set on such a dispatch, because the behavior of these methods cannot be known at compile-time. This also applies for dispatches to methods outside the class of interest. However, for dispatches to private or final methods within the class of interest, the behavior is deterministic and the dominator analysis may analyze these dispatches as they occur in the traversal, with a context sensitive approach, where the context information is the points-to set, the lock set and the dominator set at the location of the dispatch. This behavior will be demonstrated in the example in listing 3.17.

```
1  public class DominatorDispatchExample {
2    private Object o1 = new Object();      /* o1=2, new Object()=3 */
3    public void doDispatch() {
4      synchronized(this) {                 /* this=1 */
5        doAssignment();
6        /* etc. */
7      }
8    }
9    public final void doAssignment() {
10     o1 = new Object();                   /* new Object()=4 */
11   }
12 }
```

Listing 3.17: An example with a dispatch to a final method, that must be handled by the dominator analysis.

The dominator set after line 5 of the Java source must reflect that `o1` has been assigned to under the lock of `this`. The analysis handles this, by identifying that a dispatch to the final method within the class of interest, `doAssignment`, is invoked in line (5). Then it initiates an analysis of `doAssignment` with $\iota = DS_\circ(5)$ and performs the analysis of `doAssignment`. As the assignment to `o1` in line 10 occurs, the dominator set result is then updated to $DS = \{(2, \{1\})\}$, representing that `o1` is

assigned to under the lock of `this`. The dominator sets at the end points of the CFG of `doAssignment` are then combined according to $\sqcup_{DA}$; in this case there is only one end point of the CFG, so the computed result is the output dominator set of line 5, namely $DS_{\bullet}(5) = \{(2, \{1\})\}$.

3.5.2.1 Transfer Function

We now formalize the dominator analysis transfer function that handles dispatches. For this purpose, we define the function $\mathcal{R}_{DA}(m, DS_{\circ}(\ell))$, which is a function that initiates a dominator analysis and analyzes the dispatch on location ℓ with the initial analysis information, $\iota = DS_{\circ}(\ell)$, combines all return facts according to $\sqcup_{DA}$ and returns the resulting dominator set. The formalization is shown in Table 3.10.

$ins(\ell)$	$DS_{\bullet}(\ell) = f_{\ell}(DS_{\circ}(\ell))$
All	$[Given : DS_{\circ}(\ell) = \mathcal{P}(S \times L)]$ $DS_{\bullet}(\ell) = \{(s, l) \mid \forall s \in S, \forall l \in DS_{\circ}(\ell)(s) : l \in LS(\ell)\}$

Table 3.10: Transfer function for dispatches in the dominator analysis.

3.5.3 Removing Dominating Assignments

What remains to describe for the dominator analysis, is that values representing variables that have been assigned to within a specific locked scope, must be removed again from the dominator set, if the locked scope a value is assigned in ends.

```
1 public class DominatorRemoveEntryExample {
2   public void doSynchronizedAssignments() {
3     Object o1;                /* o1=2 */
4     synchronized(this) {      /* this=1 */
5       o1 = new Object();      /* new Object()=3 */
6     }
7     /* etc. */
8   }
9 }
```

Listing 3.18: In the example the point of interest is when the synchronized region ends, which must influences the dominator set.

Consider line 7 in listing 3.18. Here the synchronization on `this` has ended, and therefore interleavings may exist between the assignment to `o1` and line 7, and `o1` may not be present with an assignment under the lock `this` in the dominator set in line 7. The dominator analysis handles this as it visits locations. It looks up the lockset at the location it visits, $LS(\ell)$, and if any dominating assignments are present

in the dominator set with any locks not present in $LS(\ell)$, then these assignments must be removed from the set. In the example, the lock set in line 7 does not contain any locks. However, the input dominator set contains an assignment to `o1` under the `this` lock. Then $l \setminus LS(7) = \{1\} \setminus \{\} \neq \emptyset$, as $DS_\circ(7)(2) = l = \{1\}$, and $(2, 1)$ must be removed from the dominator set, yielding an empty dominator set in line 7.

3.5.3.1 Transfer Functions

We can formalize the behavior just described in a transfer function that applies to all locations, without being a function of the instruction, as it only relies on the incoming dominator set and the lockset on the location. This can be seen in table 3.11.

$ins(\ell)$	$DS_\bullet(\ell) = f_\ell(DS_\circ(\ell))$
All	$[Given : DS_\circ(\ell) = \mathcal{P}(S \times L)]$ $DS_\bullet(\ell) =$ $\{(s, l) \mid \forall s \in S, \forall l \in DS_\circ(\ell)(s) : l \in LS(\ell)\}$

Table 3.11: Transfer function for removing dominating assignments in the dominator analysis.

3.5.4 Summary

In the previous sections, it is documented how the dominator analysis behaves and transfer functions defining this behavior are stated. We now sum these up in the Table 3.12.

$ins(\ell)$	$DS_{\bullet}(\ell) = f_{\ell}(DS_{\circ}(\ell))$
PUTFIELD	$[Given : s = cpindex(ins(\ell))]$ $t =$ `this` : $f_{\ell}(DS_{\circ}(\ell)) = DS_{\circ}(\ell) \cup \{(s,l) \mid l = dlocks(LS(\ell)) \neq \emptyset\}$ $otherwise$: $f_{\ell}(DS_{\circ}(\ell)) = DS_{\circ}(\ell)$
ISTORE LSTORE FSTORE DSTORE ASTORE	$[Given : s = lindex(ins(\ell))]$ $t =$ `this` : $f_{\ell}(DS_{\circ}(\ell)) = DS_{\circ}(\ell) \cup \{(s,l) \mid l = dlocks(LS(\ell)) \neq \emptyset\}$ $otherwise$: $f_{\ell}(DS_{\circ}(\ell)) = DS_{\circ}(\ell)$
INVOKEINTERFACE INVOKESPECIAL INVOKEVIRTUAL	$[Given : m = method(ins(\ell)), t = target(ins(\ell))]$ $t =$ `this` $\wedge\, m$ is `private` $\vee$ `final` : $f_{\ell}(DS_{\circ}(\ell)) = \mathcal{R}_{DA}(m, DS_{\circ}(\ell))$ $otherwise$: $f_{\ell}(DS_{\circ}(\ell)) = DS_{\circ}(\ell)$
All	$[Given : DS_{\circ}(\ell) = \mathcal{P}(S \times L)]$ $DS_{\bullet}(\ell) = \{(s,l) \mid \forall s \in S, \forall l \in DS_{\circ}(\ell)(s) : l \in LS(\ell)\}$

Table 3.12: Transfer functions for the dominator analysis.

3.6 Concurrent Points-To Analysis

The concurrent points-to analysis is responsible of collecting points-to sets from other location in the class of interest, such that contains information of what variables point to, when considering use in a multi-threaded environment. This analysis is flow-insensitive, as the computation of the current fact does not depend on the outcome of previously computed facts. The concurrent points-to analysis does not propagate the fact through control flow; it bases the fact only on the facts computed by previous analyses at locations in the class. We define the fact for the concurrent points-to analysis:

$$CPTS(\ell_{C,M,n}) = \langle S, D \rangle$$

called the concurrent points-to set fact. It is a 2-tuple, similar to the $PTS(\ell)$, expressing that for each source $s_i \in S$, $d_i \in D$ is a value that the source points to. Several sources may be the same, which represents that the same variable may point to several values.

Until this point, the analyses described have been intra-procedural. The concurrent points-to analysis, however, is inter-procedural. This is because threads may be present and executing at any location in the class of interest, making the possible set of values variables may point to at a given location naturally bigger, than for

sequential program flow.

As the concurrent points-to analysis does not have any flow-functions, because the facts are independent of program flow, we do not either have a transfer function, as such. The only function we shall define in this section, is $CPTS(\ell_{C,M,n})$.

The concurrent points-to analysis uses the locksets for locations obtained from the lock analysis. The concurrent points-to analysis should be able to compare two such locksets and determine, whether the region locked by one lockset will be able to execute in parallel with a region locked by another lockset. For this, we introduce the function $enter(LS_1, LS_2) = b$ where $b \in \{true, false\}$. Before formally defining this function, we introduce two functions, $rl(l) = c$ and $wl(l) = c$, that yield a lock c if the lock l is a readlock or writelock respectively. Note that c may be $\emptyset$, which then means that the functions does not find l to be a read- or writelock respectively. For the locksets $LS_1 = \mathcal{P}(L_1 \times N_1)$ and $LS_2 = \mathcal{P}(L_2 \times N_2)$ we define the function:

$$enters(LS_1, LS_2) =$$

$$\forall l_1 \in L_1, \forall l_2 \in L_2 : \bigwedge \begin{cases} wl(l_2) \neq c & \text{if } rl(l_1) = c, c \neq \emptyset \\ rl(l_2) \neq c \wedge wl(l_2) \neq c & \text{if } wl(l_1) = c, c \neq \emptyset \\ wl(l_1) \neq c & \text{if } rl(l_2) = c, c \neq \emptyset \\ rl(l_1) \neq c \wedge wl(l_1) \neq c & \text{if } wl(l_1) = c, c \neq \emptyset \\ l_1 \neq l_2 & \text{if otherwise} \end{cases}$$

where $\bigwedge$ is the logical AND for all the expressions in $enter(LS_1, LS_2)$.

In the following we describe the behavior of the concurrent points-to analysis, initially from an example. Later, we shall formalize the function that computes the concurrent points-to set for a specific location in the class of interest.

In Listing 3.19, we describe the behavior of the concurrent points-to analysis, if it was queried what variables may point to in line 10 of the Java source code.

Initially, what variables can definitely point to at the location, is the fact from the points-to analysis at line 10. We assume that `null` has the value 0. The points-to set in line 10, is $PTS(10) = \{(2,3), (4,7), (5,6)\}$. The analysis will then traverse all locations in public methods in the class, to be able to reveal what other assignments to global variables there might be. Line 7 and line 10 will then be visited, however trivially they will not add anything not already present. The analysis then visits location 16, 17 and 18 in turn (at some time). On these locations, the analysis must initially check that the current locks held (at line 10), $LS(10)$, allow entering with the locks held in line 16, 17 and 18 respectively, in short: $enters(LS(10), LS(16-18))$. If it yields $false$, then the regions are mutual exclusive, and no threads may execute from line 16-18 while currently at line 10. In this case, $LS(10) = \{(2,1)\}$ and $LS(16-18) = \{(2,1)\}$ intersect and therefore $enters(LS(10), LS(16-18)) = true$ and none of the assignments in line 16-18 are added at this point. However, the assignment to `o3` in line 16 does actually have to be included. We now explain why and how the analysis proceeds to obtain that.

```
1  public void ConcurrentPointsToExample {
2    private final Object o1 = new Object(); /* o1=2, new Object()=3 */
3    private Object o2;                      /* o2=4 */
4    private Object o3;                      /* o3=5 */
5
6    public void doAssign1() {
7      o3 = new Object();                    /* new Object()=6 */
8      synchronized(o1) {
9        o2 = new Object();                  /* new Object()=7 */
10       /* etc */
11     }
12   }
13
14   public void doAssign2() {
15     synchronized(o1) {
16       o3 = new Object();                  /* new Object()=8 */
17       o2 = new Object();                  /* new Object()=9 */
18       o2 = new Object();                  /* new Object()=10 */
19     }
20   }
21 }
```

Listing 3.19: An example class demonstrating examples of what the concurrent points-to analysis must analyze.

In line 7 `o3` is assigned a value, however not within a locked scope. Therefore an interleaving may exist before the locked scope from line 8 is entered, such that a thread then executes line 15-19 and thereby `o3` may also point to another value. However, `o2` is also assigned in that interleaving, but it should not have any other values added, because in line 9 it is assigned, and that is within the same locked region as line 10 and therefore dominates the assignments to `o2` in lines 17 and 18. The keypoint to make our analysis aware of this, is the information from the dominator analysis, $DS(10) = \{4, \{2\}\}$, which indicates that the value of `o2` from $PTS(10)$ is assigned under the locked scope of `o1` (value 2). For further information, see Listing 3.16 and the corresponding description of the computation of $DS(10)$. What the analysis does, is that it compares the locksets from all locations to the lockset at the current location, and if any points-to information is in the points-to set that is not in the dominator set, then that points-to information is added. In this case, that means that at line 19, where $PTS(19) = \{(2,3),(5,8),(4,10)\}$, the locksets are not mutually exclusive, and as $5 \notin DS(10)$, the points-to $(5,8)$ is added (and $(2,3)$ as well, but that is already present). Thereby the concurrent points-to set at line 10 becomes $CPTS(10) = \{(2,3),(4,6),(5,7),(5,10)\}$.

For the concurrent points-to analysis, it visits all locations within non-`private` methods, because these are the locations that are immediately reachable for an SPA, that may invoke these methods. However, in case a non-`private` method invokes a `private` method within the class of interest, the locations within that `private` dispatch will also have to be visited. This approach offers better precision, than just visiting all methods within the class of interest.

We then define the overall computation of the concurrent points-to set at a given location. For convenience, we define the set P of methods the concurrent points-to analysis must visit all locations of, initially the non-`private` methods in the class of interest, C. We allow this set to be expanded, when dispatches to private methods, $m_{priv} \notin P \wedge m_{priv} \in C$, are seen during the visits of locations, such that these methods are visited, before the computation ends:

$$CPTS(\ell_{C,M,n}) = PTS(\ell) \cup \left\{ \begin{array}{l} \forall \ell_{C,m_i,j}, \forall m_i \in P, \\ enters(LS(\ell_{C,M,n}), LS(\ell_{C,m_i,j})) = true, \\ s = target(ins(\ell_{C,m_i,j})) \end{array} \right| \bigcup \left(\begin{array}{ll} \{(s,d_j) | \forall d_j \in PTS(\ell_{C,m_i,j})(s)\} & ins(\ell_{C,m_i,j}) \in \{\texttt{PUTFIELD}, \texttt{PUTSTATIC}\} \\ & \vee\ (DS(\ell_{C,M,n})(s) = \emptyset\ \wedge \\ & \neg(ins(\ell_{C,m_i,j}) = \texttt{INVOKEINSTRUCTION} \\ & \wedge\ s \in m_{priv}, m_{priv} \notin P \wedge m_{priv} \in C)) \\ & \\ M = M \cup m_{priv} & ins(\ell_{C,m_i,j}) = \texttt{INVOKEINSTRUCTION} \\ & \wedge\ s \in m_{priv}, m_{priv} \notin P \wedge m_{priv} \in C \end{array} \right) \Bigg\}$$

3.7 Applying the Analyses

Based on the analyses we have now described, we will now return to the conditions of thread safety and describe how each aspect of these definitions may be tested with the help of the analyses. We take each condition in turn and describe the required use of the analyses to determine thread-safety violations with respect to SPA.

3.7.1 Encapsulation

First of all, violations of thread-safety regarding encapsulation can be identified if any fields in the class of interest are declared `public` or `protected` without being declared `final`. This does not require any of the analyses described in the precedings of this section, to test. Simply iterate the field variables in the bytecode and test their visibility and immutability modifier against the criteria.

For a more precise analysis, the mutability of objects in the field variables of the class has to be determined. Depending on the mutability of an object in a field variable, it may only be declared `private` if the object is mutable and non-`private` in addition, if it is immutable but also declared `final`.

3.7.2 Absence of Deadlock

For mutually exclusive locks, we have the definition of a potential deadlock from (2.1):

$$ls(l_1) \cap ls(l_2) = \emptyset \wedge ls(l_1)' \cap ls(l_2) \neq \emptyset \wedge ls(l_2)' \cap ls(l_1) \neq \emptyset$$

This definition requires the information about which locks are held at and after a particular location. This information is offered by the lock analysis as $LS_\circ(\ell)$ for the locks *at* ℓ and $LS_\bullet(\ell)$ for the locks held *after* ℓ. Checking for potential deadlocks is then as simple as comparing all combinations of $(LS_\circ(\ell), LS_\bullet(\ell)) \times (LS_\circ(\ell), LS_\bullet(\ell))$ to check if any fulfill the condition in (2.1), meaning a potential deadlock exists. The check can even be substantially improved, by only traversing the locations at which a lock is taken, -this is how the implementation works.

The check above also applies to writer-locks that may be a part of the locksets. However, for reader- and writer-locks, other situations of deadlocking may occur. This is formalized in (2.2), that says:

$$\begin{array}{ll} (rl(L), > 0) \in LS_\circ(\ell) \wedge (wl(L), 1) \in LS_\bullet(\ell) & \vee \\ (wl(L), 1) \in LS_\circ(\ell) \wedge (rl(L), > 0) \in LS_\bullet(\ell) & \vee \\ (wl(L), > 1) \in LS_\circ(\ell) & \end{array}$$

The above is tested in a similar manner, by only looking at locations where locks are being acquired.

The final case of deadlock is caused by bad encapsulation, namely that an object used as lock is not declared private. In that case, an SPA may hold the lock in another class, i.e., causing a deadlock or spinning a loop forever with that lock held. This property can be verified by running through all locations in the class of interest and querying the locksets if a lock held at a location is not declared private. In fact this property can be determined already as the lock analysis is performed, however this is left for as an implementation detail.

3.7.3 Escaped Objects

To analyze whether objects influencing the state of the class of interest may escape, so that SPA undesiredly may alter the state of the class of interest, the concurrent points-to analysis can be put to use. For all locations where `public` or `protected` methods within the class returns an object, the concurrent points-to set at that location should be queried to reveal if any objects influencing the state of the class of interest are pointed to by any of the returned objects. Also, for all invocations of methods outside the class of interest, parameters should not point to any state object references, which may also be checked with the concurrent points-to sets at

the locations of dispatches. However, it remains to approximate the objects that may have influence on the state of the class of interest.

The trivial objects that influence the state of the class of interest, are the field variables. But field variables may themselves contain a state, e.g., a `java.util.collections.ArrayLists` state depends on the elements of the list, which may also be returned by methods in the class of interest, thus escaping objects with influence on the state. Approximating this behavior and displaying proper warnings, the muteability of the objects in the field variables must somehow be determined. One way to do so, could be to traverse invocations on the field objects into the class where the method is defined and analyze if that invocation may lead to a change of the state in that object. Doing so, requires an analysis of it self and somewhat does not comply to the class-wise analysis approach, we would like to delimit the analyses to.

Another approximation could be to interpret all return values from methods not within the class of interest, as potentially returning escaped values. Likewise, an approximation regarding all parameters passed to methods outside the class of interest, could be to assume these have escaped.

The latter approach would be the easiest to apply and compute, however, at the cost of much lower precision. For methods within the class of interest that are possible to override, that is, not declared `final` and `public` or `private`, the latter approach though is the best approximation one can get, because the SPA may override such methods and return whatever, possibly escaping the state of the class of interest.

Finally, if the availability of an analysis computing some approximation of where escaped values enter and exit the class of interest is assumed, then violations of thread-safety regarding escaping objects, would be on occurrences where the concurrent points-to analysis reveals that a write to a member of the state takes place, where the value that is written is an escaped value, and the occurrence where a member of the state is passed as argument to an invocation outside the class or a return from a non-`private` method within the class. For `final` and private methods within the class, context-sensitivity could be applied to the analysis, such that the arguments could be followed in the control flow and it could be determined if they escape.

3.7.4 Locking

All locks acquired within a non-`private` method within the class of interest, must be released again before the dispatch returns. This property can be analyzed from the lockset entry and exit facts of all non-private methods within the class of interest. The exit fact may not hold more locks than the entry fact holds.

However, another condition applies to the locks in the class of interest, namely that they may not be changed at any program point in the class. In case a lock changed in the analysis, it will appear from the concurrent points-to analysis, and therefore a test

whether the assumption holds can be performed with the support of the concurrent points-to analysis. Furthermore, locks may not be an escaped object, which with the support of a capable escape analysis can be detected.

3.7.5 Thread-safe Field Access

To analyze how field variables are used in accordance with thread-safety, a checker should look at all `PUTFIELD` and `GETFIELD` instructions, and if the use is not in correspondence with the statement

> *All writes to a field variable may not give occasion to non-deterministic results possibly being read elsewhere in the class. Neither must multiple writes to the same field variable take place at any time.*

a violation has occured. In practice, violations can be expressed formaly. Assume $P(x)$ is a `PUTFIELD` operation to field x and $G(x)$ is a `GETFIELD` operation on the field x, and we write $\ell_i = P(x)$ to express that the location ℓ_i is a write operation to x, and similar for reads. Then violations can be expressed:

$$\exists \ell_i = P(x) : LS(\ell_j) \cap LS(\ell_i) = \emptyset, \ell_j = P(x) \vee \ell_j = G(x)$$

This formalization can be applied in practice and will reveal violations of thread-safety regarding field accesses.

3.7.6 Stale Data

For stale data, the check for thread-safe field access will find all occurences, where stale data might occur also, given that the check also looks at assignments in constructors. However, a special case for constructors, is that if the field assigned to is declared `final` or `volatile`, then stale data is guaranteed not to occur with that constructor.

Chapter 4

Implementation

In this Section we discuss the implementation of the analyses, the detectors and finally how these are integrated in FindBugs. As described in Section 2.3, FindBugs utilizes both BCEL and ASM to provide the developer with the best from both worlds. In order to ease the implementation FindBugs provides a feature rich API which to some extend works as a layer above the BCEL and ASM API's, uniting the API's. FindBugs provides an intra-procedural control flow analysis that may be used to construct a CFG for a given method. The CFG's provide the foundation for the analyses that we now describe the implementation of.

4.1 The Analyses

In the following we describe the implementation of the analyses formalized in section 3. As mentioned FindBugs provides a number of base classes useful for creating flow sensitive analyses. Because our analyses have many things in common we have created the base class `AGenericForwardDataflowAnalysis` which can be found in the source code[1]. The `AGenericForwardDataflowAnalysis` is an abstract class that extends the `ForwardDataflowAnalysis` which is a part of the FindBugs framework. The primary goal of the intermidiate `AGenericForwardDataflowAnalysis` class, is to provide a transparent way for handling dispatches, as we want to be context sensitive for `private` and `final` methods within the context of the class.

[1] dtu.imm.findbugs.plugin.analysis.AGenericForwardDataflowAnalysis.java

In order to provide a nice abstration, we make extensive use of generics, thus the `ForwardDataflowAnalysis` is a generic class, which let us define the type of the fact that we want to use in the analysis. There are no constraints regarding the type of the fact, meaning that the fact does not need to be a subclass of some common fact class. This makes it possible for us to create our own common base class for the fact in each of our analyses. All facts implement the common interface shown in Listing 4.1, which makes it possible to handle all the functionality, that these methods provide, in the `AGenericForwardDataflowAnalysis`, simplifying the implementation of the analyses.

An analysis class extending `AGenericForwardDataflowAnalysis` is forced to implement methods for transfering a fact from one location to the next, as well as creating a new dataflow for a context sensitive dispatch.

```
1 public interface IGenericFact<Fact> {
2   public void copyFrom(Fact other);
3   public Fact makeCopy();
4   public boolean isTop();
5   public boolean sameAs(Fact other);
6   public void meetWith(Fact other);
7   public void clear();
8 }
```

Listing 4.1: The generic interface that the `PointsToSet`, `ExtendedLockset` and `Dominatorset` implements.

As it can be seen from the class header for the `AGenericForwardDataflowAnalysis` shown in Listing 4.2, we force facts to implement our `IGenericFact<Fact>`. As one may note we give filenames a preceding letter denoting if the file contains a (`I`)nterface or an (`A`)bstract class.

```
1 public abstract class AGenericForwardDataflowAnalysis
2 <
3   Fact extends IGenericFact<Fact>,
4   Dataflow extends AGenericDataflow<Fact, ?, ?>
5 >
6 extends ForwardDataflowAnalysis<Fact> { ...
```

Listing 4.2: The class header for the `AGenericForwardDataflowAnalysis`, which is our base class for all our analyses.

A generic dataflow base class extending the `Dataflow` provided by FindBugs, has also been developed to generalize the basic functionality that `AGenericForwardDataflowAnalysis` provides. The class header for this class, `AGenericDataflow`, is shown in Listing 4.3

The interface `IGenericFact<Fact>`, and the two classes, `AGenericDataflow` and

```
1 public abstract class AGenericDataflow
2 <
3   Fact extends IGenericFact<Fact>,
4   Analysis extends AGenericForwardDataflowAnalysis<Fact,Dataflow>,
5   Dataflow extends AGenericDataflow<Fact, ?, ?>
6 >
7 extends edu.umd.cs.findbugs.ba.Dataflow<Fact, Analysis> {
```

Listing 4.3: The class header for the `AGenericDataflow` class, which is our base class for all our dataflows.

`AGenericForwardDataflowAnalysis`, together form the foundation for our analyses. Figure 4.1 shows how our points-to analysis inherit these base classes.

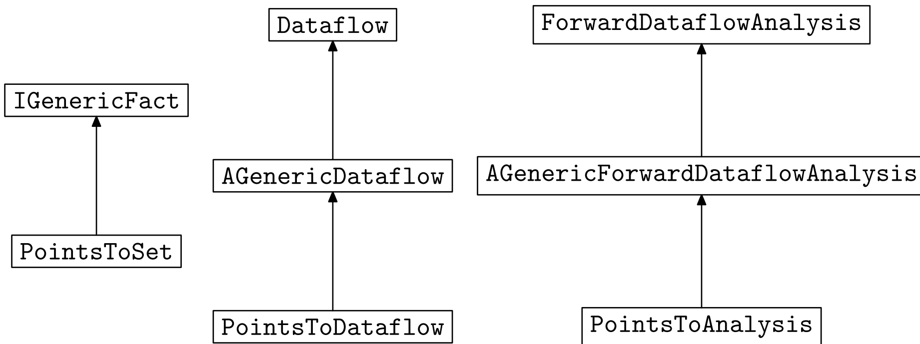

Figure 4.1: The figure illustrates how the points-to analysis inherit form our foundation classes

As mentioned the `AGenericForwardDataflowAnalysis` handles context sensitive method invocations, almost transparently for the subclasses. This is achieved by forcing subclasses to implement the abstract method in Listing 4.4, which returns a new dataflow using the initial analysis information given by the fact at the location of the method invocation.

```
1 protected abstract Dataflow getDispatchDataflow(ClassContext classContext,
2                                                 Method method,
3                                                 Location location,
4                                                 InvokeInstruction ins,
5                                                 Fact entryFact)
```

Listing 4.4: Subclasses of `GenericForwardDataflowAnalysis` is forced to implement this method, so that dispatches can be handled in the common `AGenericForwardDataflowAnalysis` class.

4.1.1 Points-to analysis

In the following section we describe the details concerning the implementation of the points-to analysis. The `PointsToAnalysis`[2] inherit from the common analysis base class, `GenericForwardDataflowAnalysis`. Likewise does the `PointsToDataflow`[3] inherit from `AGenericDataflow` and the `PointsToSet`[4] implements the common fact interface shown in Listing 4.1, where the `meetWith` method is a direct translation of the formal meet operator defined in Section 3.3, see Listing 4.5.

```
1  public void meetWith(PointsToSet other) {
2    for (ValueNumber vn : other.pointsToSet.keySet()) {
3      if (this.pointsToSet.containsKey(vn)) {
4        this.pointsToSet.get(vn).addAll(other.pointsToSet.get(vn));
5      }
6      else {
7        this.pointsToSet.put(vn, other.pointsToSet.get(vn));
8      }
9    }
10 }
```

Listing 4.5: The implementation of the meet function is a direct translation of the formal definition expressed in Section 3.3

With FindBugs comes a intra-procedural `ValueNumberDataflowAnalysis` that may be used to model the production of values in a stack frame. The analysis is context insensitive, and therefore over approximates return values. The analysis uses instances of the class `ValueNumber` to represent information, a `ValueNumber` basically consists of a number and flags, e.g., a number may have the flag `RETURN_VALUE`, indicating that this `ValueNumber` number entered the stackframe as a return value from a method invocation. A `ValueNumber` cannot be instantiated, but have to be instantiated by a `ValueNumberFactory` where instances of `ValueNumbers` produced by the same `ValueNumberFactory` are unique, so reference equality may be used to determine whether or not two value numbers are the same. In general, `ValueNumbers` from different factories cannot be compared, thus we cannot use `ValueNumber`'s produced by the `ValueNumberDataflowAnalysis` in our points-to analysis, because we want to be able to compare points-to information across methods. To solve this problem we use a single instance of a `ValueNumberFactory` when analyzing a class, thereby mapping values from a instance of a `ValueNumberDataflowAnalysis` into our `ValueNumber`-*space*.

Remember that points-to information at a specific location, ℓ in Section 3.3 was defined as a 2-tuple, where a source, s, could point to multiple destinations d_n.

[2]Implemented in: *dtu.imm.findbugs.plugin.analysis.pta.PointsToAnalysis*
[3]Implemented in: *dtu.imm.findbugs.plugin.analysis.pta.PointsToDataflow*
[4]Implemented in: *dtu.imm.findbugs.plugin.analysis.pta.PointsToSet*

Implementation wise this is achieved by using the structure:

```
HashMap<ValueNumber, Set<ValueNumber>>
```

where the *key* is the source, pointing to a set of destinations.
Because the `PointsToAnalysis` is a subclass of `GenericForwardDataflowAnalysis` it implements the `getDispatchDataflow` method. In order to create a new dispatch dataflow for an invoked method, arguments are be popped from the stack, mapped into our `ValueNumber`-*space*. and passed to the constructor of the new `GenericForwardDataflowAnalysis`. Because the arguments simply are sources in the initial points-to set, the new analysis knows what the arguments points to.
Finally when a fact is transfered from one location to the next, then points-to information in the fact is first modified, and then φ-values are introduced where necessary. Both actions conforms to the formal definition found in Section 3.3.

4.1.2 Lock analysis

We now turn our attention to the implementation of the lock analysis. As for the points-to analysis the lock analysis inherit our three foundation classes.
The `LockAnalysis`[5] inherit from the common `GenericForwardDataflowAnalysis`, `LockDataflow`[6] inherit from `AGenericDataflow` and finally does `ExtendedLockSet`[7] implement the common fact interface shown in Listing 4.1 as well as inheriting from a generel `LockSet` class provided by FindBugs. The `LockSet` class in FindBugs is used in a simple intra-procedural lock analysis, which seems very unprecice, however we found the `LockSet` class useful. The `meetWith` method for the `LockSet` conforms to the formal definition expressed in Section 3.4. As for the `PointsToAnalysis` the `LockAnalysis` implements the `getDispatchDataflow` method, however no arguments are popped from the stack, because only the initial lock information must be parsed to the new `LockAnalysis`.

The transfer function for the `LockAnalysis` detects when a `lock` or `unlock` method is invoked on subclasses on `java.util.concurrent.locks.Lock` as well as detecting `MONITORENTER` and `MONITOREXIT` instructions. When that happens the `LockAnalysis` uses the `PointsToAnalysis` to resolve the target that the lock is acquired on, maps the target to a monitor or a lock `ValueNumber` instance, and either increases or decreases the lock count for that specific lock target. Finally the lock analysis detects method calls to `readLock` and `writeLock` on instances of `ReentrantReadWriteLock`, and flag the return value with respectively `READ_LOCK` or `WRITE_LOCK`. This makes it possible for later analyses to detect if a region is locked by a read or write lock.

[5]Implemented in: *dtu.imm.findbugs.plugin.analysis.la.LockAnalysis*
[6]Implemented in: *dtu.imm.findbugs.plugin.analysis.la.LockDataflow*
[7]Implemented in: *dtu.imm.findbugs.plugin.analysis.la.ExtendedLockSet*

4.1.3 Dominator analysis

The dominator analysis is, as the other analyses, based on our three foundation classes. The analysis is implemented in the `DominatorAnalysis`[8] class which inherit from `GenericForwardDataflowAnalysis`. The `DominatorDataflow`[9] inherit from `AGenericDataflow` and the `DominatorSet`[10] implements the common fact interface shown in Listing 4.1.

The `meetWith` method for the `DominatorSet` conforms to the formal definition expressed in Section 3.5. Regarding the implementation, a given variable may be dominated by multiple locks and therefore we use the data structure

```
HashMap<ValueNumber, Set<ValueNumber>>
```

to represent dominater information at a given location. The transfer function for the `DominatorAnalysis` detects when a `PUTFIELD` instruction occurs. If there are locks held at that location, the fact is updated, indicating that the given field is assigned under that given lock set. When locks are released or get the value $\perp$, they are removed from the dominator set. Note that the `DominatorAnalysis` depends on both the `PointsToAnalysis` to map the instruction into our `ValueNumber`-*space*, as well as the `LockAnalysis` which provides the lock information. As for the other analyses the `DominatorAnalysis` is forced to implement the `getDispatchDataflow` method, which simply returns a new `DominatorDataflow`, where the dominator information at the location of the method invocation is used as initial information in the new `DominatorAnalysis`.

4.1.4 Concurrent points-to analysis

We now turn our attention to the concurent points-to analysis the uses the three other analyses to provide a interprocedural points-to analysis that take locks into account. The analysis is implemented in `ConcurrentPointsToAnalysis`[11] which inherit from the common analysis base class, `GenericForwardDataflowAnalysis`. Note that even though the concurrent points-to analysis is not flow sensitive, we still get an easy way of traversing program locations by extending our foundation classes. The `ConcurrentPointsToDataflow`[12] inherit from `AGenericDataflow` and the fact used in this analysis is the same as used in the `PointsToAnalysis`, namely the `PointsToSet` that implements the common fact interface listed in 4.1.

Points-to information at a given location is computed by first copying the content of the points set from the sequential points-to analysis into the points-to set for

[8]Implemented in: *dtu.imm.findbugs.plugin.analysis.da.DominatorAnalysis*
[9]Implemented in: *dtu.imm.findbugs.plugin.analysis.da.DominatorDataflow*
[10]Implemented in: *dtu.imm.findbugs.plugin.analysis.da.DominatorSet*
[11]Implemented in: *dtu.imm.findbugs.plugin.analysis.cpta.ConcurrentPointsToAnalysis*
[12]Implemented in: *dtu.imm.findbugs.plugin.analysis.cpta.ConcurrentPointsToDataflow*

the concurrent points-to analysis. Then sequential points-to information for all other locations where another thread could be simultaneously and where a write to a field is done, are added to the current fact. Furthermore is points-to information at locations where another thread could be simultaneously and where a lock is released, added. Note that if a φ-value is added, the whole points-to chain is added.

4.2 Detecting Bugs

One of our requirements is to make it easy for developers to utilize our plugin. As Java 1.5 provides the ability to annotate code, we use these to enable the developer to provide information for the analyses. At the time of writing BCEL does not include any support for reading annotations in compilled Java classes, but fortunately ASM does.

As FindBugs includes the annotation package from the book *Java concurrency in practice* [12], we have decided to use the `@ThreadSafe` annotation from that package to invoke our bug detectors. Therefore all that one must do, is to annotate a given class `@ThreadSafe`, to get our bug detectors running on the class.

In this section we describe the implementation of the bug detectors that utilize our analysis to find and report bugs. All our bug detectors extends our common detector class `ConcurrentDetector`. The primary goal of the `ConcurrentDetector` is to statically cache the instance of the `PluginAnalysisContext`, which is a class that contains instances of dataflows. We do this to avoid running the analyses once for every bug detector. Furthermore the `ConcurrentDetector` almost makes context sensitive dispatches transparent for the subclass. Because every dataflow may contain zero or more dispatch dataflows, visiting dataflows are done in a recursive manner, as the problem suggests. The bug detectors that we have implemented can be found in the `dtu.imm.findbugs.plugin.detect` package. Below we summerize the functionality of these:

- The `DeadlockDetector` finds Deadlocks by looking at location paris in the class, where the Deadlock condition described in Section 3.7 holds. For each of such paris a bug is reported.

- The `LockCheckDetector` performs basic lock checking to detect if a lock target may point to more than one object instance, if that is the case a bug is reported. Furthermore it detects if a lock is done on a variable that only has local scope, thus it has no effect on the state of the class being analyzed.

- The `LockHeldAtReturnDetector` detects if the combined return lock set from a non-`private` method is different from the initial lock set. If that is the case a bug is reported because such behaviour may violate *liveness* properties.

- The `LockOnNullDetector` is used to detect if a lock is taken on a variable that may point to `null`, thus a bug is reported if the lock target may point to at least one `null` value.
- The `NonPrivateFieldDetector` detects if the state of the class is properly encapsulated, otherwise a bug is reported.
- The `UnsafeFieldAccessDetector` is used to detect if reads and writes from a specific field is guarded by a lockset having at least one common lock. Otherwise a bug is reported. This detector also detects missing `volatile` or `final` declarations.

Finally we turn our attention to the implementation details concerning the integration with FindBugs. In order for a plugin to work with FindBugs the following prerequisets must be met. First of all, FindBugs expects two XML files to be found in the root of the plugin Jar. The `findbugs.xml` contains information about what detectors that exist in the plugin, and where in the package hierachy they exist. The other file, namely the `messages.xml` contains the actual bug descriptions reported by the detectors. For more information on these two XML files and the Apache Ant build script, used for building the plugin, see Appendix A which outlines the directory structure of our project, and Appendix B which contains the `findbugs.xml` and `messages.xml`.

4.3 Testing

In this section we describe how the implemented analyses and bug detectors have been tested to verify that they behave correctly.
Below we summarize the two most used approaches for verifying software:

- **Structural testing**
  This kind of testing is often denoted as "white box testing" or "clear box testing". In a structural test the internal perspective of a system is taken into account, meaning that the test is based on the internal representation of the system. The goal of such a test is to identify all paths through the system and verify that no errors exist on any of these paths. In many cases it will however be to time consuming or difficult to identify all paths.
- **Functional testing**
  Functional testing is often denoted as "black box testing" because only output from the system is verified to see if it conforms to the expected output. This means that only the external perspective of the system is taken into account. The disadvantage of a functional test compared to the structural is that one cannot be sure that all existent paths through the system are tested. Therefore even though a functional test succeeds errors might still exist.

Even though a complete structural test would be more precise, we have decided to settle for a functional test, simply because a structural test would be to time consuming. We have also decided that it is sufficient to only test the bug detectors because they depend on the analyses and thereby test them implicitly.

Our testing framework is based on the unit testing framework, JUnit. To avoid any confusion we will in the following denote our testing framework as a “test suite”. The main reason why we decided to use JUnit, is because it intergrates very nicely with Apache Ant which we use to build and run our plugin with. JUnit also provides the necessary functionality to report errors in a convient way, eg. by making assertions or simply failing with a given error message. Finally JUnit has the ability to generate status reports based on errors found in the test(s).

A number of test cases has been developed to verify that the bug detectors report the right bugs, and for each of the test cases expected output has been defined. The test suite we have implemented verifies that the expected output conforms to the actual bugs reported. The expected output has been embedded into each test case so that all information related to that given test case is contained within the given test file. An example of a test case is shown in figure 4.6

```
1  /*
2  <ExpectedBugs>
3    <BugInstance type="NPF" priority="2" abbrev="MTSE" category="MT_CORRECTNESS">
4      <Class classname="dtu.imm.findbugs.testing.examples.ExposedStateVariables">
5        <SourceLine classname="dtu.imm.findbugs.testing.examples.
             ExposedStateVariables"/>
6      </Class>
7      <Field classname="dtu.imm.findbugs.testing.examples.ExposedStateVariables"
           name="i" signature="I" isStatic="false">
8        <SourceLine classname="dtu.imm.findbugs.testing.examples.
             ExposedStateVariables"/>
9      </Field>
10   </BugInstance>
11 </ExpectedBugs>
12 */
13
14 @ThreadSafe
15 public class ExposedStateVariables {
16
17   public int i = 0;
18
19 }
```

Listing 4.6: An example of a test case. The XML in the comment is the expected output, namely a textural representation of the bug(s) that should be detected when analyzing the class. The expected output is parsed by the testing framework and compared with the computed output.

Our test suite is implemented as a subclass of `TestCase`, a predefined class in JUnit. The `TestCase` class provides a number of methods used to report errors with. Our test suite is implemented as a “fixture”, meaning that a test starts by setting up the

test case, running a number of tests, and finally cleaning up the used resources. The implementation of the test suite is found in: `dtu.imm.findbugs.testing.TestSuite` and the test cases for testing all detectors can be found in Appendix C.

Chapter 5

Conclusion

In previous sections we describe analyses that are the foundation of a tool capable of analyzing Java classes class-wise for thread-safety. The approach is to test if a class specified for concurrent use, will be able to prevent the strongest possible attacker, being the over-approximation of concurrent accesses, from using the class in a manner that may leave the class in a state without liveness properties preserved or with the class state changing unexpectedly.

5.1 Achievements

We have developed analyses capable of analyzing Java classes that use the most common synchronization primitives offered by Java, namely `synchronized` and mechanisms derived from `java.util.concurrent.Lock`, in particular readers-writer locks, that introduce special lock semantics. The analyses offer the foundation of detecting whether the properties introduced in Section 2.6 are violated within a class meant for use in a threaded environment.

A key achievement is the concurrent points-to analysis, which performs an analysis, to determine what variables may point to at a given location. It does so, by identifying what other regions of context in the class, that may be executed in parallel by intersecting the locksets between locations, thus providing another approach to performing a may-happen-in-parallel analysis [19, 23]; traditionally suggested to be based on a *parallel execution graph*.

We have developed detectors for detecting most of the properties introduced in Section 2.6, making our tool capable of detecting many synchronization errors. The detectors that has been developed, are capable of detecting violations of:

- **Encapsulation** We currently detect violations of encapsulation by identifying fields that are `protected` or `public`, and not `final`, which may be directly accessed and written to by SPA. It remains to develop an analysis to check if fields may be mutable objects and thus only may be declared `private`.
- **Absence of Deadlock** Potential deadlocks are successfully identified for all uses of `synchronized`, objects implementing `java.util.concurrent.Lock` and readers-writer locks.
- **Locking** All non-`private` methods that has acquired more locks than they have released, or vice versa, at any return point of the method, are detected. Furthermore, it is detected if a lock is taken on an object that may point to several values, which is classified as an error.
- **Thread-safe Field Access** Race conditions are detected, such that two different locations may execute in parallel, where one writes to the same field as the other reads or writes to.
- **Stale data** Stale data is detected as violations of thread-safe field access. The special case with `final` or `volatile` field variables initialized in a constructor, is handled, so that it does not generate unnecessary errors.

What remains is detection of escaped values, which relies on the development of an escape analysis, which is left as a remainder for future work.

The analysis implementations are based on a generalized platform on which other analyses, such as an escape analysis, can be based on. The generic foundation of the analyses offer an easy way for other analyses to perform context-sensitive analysis regarding invocations of methods with unchangeable context at compile time. It also offers to only change in the general, generic analysis, if, e.g., context-sensitivity is desired to cover entire programs.

Similarly, a generic platform for detectors using our analyses has been developed. This greatly eases the implementation of the remainding detectors, that are left for future work.

The tool we have developed integrates with Eclipse, such that the developer receives bug markings according to violations of thread safety, in classes marked for analysis with the `@ThreadSafe` annotation.

5.2 Applications

With the analyses we have developed, the foundation for a class-wise thread-safety test tool for Java classes has been created. Despite the limitations of our analyses, the detectors implemented covers a range of potential thread-safety violations. Our tool excels as a component based test approach to Java classes that are to be used in a multi-threaded environment. Especially in situations where the program context utilizes a huge amount of different threads and a program-wise analysis would be hard to compute, our class-wise approach proves beneficial, as it assumes any number of concurrent threads utilizing the class and therefore over-approximates any program specific contexts, where the class may be used concurrently.

The generic foundations of both the analyses and detectors implemented, allows for changing the common behavior of our analyses in an easy way. E.g., future work or other applications of the analyses than those we present, may desire more or less precision, which in many cases can be accomplished by changing the generic foundations of the analyses.

5.3 Future work

Some aspects are left for future work. An escape analysis that bases its fact on the concurrent points-to analysis and the remaining detector to be able to analyze where values influencing state may escape the class of interest. Also, the `Semaphore` synchronization primitive is not currently accounted for by the lock analysis, which can be implemented as future work. An improvement on the otherwise generic structure of the implementation, would be to create a generic lock type, such that only a description of how a custom lock mechanism act and interact would be required to support new lock mechanisms in the lock analysis.

Performance has not been a main concern for our work, and therefore future work may involve optimizing computations. Also precision may be improved on some aspects. E.g., future work might involve the implementation of a lazy analysis to determine the mutability of objects in the field variables in the class of interest. Lazy, because it should not be run on all field objects, only those that are declared `final` and are non-`private`.

A comparison between our tool and other projects offering similar efforts is also left for future work, as the time constraints of our work has not permitted such. Though, our tool takes another approach than most other concurrency testing tools for Java, as we analyze class-wise and therefore over-approximates the program-wise context the class of interest may be used in. Therefore our tool may raise warnings that may be guaranteed not to occur by other means at program level, but are correct warnings regarding class-wise thread-safety.

Our testing framework demonstrates the wide range of functionality our tool handles, e.g., the use of both `synchronized` and all subclasses of `java.util.concurrent.locks.Lock`, including special treatment of readers-writer locks. The latter locking mechanisms are even supported to be acquired through invocations of `private` and `final` methods within the class. Our over-approximation of program behavior, does succesfully raise warnings to all violations of class-wise thread-safety, except for the remainding analysis of escaped objects.

List of Notation

$\bigsqcup_A$,$\sqcup_A$ – The combine operators for the analysis type A. See ... 47

$\bot$ – The least element a.k.a. *bottom*. See ... 7

$\sqcap$ – The *meet* operator. See ... 7

$\sqcup$ – The *join* operator. See ... 7

$\top$ – The greatest element a.k.a. *top*. See ... 7

AST – Abstract Syntax Tree. See ... 17

CFG – Control Flow Graph. See ... 6

$cpindex(ins)$ – The index in the constant pool to which the instruction refers (for instructions with an index in the constant pool as operand). See ... 45

$DS(\ell)$ – The fact computed by the dominator analysis called the dominator set. See 68

$DS(\ell)(s) = ls$ – A function that yields ls in the pair $(s, ls) \in DS(\ell)$. See ... 68

E – Edge. See ... 6

f_ℓ – The transfer function at location ℓ.. See ... 46

ι – The initial analysis information.. See ... 46

$ins(\ell)$ – Yields the bytecode instruction at the location ℓ. See ... 46

ℓ – An arbitrary location. See ... 44

ℓ_n – A location in an implicit class and method, where n is the line number in the bytecode instructions sequence for the implicit method.. See ... 44

$\ell_{C,M,n}$ – A location in the bytecode for the method M in class C, where n is the line number.. See ... 44

$\ell_{M,n}$ – A location in an implicit class, where n is the line number in the bytecode instructions sequence for the method M.. See ... 44

L – Lattice. See ... 6

$\ell_\circ$ – The first location of the entry basic-block.. See ... 46

$lindex(ins)$ – The index in the local variable frame into where the instruction indexes (for instructions with an index in the local variable frame as operand). See 45

$lock_{id}(ins(\ell), target(ins(\ell)))$ – A function that given the same target, returns distinct locks for `MONITOR` and `INVOKEINTERFACE` instructions. See 61

$locks$ – A function that given a lockset, returns the set of locks held by that lockset. See 60

$LS(\ell)$ – The fact computed by the points-to analysis called the points-to set. See 60

$ls(\ell_i)$ – The set of locks held *at* the location ℓ_i. See 29

$ls(\ell_i)'$ – The set of locks held *after* the location ℓ_i. See 29

M_i – The ith mutual exclusive lock. See 29

$method(ins)$ – Is the method of the instruction given as parameter (for instructions that has a dispatch target). See 45

φ-values – A single value that points to several other values. See 47

$PTS(\ell)$ – The fact computed by the points-to analysis called the points-to set. See 45

$PTS(\ell)(s_i)$ – A function that yields a set, D_i, containing all destinations, d_{s_i}, s_i points to. See 46

R_i – The readlock for the ith reader-writer-lock. See 29

$\mathcal{R}_A(m, \iota)$ – Denotes the return facts for the analysis A for a method m with initial analysis information ι. See 53

$retval(t, i)$ – A function that yields a unique value corresponding to a pair of target object t and constant pool index i. See 56

SPA – Strongest Possible Attacker. See 2

SSA – Static Single Assignment (Form). See 9

$target(ins)$ – The object that is the *target* of an instruction. In general this is the object into which a value is saved or an invocation is performed on. See 45

$traverse(PTS(\ell), s)$ – traverses the points-to set for all points-to information for the value s and yields a set containing all possible leaf-nodes of the subtree with s as root and in addition, all global variable values in the subtree, that are not dominated by another global variable value in the subtree with s as root. See 51

V – Vertex. See 6

W_i – The writelock for the ith reader-writer-lock. See 29

Bibliography

[1] Apache software license (last visited june 29, 2007). http://www.apache.org/licenses/LICENSE-2.0.

[2] Asm, latest version: 3.0 (november 1, 2006) (last visited june 29, 2007). http://asm.objectweb.org/.

[3] Bcel - the byte code engineering library, latest version: 5.2 (june 6, 2006) (last visited june 29, 2007). http://jakarta.apache.org/bcel/index.html.

[4] Findbugs, latest version: 1.2.1 (may 31, 2007) (last visited june 29, 2007). http://findbugs.sourceforge.net/.

[5] Findbugs, part 1: Improve the quality of your code (last visited june 29, 2007). http://www-128.ibm.com/developerworks/java/library/j-findbug1/.

[6] Findbugs, part 2: Writing custom detectors (last visited june 29, 2007). http://www.ibm.com/developerworks/java/library/j-findbug2/.

[7] Gnu lesser general public license (last visited june 29, 2007). http://www.gnu.org/copyleft/lesser.html.

[8] Soot: a java optimization framework, latest version: 2.2.4 (april 27, 2007) (last visited june 29, 2007). http://www.sable.mcgill.ca/soot/.

[9] Eric Bruneton. Asm 3.0 - a java bytecode engineering library (last visited june 29, 2007). http://download.forge.objectweb.org/asm/asm-guide.pdf, February 2007.

[10] Ciera Nicole Christopher. Evaluating static analysis frameworks (last visited june 29, 2007). http://www.cs.cmu.edu/ aldrich/courses/654/tools/christopher-analysis-frameworks-06.pdf, May 2006.

[11] Create and Read J2SE 5.0 Annotations with the ASM Bytecode Toolkit. *Create and Read J2SE 5.0 Annotations with the ASM Bytecode Toolkit (last visited June 29, 2007)*, October 2004.

[12] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, May 2006.

[13] Michiel Graat. Static analysis of java card applications. Master's thesis, Radboud University Nijmegen, August 2006.

[14] Hatcliff and Dwyer. Using the bandera tool set to model-check properties of concurrent java software, 2001.

[15] Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading, June 2004.

[16] William Hovemeyer, David & Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[17] R. Karol. A tool for analysis of concurrent programs. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2006.

[18] O. Lhotak and L. Hendren. Scaling Java points-to analysis using SPARK. *Lecture Notes in Computer Science*, 2622:153–169, 2003.

[19] Lin Li and Clark Verbrugge. A practical MHP information analysis for concurrent java programs, 2004.

[20] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for java, June 18–19 2001.

[21] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[22] Brad Long, Roger Duke, Doug Goldson, Paul A. Strooper, and Luke Wildman. Mutation-based exploration of a method for verifying concurrent java components, 2004.

[23] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing *mhp* information for concurrent java programs. 1999.

[24] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[25] Bjarne Steensgaard. Points-to analysis in almost linear time, 1996.

[26] Navindra Umanee. Shimple: An investigation of static single assignment form. Master's thesis, McGill University, February 2006.

[27] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework, 1999.

[28] John Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. PhD thesis, Stanford University, March 2007.

[29] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. *Lecture Notes in Computer Science*, 2477:180–??, 2002.

Appendix A

README

```
Installation:

To build the plugin simply go to ./scripts/plugin/ and run 'ant'.
The build.xml script may also be used to test all the detectors by
running 'ant test.all' or to test a single detector,
by running 'ant test.<Name of detector>'
To make the plugin integrate with Eclipse, install the normal
Findbugs plugin and copy the plugin jar to the Findbugs plugin
installation directory.

Directory structure:
./lib/ :
Contains external libraries used.

./scripts/plugin/ :
Contains the XML files used in the plugin.

./external/ :
Contains the implementation of findbugs used in the project.

./src/ :
Contains the source code.
```

Appendix B

FindBugs XML

B.1 messages.xml

```
1  <MessageCollection>
2
3    <Detector class="dtu.imm.findbugs.plugin.detect.DeadlockDetector">
4      <Details><![CDATA[<p> This detector finds deadlock within single
           classes]]>
5      </Details>
6    </Detector>
7
8    <Detector class="dtu.imm.findbugs.plugin.detect.
         NonPrivateFieldDetector">
9      <Details><![CDATA[<p> This detector finds exposed state variables]]>
10     </Details>
11   </Detector>
12
13   <Detector class="dtu.imm.findbugs.plugin.detect.LockOnNullDetector">
14     <Details><![CDATA[<p> This detector finds locations where a lock may
           be taken on a null reference]]>
15     </Details>
16   </Detector>
17
18   <Detector class="dtu.imm.findbugs.plugin.detect.
         LockHeldAtReturnDetector">
19     <Details><![CDATA[<p> This detector finds methods that may result in
           locks being held when returning]]>
20     </Details>
21   </Detector>
```

```
22
23     <Detector class="dtu.imm.findbugs.plugin.detect.LockCheckDetector">
24       <Details><![CDATA[<p> This detector determines if locks may point to
             different tings, as well as locking on local variables]]>
25       </Details>
26     </Detector>
27
28     <Detector class="dtu.imm.findbugs.plugin.detect.
         UnsafeFieldAccessDetector">
29       <Details><![CDATA[<p> This detector determines if a field i accessed
             only while holding some common lock]]>
30       </Details>
31     </Detector>
32
33     <BugPattern type="DL">
34       <ShortDescription>Deadlock potential found!</ShortDescription>
35       <LongDescription>Deadlock potential found in {1}!</LongDescription>
36       <Details>
37       </Details>
38     </BugPattern>
39
40     <BugPattern type="LOLV">
41       <ShortDescription>Locking on local variable!</ShortDescription>
42       <LongDescription>Locking on local variable in {1}!</LongDescription>
43       <Details>
44       </Details>
45     </BugPattern>
46
47     <BugPattern type="LONDV">
48       <ShortDescription>Locking on non-deterministic variable!</
            ShortDescription>
49       <LongDescription>Locking on non-deterministic variable in {1}!</
            LongDescription>
50       <Details>
51       </Details>
52     </BugPattern>
53
54     <BugPattern type="NPF">
55       <ShortDescription>A field is non-private!</ShortDescription>
56       <LongDescription>A field is non-private at {1}!</LongDescription>
57       <Details>
58       </Details>
59     </BugPattern>
60
61     <BugPattern type="LHAR">
62       <ShortDescription>A lock may be held when returning from non-private
             method!</ShortDescription>
63       <LongDescription>A lock may be held when returning from non-private
            method in {1}!</LongDescription>
64       <Details>
65       </Details>
66     </BugPattern>
67
68     <BugPattern type="LON">
69       <ShortDescription>Looks like you lock on null!</ShortDescription>
```

```
70      <LongDescription>Looks like you lock on null in {1}!</
            LongDescription>
71      <Details>
72      </Details>
73    </BugPattern>
74
75    <BugPattern type="NPF">
76      <ShortDescription>A field is non-private!</ShortDescription>
77      <LongDescription>A field is non-private at {1}!</LongDescription>
78      <Details>
79      </Details>
80    </BugPattern>
81
82    <BugPattern type="UFA">
83      <ShortDescription>Unsafe access to a field!</ShortDescription>
84      <LongDescription>Unsafe access to a field {1}!</LongDescription>
85      <Details>
86      </Details>
87    </BugPattern>
88
89    <BugCode abbrev="MTSE">Multi-threaded synchronization error</BugCode>
90  </MessageCollection>
```

B.2 findbugs.xml

```
1   <FindbugsPlugin>
2
3    <Detector class="dtu.imm.findbugs.plugin.detect.DeadlockDetector" speed
         ="fast" />
4    <BugPattern abbrev="MTSE" type="DL" category="MT_CORRECTNESS" />
5
6    <Detector class="dtu.imm.findbugs.plugin.detect.
         LockHeldAtReturnDetector" speed="fast" />
7    <BugPattern abbrev="MTSE" type="LHAR" category="MT_CORRECTNESS" />
8
9    <Detector class="dtu.imm.findbugs.plugin.detect.LockCheckDetector"
         speed="fast" />
10   <BugPattern abbrev="MTSE" type="LOLV" category="MT_CORRECTNESS" />
11   <BugPattern abbrev="MTSE" type="LONDV" category="MT_CORRECTNESS" />
12
13   <Detector class="dtu.imm.findbugs.plugin.detect.NonPrivateFieldDetector
         " speed="fast" />
14   <BugPattern abbrev="MTSE" type="NPF" category="MT_CORRECTNESS" />
15
16   <Detector class="dtu.imm.findbugs.plugin.detect.LockOnNullDetector"
         speed="fast" />
17   <BugPattern abbrev="MTSE" type="LON" category="MT_CORRECTNESS" />
18
19   <Detector class="dtu.imm.findbugs.plugin.detect.
         UnsafeFieldAccessDetector" speed="fast" />
20   <BugPattern abbrev="MTSE" type="UFA" category="MT_CORRECTNESS" />
21
22  </FindbugsPlugin>
```

Appendix C

Test cases

C.1 LockTryFinally.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5
6
7  import net.jcip.annotations.ThreadSafe;
8
9  /*
10 <ExpectedBugs>
11  <BugInstance type="LHAR" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
12      <Class classname="dtu.imm.findbugs.testing.All.LockTryFinally">
13        <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally
            "/>
14      </Class>
15      <Method classname="dtu.imm.findbugs.testing.All.LockTryFinally" name
          ="test1" signature="(Z)V" isStatic="false">
16        <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally
            "/>
17      </Method>
18      <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally"
          startBytecode="67" endBytecode="67"/>
19    </BugInstance>
20    <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
```

```
21      <Class classname="dtu.imm.findbugs.testing.All.LockTryFinally">
22        <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally
            "/>
23      </Class>
24      <Method classname="dtu.imm.findbugs.testing.All.LockTryFinally" name
          ="test1" signature="(Z)V" isStatic="false">
25        <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally
            "/>
26      </Method>
27      <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally"
          startBytecode="20" endBytecode="20"/>
28      <Int value="0"/>
29    </BugInstance>
30    <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
31      <Class classname="dtu.imm.findbugs.testing.All.LockTryFinally">
32        <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally
            "/>
33      </Class>
34      <Method classname="dtu.imm.findbugs.testing.All.LockTryFinally" name
          ="test2" signature="(Z)V" isStatic="false">
35        <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally
            "/>
36      </Method>
37      <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally"
          startBytecode="20" endBytecode="20"/>
38      <Int value="0"/>
39    </BugInstance>
40    <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
41      <Class classname="dtu.imm.findbugs.testing.All.LockTryFinally">
42        <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally
            "/>
43      </Class>
44      <Method classname="dtu.imm.findbugs.testing.All.LockTryFinally" name
          ="test2" signature="(Z)V" isStatic="false">
45        <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally
            "/>
46      </Method>
47      <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally"
          startBytecode="28" endBytecode="28"/>
48      <Int value="1"/>
49    </BugInstance>
50    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
51      <Class classname="dtu.imm.findbugs.testing.All.LockTryFinally">
52        <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally
            "/>
53      </Class>
54      <Field classname="dtu.imm.findbugs.testing.All.LockTryFinally" name
          ="l" signature="Ljava/util/concurrent/locks/Lock;" isStatic="
          false">
55        <SourceLine classname="dtu.imm.findbugs.testing.All.LockTryFinally
            "/>
56      </Field>
```

```
57    </BugInstance>
58  </ExpectedBugs>
59   */
60
61  //Verified and works
62  @ThreadSafe
63  public class LockTryFinally {
64
65    private Lock l = new ReentrantLock();
66
67    public void test1(boolean b) throws Exception {
68      Lock p  = b ? new ReentrantLock() : l;
69      try {
70        p.lock();
71        if(b) {
72          Object o = new Object();
73        }
74        else {
75          throw new Exception();
76        }
77      }
78      finally {
79        p.unlock();
80      }
81    }
82
83    public void test2(boolean b) throws Exception {
84      Lock p  = b ? l : new ReentrantLock();
85      p.lock();
86      Lock q = p;
87      p.lock();
88      try {
89        if(b) {
90          Object o = new Object();
91        }
92        else {
93          throw new Exception();
94        }
95      }
96      finally {
97        p.unlock();
98        q.unlock();
99      }
100   }
101
102 }
```

C.2 ReaderWriterLocks.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantReadWriteLock;
```

```
5
6 import net.jcip.annotations.ThreadSafe;
7
8 /*
9 <ExpectedBugs>
10  <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
11      <Class classname="dtu.imm.findbugs.testing.All.ReaderWriterLocks">
12        <SourceLine classname="dtu.imm.findbugs.testing.All.
            ReaderWriterLocks"/>
13      </Class>
14      <Method classname="dtu.imm.findbugs.testing.All.ReaderWriterLocks"
          name="test3" signature="(Z)V" isStatic="false">
15        <SourceLine classname="dtu.imm.findbugs.testing.All.
            ReaderWriterLocks"/>
16      </Method>
17      <SourceLine classname="dtu.imm.findbugs.testing.All.
          ReaderWriterLocks" startBytecode="23" endBytecode="23"/>
18      <Int value="0"/>
19    </BugInstance>
20    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
21      <Class classname="dtu.imm.findbugs.testing.All.ReaderWriterLocks">
22        <SourceLine classname="dtu.imm.findbugs.testing.All.
            ReaderWriterLocks"/>
23      </Class>
24      <Field classname="dtu.imm.findbugs.testing.All.ReaderWriterLocks"
          name="l" signature="Ljava/util/concurrent/locks/Lock;" isStatic
          ="false">
25        <SourceLine classname="dtu.imm.findbugs.testing.All.
            ReaderWriterLocks"/>
26      </Field>
27    </BugInstance>
28    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
29      <Class classname="dtu.imm.findbugs.testing.All.ReaderWriterLocks">
30        <SourceLine classname="dtu.imm.findbugs.testing.All.
            ReaderWriterLocks"/>
31      </Class>
32      <Field classname="dtu.imm.findbugs.testing.All.ReaderWriterLocks"
          name="rwl" signature="Ljava/util/concurrent/locks/
          ReentrantReadWriteLock;" isStatic="false">
33        <SourceLine classname="dtu.imm.findbugs.testing.All.
            ReaderWriterLocks"/>
34      </Field>
35    </BugInstance>
36 </ExpectedBugs>
37 */
38
39 //Verified and works
40 @ThreadSafe
41 public class ReaderWriterLocks {
42
43
44   private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
```

```
45   private Lock l;
46 
47   public void test1(boolean b) {
48     ReentrantReadWriteLock local_rwl = rwl;
49     local_rwl.readLock().lock();
50     rwl.readLock().unlock();
51   }
52 
53   public void test2(boolean b) {
54     rwl.writeLock().lock();
55     rwl.writeLock().unlock();
56   }
57 
58   public void test3(boolean b) {
59     Lock p = b ? rwl.readLock() : rwl.writeLock();
60     p.lock();
61     Lock q = p;
62     q.unlock();
63   }
64 
65   public void test4(boolean b) {
66     l = rwl.readLock();
67     l.lock();
68     rwl.readLock().unlock();
69   }
70 
71 }
```

C.3 ExposedStateVariables.java

```
1  package dtu.imm.findbugs.testing.All;
2 
3  import net.jcip.annotations.ThreadSafe;
4 
5  /**
6   * Class that exposes mutable state variables, and are therefore not
       thread-safe.
7   */
8  /*
9  <ExpectedBugs>
10  <BugInstance type="NPF" priority="2" abbrev="MTSE" category="
       MT_CORRECTNESS">
11     <Class classname="dtu.imm.findbugs.testing.All.ExposedStateVariables
          ">
12       <SourceLine classname="dtu.imm.findbugs.testing.All.
            ExposedStateVariables"/>
13     </Class>
14     <Field classname="dtu.imm.findbugs.testing.All.ExposedStateVariables
          " name="d" signature="D" isStatic="false">
15       <SourceLine classname="dtu.imm.findbugs.testing.All.
            ExposedStateVariables"/>
16     </Field>
17   </BugInstance>
```

```
18    <BugInstance type="NPF" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
19      <Class classname="dtu.imm.findbugs.testing.All.ExposedStateVariables
            ">
20        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ExposedStateVariables"/>
21      </Class>
22      <Field classname="dtu.imm.findbugs.testing.All.ExposedStateVariables
            " name="i" signature="I" isStatic="false">
23        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ExposedStateVariables"/>
24      </Field>
25    </BugInstance>
26  </ExpectedBugs>
27  */
28
29  //Verified and works
30  @ThreadSafe
31  public class ExposedStateVariables {
32
33    public int i = 0;
34    public double d = 2.2;
35
36    public ExposedStateVariables() {
37      double dd = d;
38      int i = (int)dd;
39    }
40
41  }
```

C.4 ReaderWriterDeadLock.java

```
1   package dtu.imm.findbugs.testing.All;
2
3   import java.util.concurrent.locks.Lock;
4   import java.util.concurrent.locks.ReentrantReadWriteLock;
5
6   import net.jcip.annotations.ThreadSafe;
7
8   /*
9   <ExpectedBugs>
10    <BugInstance type="DL" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
11      <Class classname="dtu.imm.findbugs.testing.All.ReaderWriterDeadLock
            ">
12        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ReaderWriterDeadLock"/>
13      </Class>
14      <Method classname="dtu.imm.findbugs.testing.All.ReaderWriterDeadLock
            " name="test1" signature="(Z)V" isStatic="false">
15        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ReaderWriterDeadLock"/>
16      </Method>
```

```
17      <SourceLine classname="dtu.imm.findbugs.testing.All.
            ReaderWriterDeadLock" startBytecode="13" endBytecode="13"/>
18    </BugInstance>
19    <BugInstance type="DL" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
20      <Class classname="dtu.imm.findbugs.testing.All.ReaderWriterDeadLock
            ">
21        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ReaderWriterDeadLock"/>
22      </Class>
23      <Method classname="dtu.imm.findbugs.testing.All.ReaderWriterDeadLock
            " name="test2" signature="(Z)V" isStatic="false">
24        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ReaderWriterDeadLock"/>
25      </Method>
26      <SourceLine classname="dtu.imm.findbugs.testing.All.
            ReaderWriterDeadLock" startBytecode="21" endBytecode="21"/>
27    </BugInstance>
28    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
29      <Class classname="dtu.imm.findbugs.testing.All.ReaderWriterDeadLock
            ">
30        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ReaderWriterDeadLock"/>
31      </Class>
32      <Field classname="dtu.imm.findbugs.testing.All.ReaderWriterDeadLock"
             name="rl" signature="Ljava/util/concurrent/locks/Lock;"
            isStatic="false">
33        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ReaderWriterDeadLock"/>
34      </Field>
35    </BugInstance>
36    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
37      <Class classname="dtu.imm.findbugs.testing.All.ReaderWriterDeadLock
            ">
38        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ReaderWriterDeadLock"/>
39      </Class>
40      <Field classname="dtu.imm.findbugs.testing.All.ReaderWriterDeadLock"
             name="rwl" signature="Ljava/util/concurrent/locks/
            ReentrantReadWriteLock;" isStatic="false">
41        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ReaderWriterDeadLock"/>
42      </Field>
43    </BugInstance>
44    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
45      <Class classname="dtu.imm.findbugs.testing.All.ReaderWriterDeadLock
            ">
46        <SourceLine classname="dtu.imm.findbugs.testing.All.
              ReaderWriterDeadLock"/>
47      </Class>
48      <Field classname="dtu.imm.findbugs.testing.All.ReaderWriterDeadLock"
             name="wl" signature="Ljava/util/concurrent/locks/Lock;"
```

```
        isStatic="false">
49      <SourceLine classname="dtu.imm.findbugs.testing.All.
          ReaderWriterDeadLock"/>
50    </Field>
51  </BugInstance>
52 </ExpectedBugs>
53  */
54
55 //Verified and works
56 @ThreadSafe
57 public class ReaderWriterDeadLock {
58
59   private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
60   private Lock rl = rwl.readLock();
61   private Lock wl = rwl.writeLock();
62
63   public void test1(boolean b) {
64
65     rl.lock();
66     wl.lock();
67     wl.unlock();
68     rl.unlock();
69
70   }
71
72   public void test2(boolean b) {
73     Lock p = rwl.readLock();
74     p.lock();
75     rwl.writeLock().lock();
76     p.unlock();
77     Lock q = rwl.writeLock();
78     q.unlock();
79   }
80
81 }
```

C.5 PublicNonFinalDispatch.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5
6  import net.jcip.annotations.ThreadSafe;
7
8  /*
9  <ExpectedBugs>
10  <BugInstance type="LHAR" priority="2" abbrev="MTSE" category="
      MT_CORRECTNESS">
11    <Class classname="dtu.imm.findbugs.testing.All.
        PublicNonFinalDispatch">
12      <SourceLine classname="dtu.imm.findbugs.testing.All.
          PublicNonFinalDispatch"/>
```

```
13      </Class>
14      <Method classname="dtu.imm.findbugs.testing.All.
            PublicNonFinalDispatch" name="test1" signature="(Z)V" isStatic="
            false">
15        <SourceLine classname="dtu.imm.findbugs.testing.All.
              PublicNonFinalDispatch"/>
16      </Method>
17      <SourceLine classname="dtu.imm.findbugs.testing.All.
            PublicNonFinalDispatch" startBytecode="72" endBytecode="72"/>
18    </BugInstance>
19    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
20      <Class classname="dtu.imm.findbugs.testing.All.
            PublicNonFinalDispatch">
21        <SourceLine classname="dtu.imm.findbugs.testing.All.
              PublicNonFinalDispatch"/>
22      </Class>
23      <Field classname="dtu.imm.findbugs.testing.All.
            PublicNonFinalDispatch" name="l" signature="Ljava/util/
            concurrent/locks/Lock;" isStatic="false">
24        <SourceLine classname="dtu.imm.findbugs.testing.All.
              PublicNonFinalDispatch"/>
25      </Field>
26    </BugInstance>
27  </ExpectedBugs>
28   */
29
30  @ThreadSafe
31  public class PublicNonFinalDispatch {
32
33    private Lock l = new ReentrantLock();
34
35    public void test1(boolean b) {
36      getUnknownLock(b,l,l).lock();
37      getUnknownLock(b,l,l).lock();
38      getUnknownLock(b,l,l).unlock();
39      getUnknownLock(b,l,l).unlock();
40    }
41
42    public Lock getUnknownLock(boolean b, Lock l,Lock q) {
43      return b ? l : q;
44    }
45
46    public void test2(boolean b) {
47      getKnownLock(b,l,l).lock();
48      getKnownLock(b,l,l).lock();
49      getKnownLock(b,l,l).unlock();
50      getKnownLock(b,l,l).unlock();
51    }
52
53    public final Lock getKnownLock(boolean b, Lock l,Lock q) {
54      return b ? l : q;
55    }
56
57  }
```

C.6 DispatchTest.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5  import java.util.concurrent.locks.ReentrantReadWriteLock;
6
7  import net.jcip.annotations.ThreadSafe;
8
9  /*
10 <ExpectedBugs>
11  <BugInstance type="LON" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
12     <Class classname="dtu.imm.findbugs.testing.All.DispatchTest">
13       <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest
           "/>
14     </Class>
15     <Method classname="dtu.imm.findbugs.testing.All.DispatchTest" name="
         test1" signature="(Ldtu/imm/findbugs/testing/All/DispatchTest;)V
         " isStatic="false">
16       <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest
           "/>
17     </Method>
18     <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest"
         startBytecode="11" endBytecode="11"/>
19     <Int value="0"/>
20   </BugInstance>
21   <BugInstance type="LON" priority="2" abbrev="MTSE" category="
       MT_CORRECTNESS">
22     <Class classname="dtu.imm.findbugs.testing.All.DispatchTest">
23       <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest
           "/>
24     </Class>
25     <Method classname="dtu.imm.findbugs.testing.All.DispatchTest" name="
         test1" signature="(Ldtu/imm/findbugs/testing/All/DispatchTest;)V
         " isStatic="false">
26       <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest
           "/>
27     </Method>
28     <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest"
         startBytecode="19" endBytecode="19"/>
29     <Int value="1"/>
30   </BugInstance>
31   <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
       MT_CORRECTNESS">
32     <Class classname="dtu.imm.findbugs.testing.All.DispatchTest">
33       <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest
           "/>
34     </Class>
35     <Method classname="dtu.imm.findbugs.testing.All.DispatchTest" name="
         test1" signature="(Ldtu/imm/findbugs/testing/All/DispatchTest;)V
         " isStatic="false">
36       <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest
```

```
          "/>
37      </Method>
38      <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest"
            startBytecode="11" endBytecode="11"/>
39      <Int value="0"/>
40    </BugInstance>
41    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
42      <Class classname="dtu.imm.findbugs.testing.All.DispatchTest">
43        <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest
            "/>
44      </Class>
45      <Field classname="dtu.imm.findbugs.testing.All.DispatchTest" name="o
          " signature="Ljava/util/concurrent/locks/Lock;" isStatic="false
          ">
46        <SourceLine classname="dtu.imm.findbugs.testing.All.DispatchTest
            "/>
47      </Field>
48    </BugInstance>
49  </ExpectedBugs>
50   */
51
52  //Verified and works
53  @ThreadSafe
54  public class DispatchTest {
55
56    private Lock o = new ReentrantLock();
57
58    public void test1(DispatchTest d) {
59      Lock p = getObj(true,o);
60      p.lock();
61      Lock qq = p;
62      qq.unlock();
63    }
64
65    private Lock getObj(boolean b, Lock p) {
66      if(b) {
67        return !b ? null : new ReentrantReadWriteLock().writeLock();
68      }
69      return b ? p : new ReentrantLock();
70    }
71
72  }
```

C.7 AssignmentCycles.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5
6  import net.jcip.annotations.ThreadSafe;
7
```

```
8  /*
9  <ExpectedBugs>
10 <BugInstance type="LHAR" priority="2" abbrev="MTSE" category="
       MT_CORRECTNESS">
11     <Class classname="dtu.imm.findbugs.testing.All.AssignmentCycles">
12       <SourceLine classname="dtu.imm.findbugs.testing.All.
           AssignmentCycles"/>
13     </Class>
14     <Method classname="dtu.imm.findbugs.testing.All.AssignmentCycles"
         name="test2" signature="()V" isStatic="false">
15       <SourceLine classname="dtu.imm.findbugs.testing.All.
           AssignmentCycles"/>
16     </Method>
17     <SourceLine classname="dtu.imm.findbugs.testing.All.AssignmentCycles
         " startBytecode="41" endBytecode="41"/>
18   </BugInstance>
19   <BugInstance type="LON" priority="2" abbrev="MTSE" category="
       MT_CORRECTNESS">
20     <Class classname="dtu.imm.findbugs.testing.All.AssignmentCycles">
21       <SourceLine classname="dtu.imm.findbugs.testing.All.
           AssignmentCycles"/>
22     </Class>
23     <Method classname="dtu.imm.findbugs.testing.All.AssignmentCycles"
         name="test2" signature="()V" isStatic="false">
24       <SourceLine classname="dtu.imm.findbugs.testing.All.
           AssignmentCycles"/>
25     </Method>
26     <SourceLine classname="dtu.imm.findbugs.testing.All.AssignmentCycles
         " startBytecode="21" endBytecode="21"/>
27     <Int value="0"/>
28   </BugInstance>
29   <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
       MT_CORRECTNESS">
30     <Class classname="dtu.imm.findbugs.testing.All.AssignmentCycles">
31       <SourceLine classname="dtu.imm.findbugs.testing.All.
           AssignmentCycles"/>
32     </Class>
33     <Field classname="dtu.imm.findbugs.testing.All.AssignmentCycles"
         name="l1" signature="Ljava/util/concurrent/locks/Lock;" isStatic
         ="false">
34       <SourceLine classname="dtu.imm.findbugs.testing.All.
           AssignmentCycles"/>
35     </Field>
36   </BugInstance>
37   <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
       MT_CORRECTNESS">
38     <Class classname="dtu.imm.findbugs.testing.All.AssignmentCycles">
39       <SourceLine classname="dtu.imm.findbugs.testing.All.
           AssignmentCycles"/>
40     </Class>
41     <Field classname="dtu.imm.findbugs.testing.All.AssignmentCycles"
         name="l11" signature="Ljava/util/concurrent/locks/Lock;"
         isStatic="false">
42       <SourceLine classname="dtu.imm.findbugs.testing.All.
           AssignmentCycles"/>
```

```
43      </Field>
44    </BugInstance>
45  </ExpectedBugs>
46   */
47
48  //Verified and works
49  @ThreadSafe
50  public class AssignmentCycles {
51
52    private Lock l1 = new ReentrantLock();
53    private Lock l11;
54
55    public void test1() {
56      l11 = l1;
57      l1 = l11;
58      l1.lock();
59      try {
60        //make computation
61      }
62      finally {
63        l11.unlock();
64      }
65    }
66
67    public void test2() {
68      l1 = l1;
69      l1.lock();
70      try {
71        //make computation
72      }
73      finally {
74        l11.unlock();
75      }
76    }
77
78  }
```

C.8 ForLoopTest.java

```
1   package dtu.imm.findbugs.testing.All;
2
3   import java.util.concurrent.locks.Lock;
4   import java.util.concurrent.locks.ReentrantLock;
5   import java.util.concurrent.locks.ReentrantReadWriteLock;
6
7   import net.jcip.annotations.ThreadSafe;
8
9   /*
10   <ExpectedBugs>
11    <BugInstance type="LON" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
12      <Class classname="dtu.imm.findbugs.testing.All.ForLoopTest">
13        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
```

```
14      </Class>
15      <Method classname="dtu.imm.findbugs.testing.All.ForLoopTest" name="
            test10" signature="(Z)V" isStatic="false">
16        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
17      </Method>
18      <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"
            startBytecode="40" endBytecode="40"/>
19      <Int value="0"/>
20    </BugInstance>
21    <BugInstance type="LON" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
22      <Class classname="dtu.imm.findbugs.testing.All.ForLoopTest">
23        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
24      </Class>
25      <Method classname="dtu.imm.findbugs.testing.All.ForLoopTest" name="
            test10" signature="(Z)V" isStatic="false">
26        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
27      </Method>
28      <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"
            startBytecode="49" endBytecode="49"/>
29      <Int value="1"/>
30    </BugInstance>
31    <BugInstance type="LON" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
32      <Class classname="dtu.imm.findbugs.testing.All.ForLoopTest">
33        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
34      </Class>
35      <Method classname="dtu.imm.findbugs.testing.All.ForLoopTest" name="
            test10" signature="(Z)V" isStatic="false">
36        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
37      </Method>
38      <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"
            startBytecode="58" endBytecode="58"/>
39      <Int value="2"/>
40    </BugInstance>
41    <BugInstance type="LON" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
42      <Class classname="dtu.imm.findbugs.testing.All.ForLoopTest">
43        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
44      </Class>
45      <Method classname="dtu.imm.findbugs.testing.All.ForLoopTest" name="
            test10" signature="(Z)V" isStatic="false">
46        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
47      </Method>
48      <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"
            startBytecode="67" endBytecode="67"/>
49      <Int value="3"/>
50    </BugInstance>
51    <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
52      <Class classname="dtu.imm.findbugs.testing.All.ForLoopTest">
53        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
54      </Class>
55      <Method classname="dtu.imm.findbugs.testing.All.ForLoopTest" name="
            test10" signature="(Z)V" isStatic="false">
```

```
56        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
57      </Method>
58      <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"
            startBytecode="40" endBytecode="40"/>
59      <Int value="0"/>
60    </BugInstance>
61    <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
62      <Class classname="dtu.imm.findbugs.testing.All.ForLoopTest">
63        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
64      </Class>
65      <Method classname="dtu.imm.findbugs.testing.All.ForLoopTest" name="
            test10" signature="(Z)V" isStatic="false">
66        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
67      </Method>
68      <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"
            startBytecode="49" endBytecode="49"/>
69      <Int value="1"/>
70    </BugInstance>
71    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
72      <Class classname="dtu.imm.findbugs.testing.All.ForLoopTest">
73        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
74      </Class>
75      <Field classname="dtu.imm.findbugs.testing.All.ForLoopTest" name="p"
             signature="Ljava/util/concurrent/locks/Lock;" isStatic="false">
76        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
77      </Field>
78    </BugInstance>
79    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
80      <Class classname="dtu.imm.findbugs.testing.All.ForLoopTest">
81        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
82      </Class>
83      <Field classname="dtu.imm.findbugs.testing.All.ForLoopTest" name="q"
             signature="Ljava/util/concurrent/locks/Lock;" isStatic="false">
84        <SourceLine classname="dtu.imm.findbugs.testing.All.ForLoopTest"/>
85      </Field>
86    </BugInstance>
87   </ExpectedBugs>
88   */
89
90 //Verified and works
91 @ThreadSafe
92 public class ForLoopTest {
93   private Lock p;
94   private Lock q;
95
96   public void test10(boolean b) {
97
98     for(int i = 0; i != 100; i++) {
99       p = new ReentrantLock();
100      q = new ReentrantLock();
101    }
102    p.lock();
```

```
103     q.lock();
104
105     p.unlock();
106     q.unlock();
107   }
108 }
```

C.9 PhiResolveTest.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5
6  import net.jcip.annotations.ThreadSafe;
7
8  /*
9   <ExpectedBugs>
10  <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
11     <Class classname="dtu.imm.findbugs.testing.All.PhiResolveTest">
12       <SourceLine classname="dtu.imm.findbugs.testing.All.PhiResolveTest
            "/>
13     </Class>
14     <Field classname="dtu.imm.findbugs.testing.All.PhiResolveTest" name
          ="l" signature="Ljava/util/concurrent/locks/Lock;" isStatic="
          false">
15       <SourceLine classname="dtu.imm.findbugs.testing.All.PhiResolveTest
            "/>
16     </Field>
17   </BugInstance>
18   <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
         MT_CORRECTNESS">
19     <Class classname="dtu.imm.findbugs.testing.All.PhiResolveTest">
20       <SourceLine classname="dtu.imm.findbugs.testing.All.PhiResolveTest
            "/>
21     </Class>
22     <Field classname="dtu.imm.findbugs.testing.All.PhiResolveTest" name
          ="ll" signature="Ljava/util/concurrent/locks/Lock;" isStatic="
          false">
23       <SourceLine classname="dtu.imm.findbugs.testing.All.PhiResolveTest
            "/>
24     </Field>
25   </BugInstance>
26  </ExpectedBugs>
27  */
28
29 @ThreadSafe
30 public class PhiResolveTest {
31
32   private Lock l = new ReentrantLock();
33   private Lock ll = l;
34
```

```
35   public void test1(boolean b) {
36     Lock t = l;
37     Lock k = b ? l : t;
38     k.lock();
39     try {
40       //do some computation
41     }
42     finally {
43       l.unlock();
44     }
45   }
46
47   public void test2(boolean b) {
48     Lock t = ll;
49     Lock k = b ? l : t;
50     k.lock();
51     try {
52       //do some computation
53     }
54     finally {
55       l.unlock();
56     }
57   }
58
59 }
```

C.10 SynchronizedTests.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import net.jcip.annotations.ThreadSafe;
4
5  /*
6  <ExpectedBugs>
7   <BugInstance type="DL" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
8      <Class classname="dtu.imm.findbugs.testing.All.SynchronizedTests">
9        <SourceLine classname="dtu.imm.findbugs.testing.All.
             SynchronizedTests"/>
10     </Class>
11     <Method classname="dtu.imm.findbugs.testing.All.SynchronizedTests"
           name="Test1" signature="()V" isStatic="false">
12       <SourceLine classname="dtu.imm.findbugs.testing.All.
             SynchronizedTests"/>
13     </Method>
14     <SourceLine classname="dtu.imm.findbugs.testing.All.
           SynchronizedTests" startBytecode="13" endBytecode="13"/>
15   </BugInstance>
16   <BugInstance type="DL" priority="2" abbrev="MTSE" category="
         MT_CORRECTNESS">
17     <Class classname="dtu.imm.findbugs.testing.All.SynchronizedTests">
18       <SourceLine classname="dtu.imm.findbugs.testing.All.
             SynchronizedTests"/>
```

```
19      </Class>
20      <Method classname="dtu.imm.findbugs.testing.All.SynchronizedTests"
            name="Test2" signature="(Z)V" isStatic="false">
21        <SourceLine classname="dtu.imm.findbugs.testing.All.
              SynchronizedTests"/>
22      </Method>
23      <SourceLine classname="dtu.imm.findbugs.testing.All.
            SynchronizedTests" startBytecode="13" endBytecode="13"/>
24    </BugInstance>
25    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
26      <Class classname="dtu.imm.findbugs.testing.All.SynchronizedTests">
27        <SourceLine classname="dtu.imm.findbugs.testing.All.
              SynchronizedTests"/>
28      </Class>
29      <Field classname="dtu.imm.findbugs.testing.All.SynchronizedTests"
            name="o1" signature="Ljava/lang/Object;" isStatic="false">
30        <SourceLine classname="dtu.imm.findbugs.testing.All.
              SynchronizedTests"/>
31      </Field>
32    </BugInstance>
33    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
34      <Class classname="dtu.imm.findbugs.testing.All.SynchronizedTests">
35        <SourceLine classname="dtu.imm.findbugs.testing.All.
              SynchronizedTests"/>
36      </Class>
37      <Field classname="dtu.imm.findbugs.testing.All.SynchronizedTests"
            name="o2" signature="Ljava/lang/Object;" isStatic="false">
38        <SourceLine classname="dtu.imm.findbugs.testing.All.
              SynchronizedTests"/>
39      </Field>
40    </BugInstance>
41  </ExpectedBugs>
42   */
43
44  //Verified and works
45  @ThreadSafe
46  public class SynchronizedTests {
47
48    private Object o1 = new Object();
49    private Object o2 = new Object();
50
51    public void Test1() {
52      synchronized (o1) {
53        synchronized (o2) {
54        }
55      }
56    }
57
58    public void Test2(boolean b) {
59      synchronized (o2) {
60        synchronized (o1) {
61        }
62      }
```

```
63   }
64 }
```

C.11 ThisDeadlock.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import net.jcip.annotations.ThreadSafe;
4
5  /*
6  <ExpectedBugs>
7   <BugInstance type="DL" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
8      <Class classname="dtu.imm.findbugs.testing.All.ThisDeadlock">
9        <SourceLine classname="dtu.imm.findbugs.testing.All.ThisDeadlock
            "/>
10     </Class>
11     <Method classname="dtu.imm.findbugs.testing.All.ThisDeadlock" name="
          test1" signature="()V" isStatic="false">
12       <SourceLine classname="dtu.imm.findbugs.testing.All.ThisDeadlock
            "/>
13     </Method>
14     <SourceLine classname="dtu.imm.findbugs.testing.All.ThisDeadlock"
          startBytecode="10" endBytecode="10"/>
15   </BugInstance>
16   <BugInstance type="DL" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
17     <Class classname="dtu.imm.findbugs.testing.All.ThisDeadlock">
18       <SourceLine classname="dtu.imm.findbugs.testing.All.ThisDeadlock
            "/>
19     </Class>
20     <Method classname="dtu.imm.findbugs.testing.All.ThisDeadlock" name="
          test2" signature="()V" isStatic="false">
21       <SourceLine classname="dtu.imm.findbugs.testing.All.ThisDeadlock
            "/>
22     </Method>
23     <SourceLine classname="dtu.imm.findbugs.testing.All.ThisDeadlock"
          startBytecode="10" endBytecode="10"/>
24   </BugInstance>
25   <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
26     <Class classname="dtu.imm.findbugs.testing.All.ThisDeadlock">
27       <SourceLine classname="dtu.imm.findbugs.testing.All.ThisDeadlock
            "/>
28     </Class>
29     <Field classname="dtu.imm.findbugs.testing.All.ThisDeadlock" name="
          lock" signature="Ljava/lang/Object;" isStatic="false">
30       <SourceLine classname="dtu.imm.findbugs.testing.All.ThisDeadlock
            "/>
31     </Field>
32   </BugInstance>
33 </ExpectedBugs>
34  */
```

```
35
36 //Verified and works
37 @ThreadSafe
38 public class ThisDeadlock {
39
40   private Object lock = new Object();
41
42   public void test1() {
43     synchronized (lock) {
44       synchronized (this) {
45
46       }
47     }
48   }
49
50   public void test2() {
51     synchronized (this) {
52       synchronized (lock) {
53
54       }
55     }
56   }
57
58
59 }
```

C.12 FieldAccess.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5
6  import net.jcip.annotations.ThreadSafe;
7
8  /*
9  <ExpectedBugs>
10  <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
11     <Class classname="dtu.imm.findbugs.testing.All.FieldAccess">
12       <SourceLine classname="dtu.imm.findbugs.testing.All.FieldAccess"/>
13     </Class>
14     <Field classname="dtu.imm.findbugs.testing.All.FieldAccess" name="o"
          signature="Ljava/lang/Object;" isStatic="false">
15       <SourceLine classname="dtu.imm.findbugs.testing.All.FieldAccess"/>
16     </Field>
17   </BugInstance>
18 </ExpectedBugs>
19 */
20
21 @ThreadSafe
22 public class FieldAccess {
23
```

```
24   private final Lock l = new ReentrantLock();
25   private volatile Object o = new Object();
26
27   public void test1() {
28     l.lock();
29     try {
30       Object p = o;
31     }
32     finally {
33       l.unlock();
34     }
35   }
36
37   public void test2() {
38     l.lock();
39     try {
40       o = new Object();
41     }
42     finally {
43       l.unlock();
44     }
45     o =  new Object();
46   }
47
48 }
```

C.13 LockOnLocalVariable.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5
6  import net.jcip.annotations.ThreadSafe;
7
8  /*
9  <ExpectedBugs>
10  <BugInstance type="LOLV" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
11     <Class classname="dtu.imm.findbugs.testing.All.LockOnLocalVariable">
12       <SourceLine classname="dtu.imm.findbugs.testing.All.
           LockOnLocalVariable"/>
13     </Class>
14     <Method classname="dtu.imm.findbugs.testing.All.LockOnLocalVariable"
          name="doStuff" signature="(Z)V" isStatic="false">
15       <SourceLine classname="dtu.imm.findbugs.testing.All.
           LockOnLocalVariable"/>
16     </Method>
17     <SourceLine classname="dtu.imm.findbugs.testing.All.
         LockOnLocalVariable" startBytecode="35" endBytecode="35"/>
18     <Int value="1"/>
19   </BugInstance>
20   <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
```

```
        MT_CORRECTNESS">
21      <Class classname="dtu.imm.findbugs.testing.All.LockOnLocalVariable">
22        <SourceLine classname="dtu.imm.findbugs.testing.All.
              LockOnLocalVariable"/>
23      </Class>
24      <Method classname="dtu.imm.findbugs.testing.All.LockOnLocalVariable"
             name="doStuff" signature="(Z)V" isStatic="false">
25        <SourceLine classname="dtu.imm.findbugs.testing.All.
              LockOnLocalVariable"/>
26      </Method>
27      <SourceLine classname="dtu.imm.findbugs.testing.All.
            LockOnLocalVariable" startBytecode="29" endBytecode="29"/>
28      <Int value="0"/>
29    </BugInstance>
30  </ExpectedBugs>
31  */
32
33  @ThreadSafe
34  public class LockOnLocalVariable {
35
36
37    public void test1(boolean b) {
38      doStuff(b);
39    }
40
41    private void doStuff(boolean b) {
42      Lock l1 = new ReentrantLock();
43      Lock l2 = new ReentrantLock();
44      Lock l = b ? l1 : l2;
45      l.lock();
46      l1.lock();
47      try {
48        //do stuff
49      }
50      finally {
51        l.unlock();
52        l1.unlock();
53      }
54    }
55
56
57  }
```

C.14 PublicFinalDispatch.java

```
1  package dtu.imm.findbugs.testing.All;
2
3  import java.util.concurrent.locks.Lock;
4  import java.util.concurrent.locks.ReentrantLock;
5
6  import net.jcip.annotations.ThreadSafe;
7
8  /*
```

```
9  <ExpectedBugs>
10   <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
       MT_CORRECTNESS">
11     <Class classname="dtu.imm.findbugs.testing.All.PublicFinalDispatch">
12       <SourceLine classname="dtu.imm.findbugs.testing.All.
           PublicFinalDispatch"/>
13     </Class>
14     <Field classname="dtu.imm.findbugs.testing.All.PublicFinalDispatch"
         name="l" signature="Ljava/util/concurrent/locks/Lock;" isStatic
         ="false">
15       <SourceLine classname="dtu.imm.findbugs.testing.All.
           PublicFinalDispatch"/>
16     </Field>
17   </BugInstance>
18 </ExpectedBugs>
19 */
20
21 @ThreadSafe
22 public class PublicFinalDispatch {
23
24   private Lock l = new ReentrantLock();
25
26   public void test1(boolean b) {
27     getLock(b,l,l).lock();
28     l.lock();
29     try {
30       synchronized (getLock(b,l,l)) {
31
32       }
33     }
34     finally {
35       l.unlock();
36       Lock p = this.l;
37       getLock(b,p,l).unlock();
38     }
39
40   }
41
42   public final Lock getLock(boolean b, Lock l,Lock q) {
43     return b ? l : q;
44   }
45
46   public Lock getUnknownLock(boolean b, Lock l,Lock q) {
47     return b ? l : q;
48   }
49
50 }
```

C.15 GuardedVariable.java

```
1 package dtu.imm.findbugs.testing.All;
2
3 import java.util.ArrayList;
```

```
4  import java.util.List;
5  import java.util.concurrent.locks.Lock;
6  import java.util.concurrent.locks.ReentrantLock;
7  import java.util.concurrent.locks.ReentrantReadWriteLock;
8
9  import net.jcip.annotations.ThreadSafe;
10
11 /*
12 <ExpectedBugs>
13
14 </ExpectedBugs>
15  */
16
17 @ThreadSafe
18 public class GuardedVariable {
19
20   private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
21   private final Lock rl1 = rwl.readLock();
22   private final Lock wl1 = rwl.writeLock();
23   private volatile List<String> l = new ArrayList<String>();
24
25   public void test1() {
26     rl1.lock();
27     try{
28       l.add("test");
29     }
30     finally{
31       rl1.unlock();
32     }
33   }
34
35   public void test2(boolean b) {
36     wl1.lock();
37     try {
38       l = new ArrayList<String>();
39     }
40     finally {
41       wl1.unlock();
42     }
43   }
44
45 }
```

C.16 SynchronizedMethod.java

```
1 package dtu.imm.findbugs.testing.All;
2
3 import net.jcip.annotations.ThreadSafe;
4
5 /*
6 <ExpectedBugs>
7
8 </ExpectedBugs>
```

```
9    */
10
11  @ThreadSafe
12  public class SynchronizedMethod {
13
14    public void test1() {
15      test2();
16    }
17
18    private synchronized void test2() {
19
20    }
21
22  }
```

C.17 LockOnNullReference.java

```
1   package dtu.imm.findbugs.testing.All;
2
3   import java.util.concurrent.locks.Lock;
4   import java.util.concurrent.locks.ReentrantLock;
5
6   import net.jcip.annotations.ThreadSafe;
7
8   /**
9    * Example that illustrates the possibility to lock on a null reference
10   */
11
12  /*
13  <ExpectedBugs>
14   <BugInstance type="LHAR" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
15      <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
16        <SourceLine classname="dtu.imm.findbugs.testing.All.
            LockOnNullReference"/>
17      </Class>
18      <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
            name="test2" signature="(Z)V" isStatic="false">
19        <SourceLine classname="dtu.imm.findbugs.testing.All.
            LockOnNullReference"/>
20      </Method>
21      <SourceLine classname="dtu.imm.findbugs.testing.All.
          LockOnNullReference" startBytecode="13" endBytecode="13"/>
22    </BugInstance>
23    <BugInstance type="LHAR" priority="2" abbrev="MTSE" category="
        MT_CORRECTNESS">
24      <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
25        <SourceLine classname="dtu.imm.findbugs.testing.All.
            LockOnNullReference"/>
26      </Class>
27      <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
            name="test3" signature="(Z)V" isStatic="false">
28        <SourceLine classname="dtu.imm.findbugs.testing.All.
```

```
             LockOnNullReference"/>
29        </Method>
30        <SourceLine classname="dtu.imm.findbugs.testing.All.
             LockOnNullReference" startBytecode="27" endBytecode="27"/>
31      </BugInstance>
32      <BugInstance type="LON" priority="2" abbrev="MTSE" category="
           MT_CORRECTNESS">
33        <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
34          <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
35        </Class>
36        <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
              name="test1" signature="()V" isStatic="false">
37          <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
38        </Method>
39        <SourceLine classname="dtu.imm.findbugs.testing.All.
             LockOnNullReference" startBytecode="4" endBytecode="4"/>
40        <Int value="0"/>
41      </BugInstance>
42      <BugInstance type="LON" priority="2" abbrev="MTSE" category="
           MT_CORRECTNESS">
43        <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
44          <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
45        </Class>
46        <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
              name="test1" signature="()V" isStatic="false">
47          <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
48        </Method>
49        <SourceLine classname="dtu.imm.findbugs.testing.All.
             LockOnNullReference" startBytecode="13" endBytecode="13"/>
50        <Int value="1"/>
51      </BugInstance>
52      <BugInstance type="LON" priority="2" abbrev="MTSE" category="
           MT_CORRECTNESS">
53        <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
54          <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
55        </Class>
56        <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
              name="test2" signature="(Z)V" isStatic="false">
57          <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
58        </Method>
59        <SourceLine classname="dtu.imm.findbugs.testing.All.
             LockOnNullReference" startBytecode="8" endBytecode="8"/>
60        <Int value="0"/>
61      </BugInstance>
62      <BugInstance type="LON" priority="2" abbrev="MTSE" category="
           MT_CORRECTNESS">
63        <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
64          <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
```

```
65      </Class>
66      <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
            name="test3" signature="(Z)V" isStatic="false">
67        <SourceLine classname="dtu.imm.findbugs.testing.All.
              LockOnNullReference"/>
68      </Method>
69      <SourceLine classname="dtu.imm.findbugs.testing.All.
            LockOnNullReference" startBytecode="10" endBytecode="10"/>
70      <Int value="0"/>
71    </BugInstance>
72    <BugInstance type="LON" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
73      <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
74        <SourceLine classname="dtu.imm.findbugs.testing.All.
              LockOnNullReference"/>
75      </Class>
76      <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
            name="test3" signature="(Z)V" isStatic="false">
77        <SourceLine classname="dtu.imm.findbugs.testing.All.
              LockOnNullReference"/>
78      </Method>
79      <SourceLine classname="dtu.imm.findbugs.testing.All.
            LockOnNullReference" startBytecode="22" endBytecode="22"/>
80      <Int value="1"/>
81    </BugInstance>
82    <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
83      <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
84        <SourceLine classname="dtu.imm.findbugs.testing.All.
              LockOnNullReference"/>
85      </Class>
86      <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
            name="test1" signature="()V" isStatic="false">
87        <SourceLine classname="dtu.imm.findbugs.testing.All.
              LockOnNullReference"/>
88      </Method>
89      <SourceLine classname="dtu.imm.findbugs.testing.All.
            LockOnNullReference" startBytecode="4" endBytecode="4"/>
90      <Int value="0"/>
91    </BugInstance>
92    <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
          MT_CORRECTNESS">
93      <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
94        <SourceLine classname="dtu.imm.findbugs.testing.All.
              LockOnNullReference"/>
95      </Class>
96      <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
            name="test2" signature="(Z)V" isStatic="false">
97        <SourceLine classname="dtu.imm.findbugs.testing.All.
              LockOnNullReference"/>
98      </Method>
99      <SourceLine classname="dtu.imm.findbugs.testing.All.
            LockOnNullReference" startBytecode="8" endBytecode="8"/>
100     <Int value="0"/>
101   </BugInstance>
```

```
102    <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
           MT_CORRECTNESS">
103      <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
104        <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
105      </Class>
106      <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
             name="test3" signature="(Z)V" isStatic="false">
107        <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
108      </Method>
109      <SourceLine classname="dtu.imm.findbugs.testing.All.
             LockOnNullReference" startBytecode="10" endBytecode="10"/>
110      <Int value="0"/>
111    </BugInstance>
112    <BugInstance type="LONDV" priority="2" abbrev="MTSE" category="
           MT_CORRECTNESS">
113      <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
114        <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
115      </Class>
116      <Method classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
             name="test3" signature="(Z)V" isStatic="false">
117        <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
118      </Method>
119      <SourceLine classname="dtu.imm.findbugs.testing.All.
             LockOnNullReference" startBytecode="22" endBytecode="22"/>
120      <Int value="1"/>
121    </BugInstance>
122    <BugInstance type="UFA" priority="2" abbrev="MTSE" category="
           MT_CORRECTNESS">
123      <Class classname="dtu.imm.findbugs.testing.All.LockOnNullReference">
124        <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
125      </Class>
126      <Field classname="dtu.imm.findbugs.testing.All.LockOnNullReference"
             name="l" signature="Ljava/util/concurrent/locks/Lock;" isStatic
             ="false">
127        <SourceLine classname="dtu.imm.findbugs.testing.All.
               LockOnNullReference"/>
128      </Field>
129    </BugInstance>
130  </ExpectedBugs>
131   */
132
133  @ThreadSafe
134  public class LockOnNullReference {
135
136    private Lock l;
137
138    public LockOnNullReference(boolean b) {
139      if(b) l = new ReentrantLock();
140    }
141
```

```
142   public void test1() {
143     l.lock();
144     try {
145       //make computation
146     }
147     finally {
148       l.unlock();
149     }
150   }
151
152   public void test2(boolean b) {
153     if(b) {
154       l.lock();
155     }
156   }
157
158   public void test3(boolean b) {
159     Lock q = l;
160     if(b) {
161       q.lock();
162     }
163     else {
164       l.lock();
165     }
166   }
167
168 }
```