A General Framework for Concurrency Aware Refactorings

Maria Gouseti
mgouseti@gmail.com

August 29, 2014, 92 pages

Supervisor: Jurgen Vinju
Host organisation: Centrum Wiskunde & Informatica, www.cwi.nl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

A refactoring tool is a code transformation tool that aims to improve non-functional attributes of existing code, such as maintainability, by changing the code's structure and not the code's behaviour. In this project we study the current implementations of several refactoring tools that when they are applied on a program and the outcome is executed in parallel, such bugs are introduced, even though the refactorings are correct in sequential execution. Concurrency bugs, are difficult to find using regression tests because they may occur in rare thread schedules. For this reason we argue that it is not enough to fix the implementations of these refactorings until they pass the tests but also to prove their correctness.

In this effort we introduce a domain specific language that consists of statements that capture the data and synchronisation dependencies. The representation of a program in this language will serve as a constraint to prove its correctness. The dependencies reflected in this language should be preserved after the refactoring so that the preservation of the code behaviour is guaranteed. For example, reading a variable v has a data dependency with the last assignment to v; consequently, if another assignment to v is added between them, it will break this dependency and change the behaviour of the code if different values were assigned.

Our work focuses on three refactorings, Move Method, Convert Local Variable To Field and Inline Local, and we build a prototype tool CARR using Rascal to apply these refactorings. The results of this tool are compared with the implementation of the same refactorings in Eclipse. We claim that CARR is a correct concurrency aware refactoring implementation and we provide evidence and a proof outline to support our claim.

Chapter 1

Introduction

A refactoring tool is a code transformation tool that restructures an existing body of code, altering its internal structure without changing its external behaviour. According to Fowler, so far, refactoring tools were focused on sequential code, *"Another aspect to remember about these refactorings is that they are described with single-process software in mind"* [6]. However, as parallel execution becomes more and more popular with the increasing production of multicore processors, concurrency aware refactorings will be useful.

Schäfer et al. [25] demonstrate how the current Eclipse[1] implementation of Move Method and Inline Local introduces concurrency bugs. The authors argue that because concurrency bugs may depend on a specific scheduling of threads, such bugs are hard to find by testing, consequently, proving the correctness of concurrency aware refactoring tools is necessary.

In this work we study how three refactorings, Move Method, Convert Local Variable To Field and Inline Local, work and we make them concurrency aware. To accomplish that, we introduce an intermediate language that captures the data and synchronisation dependencies and is used after the refactoring transformation to determine if the refactoring violated the original dependencies. Under limitations and assumptions we claim that dependency preservation implies external behaviour preservation.

The biggest challenge lies in proving that the updated algorithm is reliable. Since testing is not effective to detect concurrency bugs, the proof of the refactoring's correctness is crucial. In this work we search for tools to facilitate this proof and provide a proof outline.

So far, there is limited work that combines refactorings and concurrency and mostly focuses on refactorings that make sequential code safe for concurrent execution.

1.1 Initial Study

Schäfer et al. [25] present five examples of refactorings that break behaviour preservation when the refactored code is executed concurrently and they provide amendments to fix them. To implement their refactoring tool, JastAdd Refactoring Tool (JRRT)[2], they use invariant preservation to verify that the refactoring tool has not changed code behaviour. This technique abstracts the program to a structure and requires that this structure is preserved after the refactoring. Based on the structure the authors use, they separate the refactorings into two categories, *memory trace preserving* and *dependence edge preserving* refactorings.

Memory Trace: Move Method, Pull Up Members. The authors abstract the code to Java Memory Model (JMM) actions, JMM actions will be discussed further in Section 2.2; one can think of them as a sequence of shared memory accesses. Consequently, under the assumption that the refactoring was correct in sequential execution, if the accesses to the shared memory location have the same sequence then the refactoring preserved the memory trace, which implies behaviour preservation [33].

[1] http://www.eclipse.org/
[2] https://code.google.com/p/jrrt/

One can think this as if the actions that apply changes on the memory are preserved, then the program will read and write the same values so it will exhibit the same behaviour.

Dependence Edge: Extract Local, Inline Local, Convert Int To AtomicInteger. The authors abstract the code to a graph that captures the dependencies between the JMM actions, which are called synchronisation dependencies. In this category, the authors require that the refactorings preserve the synchronisation dependence edges in addition to the data and control edges of the flow graph [20] which were already used to verify the correct refactoring of sequential code. Generally, data dependencies ensure that the dependencies between local memory accesses are preserved and the synchronisation dependencies that the dependencies of shared memory accesses are preserved. For the dependence edge preserving refactorings, the authors prove behaviour preservation for correctly synchronised programs, and guarantee not to introduce new concurrency bugs.

As future work, they mention a third category of refactorings that introduces new shared state and they name Convert Local Variable To Field as a representative example of this category.

This thesis, is an extension of their work; we implement a prototype CARR (**C**oncurrency **A**ware **R**efactorings with **R**ascal), which is a concurrency aware refactoring tool that applies a refactoring from each category, Move Method, Inline Local, using Rascal [15, 16]. The goal of CARR is to implement the concurrency aware refactorings using a comprehensive theory basis, instead of using the two different theories [25]. The Convert Local Variable To Field refactoring is added to CARR due to this theory's extensibility.

1.2 Problem Statement

The problem we study regards the current implementation of refactorings that are not concurrency aware, this means that when the refactored code is run concurrently, concurrency bugs are introduced. The current implementation we studied is the built-in implementation of Eclipse.

The difference between refactoring sequential and concurrent code lies on the manipulation of shared memory. Consequently, if the shared memory is not affected by the refactoring, no concurrency bugs are introduced [25].

However, this is not always the case, in Chapter 3, we present four examples that demonstrate refactorings that changed the behaviour of code. The first example uses Move Method to show errors introduced by *memory trace preserving* refactorings, then there are two examples of Inline Local that represent *dependence edge preserving* refactorings and the last example regards Convert Local Variable To Field that represents the *introducing new shared state* refactorings.

1.2.1 Research Questions

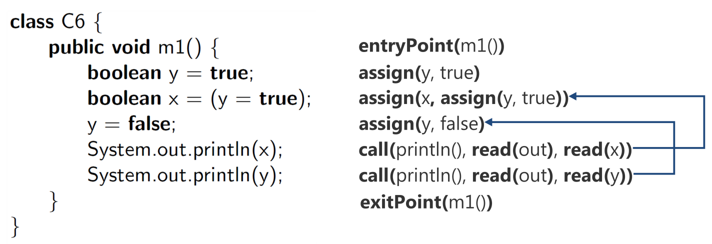
We examine the motivating examples presented in Chapter 3 from the perspective of the following research questions:

- When does the current implementation of the targeted refactorings change the behaviour if the refactored program runs concurrently?
- Can we fix the reasons that caused the new behaviour?
- Can we prove the correctness of the fixed version?

1.2.2 Solution Outline

Our solution uses the transformation of the original and the refactored program to an intermediate language to compare the dependencies of the two versions and detect changes that were not anticipated with respect to the refactoring.

The intermediate language is called Synchronised Data Flow Graph (SDFG) language and captures the data and the synchronisation dependencies of Java code, Figure 1.1 illustrates the transformation from Java to SDFG of one of the examples from Chapter 3. Figure 1.2 demonstrates how CARR uses the SDFG to answer the second research question.



```
class C6 {
    public void m1() {
        boolean y = true;
        boolean x = (y = true);
        y = false;
        System.out.println(x);
        System.out.println(y);
    }
}
```

```
entryPoint(m1())
assign(y, true)
assign(x, assign(y, true))
assign(y, false)
call(println(), read(out), read(x))
call(println(), read(out), read(y))
exitPoint(m1())
```

Figure 1.1: The transformation of Java code to simplified SDFG code.

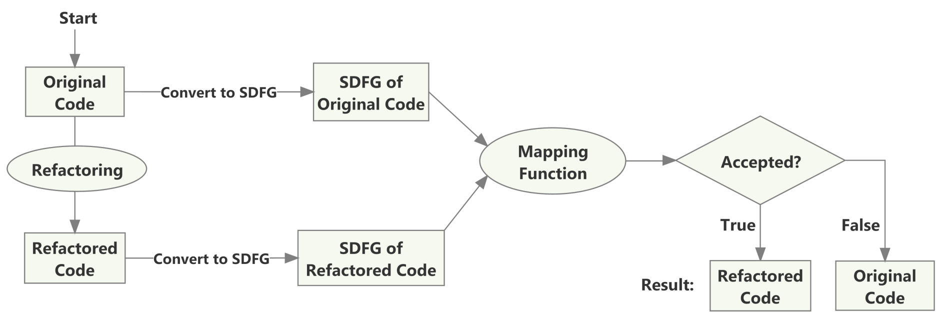


Figure 1.2: The general algorithm used in CARR. The circles represent the parts of the algorithm that are refactoring specific.

The refactoring algorithms were extended with extra steps to account for synchronisation aspects that were not needed sequentially. After applying an extended refactoring algorithm, both the original and the refactored versions are converted to their SDFG equivalent. The anticipated changes of the refactoring are formalised in a mapping function that constrains the changes of the dependencies. If the changes of the dependencies do not satisfy the constraints, then the refactoring is rejected, otherwise it is accepted.

To answer the third research question, we use the anticipated changes formalised as inference rules to provide a proof outline of the equivalence of the premise and the conclusion of these inference rules.

1.2.3 Research Method

Before attempting to answer the research questions we conducted a literature study on refactorings and programming language semantics in our effort to clarify what behaviour preservation means and how it can be proved. We also experimented with the refactoring engine of Eclipse to reverse-engineer the pre-conditions of the refactorings we were going to study. The results of this literature study are presented in Section 1.4.

Example Replication. We chose one representative example from each refactoring category [25], we replicated the experiment using CARR and confirmed what was reported in the paper. We did not use the JRRT because of the difficulties we encountered setting up and reusing this tool along with the control and data flow graph of Nilsson-Nyman et al. [20], since the current version of JastAddJ[3] is not compatible.

New Example. We added another example regarding the CONVERT LOCAL VARIABLE TO FIELD refactoring which is categorised as an *introducing new shared state* refactoring [25], the example proved that the refactored code presents new behaviour when run concurrently.

Designing SDFG. To capture all data and synchronisation dependencies [20, 25, 26, 27] that should be respected when a refactoring is applied in one structure, we designed the intermediate language SDFG.

[3] https://bitbucket.org/jastadd/jastaddj

Designing CARR. We designed a concurrency aware refactoring tool that would fix the concurrency unaware refactorings using SDFG.

Prototype. We prototyped our solution by implementing the three refactorings we used as motivating examples and examined if the refactored versions of the examples preserved behaviour.

Evaluation. To argue about the correctness of CARR, we provide evidence and a proof outline.

Evidence. We used use cases to collect evidence that our tool was sound. The use cases were two versions of a parallel implementation of the sieve of Eratosthenes (see Appendix B) that contained both synchronisation variances of Java, `volatile` variables and `synchronized` blocks, and smaller examples to demonstrate that our prototype works correctly in the "special" control flow cases [20, 26, 27].

Proof Outline. We formalised the anticipated changes per refactoring as inference rules and provided an outline of how separation logic can be used to examine the correctness of these inference rules. The formal proof is left as future work.

1.3 Contributions

Contribution #1: Example Replication. From the perspective of Schäfer et al. [25], we use two of their motivating examples and study the CARR implementation of the refactorings applied on them, Move Method and Inline Local, as it is discussed in Chapter 3; we extend their work to include the Convert Local Variable To Field refactoring which was left as future work. We confirmed the errors they found in both examples and solved them in a different way, resulting in the same outcome.

Schäfer et al. [25, 26, 27] use *memory trace preservation* and *dependence edge preservation* to verify the correctness of their framework. However, neither of them could be extended to be applied Convert Local Variable To Field. To confront that problem, we adopted the idea of abstracting Java code to a simpler structure but we searched for a theory that could be used as basis for all refactorings.

Contribution #2: Common Theory. To find the common theory, we researched different areas; these areas included programming language semantics [19], domain specific languages [1, 8], graph theory [18, 35] and separation logic [21, 23, 24, 32].

We put forth an intermediate language called Synchronised Data Flow Graph (SDFG) inspired by the Object Flow Graph (OFG) language from Tonella et al. [1]. We implemented a converter from Java to SDFG based on the Rascal implementation of a converter from Java to the OFG language[4]. The challenge was to completely capture all the relevant details of Java semantics, such as data dependencies for all memory locations shared or local, control flow and synchronisation dependencies.

SDFG maps a Java program to Java Memory Model (JMM) actions, the suitability of the JMM abstraction was also noted by Schäfer et al. [25], extended with reads and writes of local variables which allows us to argue about code re-orderings and their effect on program behaviour given the limitations enforced by an inter-procedural analysis and the assumption that all variables of the same class type refer to the same memory location. The last assumption was used to avoid analysing aliases which is undecidable [9, 12] and not in the scope of our work. Furthermore, we provide inference rules that constrain the anticipated changes.

This language in combination with the desirable constraints between the two versions of a program consists a configurable tool that can be used to argue about the behaviour of code for any refactoring. In Chapter 4, there is an extended description of the language and in Appendix C the converter's implementation.

Contribution #3: Prototype Implementation. We implemented CARR, a concurrency aware implementation of the three refactorings we studied as use cases, to evaluate the previous theory.

[4] `https://github.com/cwi-swat/rascal-OFG`

CARR uses the SDFG as a constraint and configures the anticipated changes depending on each refactoring. In case of Move Method and Inline Local, CARR performs extra steps in addition to the algorithm of the sequential refactoring to fix the concurrency bugs. In case of Inline Local CARR takes into account the synchronisation dependencies in addition to the data dependencies and rejects the refactorings that break the dependencies. For every refactoring we note our claims and provide rules that express the anticipated changes from the refactoring. These inference rules are then used to verify the invariant preservation. As a result, the correctness of the CARR is implied by the correctness of the SDFG converter and the inference rules. The CARR refactoring tool is analysed in Chapter 5.

Contribution #4: Proof Outline. After collecting evidence that CARR works correctly by applying the tool on code especially chosen to demonstrate how CARR handles the cases that were mishandled by other implementations, we provide a proof outline that argues about CARR's correctness (see Chapter 6). First we apply the inference rules on the refactored SDFG program to invert the refactoring; if the inverted program is a subset with the original SDFG one then the refactoring is correct. To prove that the inferred program is equivalent with the refactored one, we need to prove that the inference rules are equivalent. The proof of the equivalence of the inference rules is outlined but the formal proof is left as future work.

1.4 Related Work

In this section we discuss related work from various areas of research that affected our work. We specifically note the similarities and differences from our work.

Java Memory Model

The section 17.4 from the Java Memory Model Specification [7] is the standard specification of the JMM, it defines the data accesses, the synchronisation actions and the dependencies between them, in other words the JMM determines if a reordering of code preserves behaviour in concurrent execution. Manson et al. [17] and Ševčík et al. [33] analyse the current JMM, challenge its correctness and argue about properties that a memory model should support. In our work we assume that the current JMM is correct and we do not go deeper in this area. However, if there is a change in the JMM that changes the synchronisation dependencies, then the SDFG will have to adapt to these changes.

Program Dependence Graph

Program Dependence Graphs (PDG) [5] are used to define the control and data flow of a program. PDG are used to check the validity of optimisations since walking the edges between dependencies is sufficient to perform many optimisations. In our work we needed the same structure but extended with synchronisation dependencies. For this reason our intermediate language can be thought of as a combination of PDG and the Java Memory Model.

Aliasing in Java

Aliasing in Java allows different variables to *point to* the same object. Alias analysis or pointer analysis is in general undecidable [12]. Hind [9] performs a literature study and cites the results of over seventy-five papers on this area. Specifically, he separates the different algorithms for pointer analysis based on optimisations and how precise their analysis is.

In our work we need alias analysis to find the data and synchronisation dependencies between memory accesses. However, this analysis falls into the category of program understanding which according to Hind [9, 28] requires high precision. In this project, we assume that all the variables of the same class point to the same object. This assumption captures an upper bound of the dependencies found in an inter-procedural analysis potentially rejecting refactorings that may not change the program's behaviour.

Control Flow Graph

Nilsson et al. [20] use attribute grammars to build an inter-procedural control flow tool based on the JastAdd [4] extensible compiler. The authors underline the way they handled “special” statements such as **break**, **return** and **finally**. Our SDFG is based on their idea of inter-procedural analysis; however, our implementation differs not only in the execution but also in the outcome, since their outcome is a control flow graph on top of the Abstract Syntax Tree (AST) and ours is a program in a different language.

The advantages of building the control flow on top of the AST are that it allows easy manipulation and that each node contains all the information from the AST. On the other hand, our implementation results in a transformation of the initial program to an intermediate language which is an entity that can stand on its own, for instance, it can be saved to a file. Furthermore, the SDFG is a flattened set of `Stmts`, which makes the operations on it simpler.

Refactorings

In addition to the work of Schäfer et. al [26, 27, 25], which uses invariant preservation in their implementation to argue about correctness, there are other approaches concerning sequential code. Opdyke [22] establishes pre-conditions and post-conditions that have to be met in order for the refactoring to be correct. Our work uses constraints that express the expected outcome instead of pre-conditions and post-conditions which makes it more flexible, since the constraints are based on the original program and they are not predefined.

Tip et al. [30, 31] use type constraints to argue about the correctness of generalisation refactorings such as Extract Interface. A similar approach is used by Steimann et al. [29] to argue about the current implementation of refactorings such as Move Class that change the behaviour of the program because they do not update the modifiers. Kegel et al. [14] use constraint graphs originally developed for type inference [30, 31] to establish pre-conditions and post-conditions for refactoring inheritance to delegation. As we see in this paper, although type constraints are suitable for these type of refactorings, when synchronisation aspects are taken into account, the authors deal with them by enhancing pre-conditions, resulting in rejecting refactorings that could be applied.

Other papers that combine refactorings and concurrency such as the Reentrancer [34], which enables safe parallel execution through re-entrancy, the Concurrencer [3], which changes Java code to use concurrent libraries, and a toolbox of refactorings [2] that aim to improve latency, throughput, scalability and thread safety, are orthogonal to our work since they focus on tools that transform sequential code to concurrent whereas we aim at preserving the existing synchronisation structures of the program, not at introducing them.

Intermediate Languages

Christian Haack et al. [8] present a Java-like language that enables them to use separation logic to prove properties of programs, for instance, that a program is race-free. However, the way this language handles read and write permissions disallows races and this makes it inapplicable to our work since our goal is to argue about the correctness of programs with or without races. Furthermore, following the idea of the initial paper, our main focus is on shared memory accesses under the assumption that the correctness of a refactoring applied on sequential code is given. Because this language is much richer than what we need, it introduces complexity that can be avoided.

A simple language that was the inspiration for our language was the Object Flow Language [1]. This language consists of declarations and simple dependencies of reads, writes, calls and constructors. However, this language could not be used as it is, because it was control insensitive and concurrency unaware.

Proving Correctness

Wood et al. [35] suggest a method for proving that a refactoring does not change the program’s behaviour; they partition the memory into the modified by the refactoring part and the unaffected part. Then, they check if there is an isomorphic relation between executions of the two versions that can be used as basis for tools that decide the correctness of a refactoring.

In a similar concept Nielson et al. [19] use bi-simulation to prove that one step in structural semantics can be simulated by a non empty sequence of steps on the abstract machine they defined regarding the "WHILE" language; a language they used throughout the book.

Other methods that are used to argue about parallel semantics are rely-guarantee and separation logic. Rely-guarantee is introduced by Jones [13] and suggests establishing two properties of code. The first property is the *rely* property which defines what the code expects by its environment; the second property is *guarantee* which defines the changes the code is expected to cause to its environment. This method is appropriate for arguing about interleaving code; however, the analysis is applied to the global state of the program, which does not make it scalable.

A different approach is separation logic. O'Hearn [21] uses separation logic to partition the memory into a part that is affected by a snippet of code and another that is not. Two snippets of code interact with each other when there is an overlapping between the parts they affect. The advantage of this analysis is that it is local; however, it cannot analyse interleaving code due to this locality. To achieve this kind of analysis, the complexity increases because auxiliary variables are used [24].

The two methods are bridged by Vafeiadis et al. [32], the authors use both methods in the analysis depending on the case. However, this approach still remains more complex than what we need. Parkinson [23] uses separation logic and invariant preservation to analyse concurrent code. This method is more suitable than the others to prove the correctness of our solution since although it is simple, it still supports races by using an invariant instead of auxiliary variables.

1.5 Outline

In this section we outline the structure of this thesis. In Chapter 2 we introduce the background (refactorings, JMM, concurrency in Java and separation logic). In Chapter 3 we demonstrate our motivating examples. In Chapter 4, the SDFG language is presented and we provide details of the implementation of the converter. In Chapter 5 we describe the algorithms used in the implementation of CARR and provide details specifically for each refactoring. Additionally, we document the inference rules that we use as basis to claim that our implementation is correct. In Chapter 6 we present our study case and our results along with an outline of a proof regarding the claims we made in the previous chapters. Finally, we conclude in Chapter 7.

Chapter 2

Background

2.1 Refactoring

Martin Fowler defines a refactoring as follows:

> "Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour."[1]

A refactoring tool is a program that has as input the source code of another program and performs a refactoring on the input. In general, there are different ways of implementing a refactoring tool but they all have in common two phases, the code transformation and the check of correctness based on the method they use. For example, a refactoring tool follows the steps below:

1. checks if the refactoring can be performed by setting pre-conditions that should be satisfied by the original code,
2. transforms the original code and
3. checks if the refactored code satisfies the post-conditions.

An alternative method for checking the refactored code is invariant preservation, which extracts the invariant from the original code in step 1, then in step 3 extracts the invariant from the refactored code and checks if the invariant is preserved.

Based on the ideas from Fowler's book of refactorings [6] and the conclusions we made while reverse-engineering the Eclipse implementation and implementing CARR, we cite the guidelines concerning the three refactorings that we are going to focus on. In the next paragraphs we assume that the targeted code is run sequentially, no concurrency aspects are taken into account. Additionally, we assume the reader has a firm grasp of object oriented language semantics.

Move Method: This refactoring moves a method from a class to another. However, there are conditions to be met in order to make this refactoring possible. In this paragraph, *Method* is the method to be moved, and *SourceClass* and *DestinationClass* are the classes that the *Method* is moved from and to respectively.

- *Method* should not be inherited by a subclass or override a method of a superclass of *SourceClass*.
- The declaration of *Method* in *DestinationClass* should not have the same naming with an existing or an inherited method.
- Assuming that *Method* uses the *SourceClass*, it needs a way to refer to it. If it is static it can refer to it by the qualified name. Otherwise *SourceClass* should be given as a new parameter.

[1] http://www.refactoring.com

- In case that *Method* has a parameter of type *DestinationClass* it is possible to replace this parameter with the new parameter of type *SourceClass*. The old parameter can be used as the receiver of *Method*.

- If *DestinationClass* is not a parameter and *Method* is not static, then we require that *DestinationClass* is a field of *SourceClass* in order to ensure a way of calling *Method*.

- The code of *Method* needs to be updated according to which of the previous cases used to move *Method*.

- All calls of *Method* should be updated to match the refactored method.

- The visibility of fields of *SourceClass* may need to be changed to *protected* or *public*.

INLINE LOCAL: This refactoring replaces all the accesses of a local variable with the expression of the last assignment to that variable. The version of the refactoring we implemented is broader than the *Inline Temp* from Fowler [6] and Eclipse implementation, since we do not require that the local variable can be defined as final. In this paragraph, *Local* is the variable to be replaced and *Expression* is the expression that is going to be inlined.

- If *Expression* contains method calls the evaluation of the expression cannot be guaranteed since the method calls could depend on or change the program state, even if the expression is inlined only once, the method call might still return a different value, if the part of the program state used by the method changed between the initial location of the method call and the new one. Additionally, if the local variable refers to an object we cannot check that the state of the object has not changed between two subsequent reads.

CONVERT LOCAL VARIABLE TO FIELD: This refactoring converts a local variable to a field. In this paragraph, *Local* will be the targeted local variable, *Field* the new field that is going to be created and *Class* the class that contains the method in which *Local* is defined.

- *Field* should not conflict with any other field or inherited fields of *Class*.

While implementing a refactoring tool the above guidelines are a part of a bigger mechanism, there are two popular ways to implement this either by pre-conditions and post-conditions [22] that are checked before and after the refactoring or by invariant preservation [25, 26, 27] which abstracts the program into a structure and tries to preserve this structure. In CARR we use a hybrid of the two methods, we abstract Java code to an intermediate language and check that the original and the refactored versions comply with the constraints enforced by the refactoring.

2.1.1 Categories of Refactorings

As we mentioned in Section 2.1, the choice of the targeted refactorings was not random. They are representative examples of the following categories of refactorings:

- *Memory Trace Preserving.* These refactorings preserve memory trace since they do not affect variable or field accesses. Memory trace is the sequence of data access and synchronisation dependencies as defined by the JMM. Examples: MOVE METHOD, PULL UP MEMBERS.

- *Code Rearrangement.* These refactorings may change the memory trace if they reorder field accesses. Examples: INLINE LOCAL, EXTRACT LOCAL, CONVERT INT TO ATOMICINTEGER.

- *Introducing New Shared Memory.* These refactorings introduce new shared memory. For instance, CONVERT LOCAL VARIABLE TO FIELD converts the local variable to a field, which means that the field is shared by parallel executions of the same function.

In addition to the data dependencies that are already taken into account in sequential execution, refactorings needs to respect the synchronisation dependencies enforced by JMM [7] that define the ordering of memory accesses when the code is run concurrently.

2.2 Java Memory Model

The Java Memory Model (JMM) is used from Schäfer et al. [25] to create a graph of synchronisation dependencies between statements of the original code. The JMM imposes reordering constraints which allow the acceptance or the rejection of a refactoring. Consequently, the authors extract the dependency graph of the refactored program as well, and if a reordering violates one of the dependencies it is possible that it changes the behaviour of the code so the refactoring is rejected.

The dependency edges are separated into two categories: the data dependencies and the synchronisation dependencies. The data dependencies define the value that is recovered by a read of a memory location or the value that is written in a memory location. On the other hand, the synchronisation dependencies ensure how the code might interleave with other code or itself when it is executed in parallel. The synchronisation dependencies can be thought of as locks that protect blocks of code.

The JMM abstracts Java code to a sequence of actions that are connected by dependency edges and introduces rules that determine when a reordering changes the code's behaviour. A reordering is accepted when it does not change data dependencies and it does not remove code outside of a protected block.

The actions of JMM are listed below and separated into categories according to the dependencies they enforce, these dependencies establish the happens-before relation between actions and determine if a program is correctly synchronised.

- Data Access Actions
  - *read()* is the action that reads the value of a shared memory location that is not volatile (see Section 2.3.2). This action has a data dependency on the last write that is found in the code before this read.
  - *write()* is the action that writes a value on a shared memory location that is not volatile.
- Synchronisation Actions
  - *monitorEnter()* is the action that signifies entering a block that is protected by a lock. This means that this block of code cannot interleave with other blocks of code that are also protected by this lock. This action has an acquire dependency with all the actions that follow it. An acquire dependency indicates that only a thread that holds that lock can execute the dependent `Stmts`, all the other threads that have acquire dependent `Stmts` on this lock are blocked until the first releases the lock. As a result, we have a happens-before relation between the acquire action of a lock and the release action of the same lock.
  - *monitorExit()* is the action that signifies exiting a block that is protected by a lock. This action has a release dependency with all the actions that proceed it. A release dependency indicates that a thread that held that lock releases it, consequently, another threads that has `Stmts` with acquire dependencies to that lock can proceed with execution. As a result, we have a happens-before relation between the release action of a lock and an acquire action of the same lock.
- Data Access & Synchronisation Actions
  - *volatileRead()* is an action that represents the read of a `volatile` field. It has a data dependency with the last write found before this action and an acquire dependency with all the actions that follow it. The acquire dependency due to a `volatile` read means that all the subsequent actions are protected by it.
  - *volatileWrite()* is an action that represents the change of value of a `volatile` field. It has a release dependency with all the actions that proceed it. The release dependency due to a `volatile` write means that all the previous actions are protected by it.

2.2.1 Correctly Synchronised Programs

A program is characterised as correctly synchronised by JMM if it does not contain races [7]. Other synchronisation issues are deadlocks and livelocks. These terms are explained below.

Race: A race occurs when the happens-before relation is absent between two accesses of the same shared memory location from which at least one of them changes its value.

Deadlock: A deadlock occurs between one or more threads when each thread is blocked by a lock already held by the other. For instance, a thread *T1* holds lock *a* and requests lock *b*, meanwhile thread *T2* holds lock *b* and requests lock *a*, as a result both threads are blocked and cannot continue with their execution.

Livelock: A livelock occurs between one or more threads when each thread waits for an action from another thread. For instance, a thread *T1* waits on a variable *a* to turn *false* (**while**(a);) and then it can assign **false** to *b*. However, thread *T2* also waits on variable *b* to turn **false** before it can turn variable *a* to **false**. Consequently, both threads are busy-waiting on the loops and neither of them can change the state of the other.

2.3 Concurrency in Java

In this section we explain the built-in synchronisation structures of Java, the keywords `synchronized` and `volatile`.

2.3.1 Synchronized

The keyword `synchronized` is a form of monitor lock; it enforces exclusive access into the block of code it *monitors*, which is the code block it is associated with. The effect of `synchronized` is that when a thread acquires the lock, no other thread can execute the code blocks that are protected from the same lock until the first thread releases the lock. Entering a `synchronized` block corresponds to a *monitorEnter()* action of the JMM and exiting the block to a *monitorExit()* action. The lock is an object, for example, an instance of a class or a *Class* object (see the following listing). By using an instance as a lock, two different instances consist two different locks even if they are instances of the same class, whereas by using as a lock the *Class* object, the lock is unique and shared between all instances.

```
{...
        synchronized(this){
            //monitored code
        }

        synchronized(ClassName.class){
            //monitored code
        }
...}
```

Synchronized Blocks.

Another way to use `synchronized` is as a modifier of a method. This is syntactic sugar for enclosing the code of the method in a `synchronized` block with lock either the instance of the receiver of the method, **this**, or the *Class* object of the enclosing class in case of static methods.

```
class A {
    synchronized void foo() {

        //protected code

    }

    static synchronized void bar() {

        //protected code

    }
}
```

```
class A {
    void foo() {
        synchronized(this){
            //protected code
        }
    }

    static void bar() {
        synchronized(A.class){
            //protected code
        }
    }
}
```

Desugaring `synchronized` modifier.

Finally, these monitor locks are re-entrant which means that if a thread has already acquired a lock it can enter any other block of code monitored by the same lock.

2.3.2 Volatile

The keyword `volatile` is used as a modifier of a field and indicates that the changes of this field will be visible to other threads after their completion. All accesses of `volatile` fields are seen as atomic. An access of a `volatile` field enforces the dependencies described in Section 2.2 depending on whether the access is a read or a write.

One can think of a `volatile` field as a *fence* that ensures that after a thread writes on a `volatile` variable all the updates that happened before the write including the write itself will be visible by all other threads.

2.4 Hoare Logic & Separation Logic

Separation logic [24] extends Hoare logic [10, 11] and it can be used to reason about programs that share state and use mutable data structures.

Hoare logic uses the triple in equation 2.1 to define how the execution of command C changes the state of the program. Given state s, P is the pre-condition that if it holds before the execution of C then it is guaranteed that the post-condition Q will also hold for a state s'. P and Q are formulae of predicate logic and s is a mapping between variables and values.

$$\{P\}C\{Q\} \tag{2.1}$$

Separation logic separates the state from Hoare logic into two parts, the store (or stuck), which maps variables to values or to memory addresses, and the heap, which maps addresses to values.

In addition to the formulae from predicate logic, separation logic uses the following assertions:

- **emp** asserts that the heap is empty (Empty Heap).
- $e \mapsto e'$ asserts that the heap contains a cell at address e with value e' (Singleton Heap).
- $p_0 * p_1$ asserts that the heap can be separated into two disjoint parts in which p_0 and p_1 hold separately (Separating Conjunction).
- $p_0 - * p_1$ asserts that if the current heap is extended with a disjoint part in which p_0 holds, then p_1 will hold in the extended heap (Separating Implication).

In equation 2.2 we can see how concurrency is handled by separation logic given that p_1 and p_2 and q_1 and q_2 are disjoint, meaning that they refer to disjoint parts of the state:

$$\frac{\{p_1\}\ c_1\ \{q_1\},\ \{p_2\}\ c_2\ \{q_2\}}{\{p_1 * p_2\}\ c_1\ ||\ c_2\ \{q_1 * q_2\}} \tag{2.2}$$

Chapter 3

Motivating Examples

3.1 Execution Harness

All the examples from this chapter apart from C6, which does not need parallelism to change its behaviour, can be executed in parallel by the following code. One can think that the methods *m1()* and *m2()* are executed in different threads.

```
interface TM {
    void m1();
    void m2();
}

class Harness {
    public static void runInParallel(final TM tm) {
        Thread t1 = new Thread(new Runnable(){
            public void run() {
                tm.m1();
            }
        });
        Thread t2 = new Thread(new Runnable() {
            public void run() {
                tm.m2();
            }
        }
        t1.start();
        t2.start();
    }
}
```

Harness for executing methods *m1()* and *m2()* in parallel (From [25]).

The method *Harness.runInParallel(·)* is given as an argument one of the classes of the examples that implement the interface *TM* and it runs methods *m1()* and *m2()* in two different threads.

3.2 Examples

3.2.1 Move Method

The refactoring MOVE METHOD is applied on the first example and represents the category of the *dependence edge preserving* refactorings.

```
class C2 implements TM {
    static class A {
        synchronized static void m() {}
        synchronized static void n() {}
    }
    static class B {
    }
    @Override
    public void m1() {
        synchronized (B.class) { A.m(); }
    }
    @Override
    public void m2() {
        synchronized (A.class) { A.n(); }
    }
}
```

Original

```
class C2 implements TM {
    static class A {
        synchronized static void m() {}
    }
    static class B {
        synchronized static void n() {}
    }
    @Override
    public void m1() {
        synchronized (B.class) { A.m(); }
    }
    @Override
    public void m2() {
        synchronized (A.class) { B.n(); }
    }
}
```

Refactored

Move Method introduces a deadlock, when *m1()* locks on *B.class* and *m2()* locks on *A.class* and both threads are blocked on the lock held by the other one (Example from [25]).

In the previous listing we can see an example that shows that a deadlock is introduced after the refactoring. In the original code, a deadlock is impossible because the lock that both methods require is *A.class* and neither of them holds it indefinitely. However, in the refactored version both methods require both locks and there is a scheduling that creates a deadlock.

The error occurs when the method *n()* is moved from class *A* to class *B*, because the lock of *n()* is also changed respectively due to the `synchronized` keyword. As a result, if the method *m1()* locks on *B.class* and *m2()* locks on *A.class*, then both threads will block on the lock the other thread holds, *m1()* is blocked by *A.class* and *m2()* by *B.class*, resulting in a deadlock. In the original code this was not possible since the method *m2()* did not require *B.class* but it required the lock *A.class* which it already had.

3.2.2 Inline Local

The refactoring Inline Local is applied on the following example and represents the category of the *dependence edge preserving* refactorings.

```
class C4 implements TM {
    static volatile boolean a = false;
    static volatile boolean b = false;
    public void m1() {
        boolean x = (a = true);
        while(!b);
        if(x);
        System.out.println("m1 finished");
    }
    public void m2() {
        while(!a);
        b = true;
        System.out.println("m2 finished");
    }
}
```

Original

```
class C4 implements TM {
    static volatile boolean a = false;
    static volatile boolean b = false;
    public void m1() {

        while(!b);
        if((a = true));
        System.out.println("m1 finished");
    }
    public void m2() {
        while(!a);
        b = true;
        System.out.println("m2 finished");
    }
}
```

Refactored

Inline Local introduces a livelock, *m1()* is busy-waiting on *b* and *m2()* on *a*. (Example from [25])

In the listing of the INLINE LOCAL refactoring we can see an example that illustrates how a livelock is introduced after the refactoring. We should note that the original code does not have races because the fields *a* and *b* are `volatile`. The modifier `volatile` ensures that any change of a field is immediately *visible* by other threads, this means that after the variable *a* gets the value **true**, the thread running *m2()* will definitely read the value **true** and not the previous one.

In the original code, a livelock is not possible because *m1()* first turns *a* to **true** and then busy-waits on *b*. As a result, *m2()* moves past the **while**, changes the value of *b* to **true** and prints that it has finished. Consequently, *m1()* moves past the busy-waiting and also prints that it has finished.

However, in the refactored version of the example, both methods are busy-waiting the first one on *a* and the second one on *b*, consequently, both of them are waiting for the other thread to change the value of *a* or *b* but none of them can move past the **while** to actually execute the command, resulting in a livelock.

For this refactoring we provide another example, in which we apply the INLINE LOCAL refactoring but this time we witness a change in behaviour even in sequential execution. In the listing below we can see the refactoring as it is applied by the current implementation of Eclipse. Since there is no need for parallel execution this code does need to be executed in the harness. This issue was detected while experimenting with the intermediate language (see Chapter 4).

```
1  class C6 {
2      public void m1() {
3          boolean y = true;
4          boolean x = (y = true);
5          y = false;
6          System.out.println(x);
7          System.out.println(y);
8      }
9  }
```

Original

```
class C6 {
    public void m1() {
        boolean y = true;

        y = false;
        System.out.println((y = true));
        System.out.println(y);
    }
}
```

Refactored

INLINE LOCAL introduces an inconsistent read. The read of *y* at line 7 in the refactored program reads the assignment at line 6 instead of the one at line 5.

In this example, the original code prints **true** and then **false**. However, when the variable *x* is inlined the assignment of *y* replaces all the occurrences of *x*, resulting in replacing the occurrence of *x* at line 6 and consequently, overwriting the assignment at line 5 and changing the value of *y* back to **true**. The refactored code prints **true** and **true**.

3.2.3 Convert Local Variable To Field

The refactoring CONVERT LOCAL VARIABLE TO FIELD is applied on the following example and represents the category of *introducing new shared state* refactorings.

```
public class C7 implements TM {

        public void m(int caller){
                int x = 0;  //targeted local variable
                for(int i = 0; i < 100000; i++);
                x++;
                System.out.println(caller+ ": I am
                        exiting with x =" + x);
        }
        @Override
        public void m1() {
                m(1);
        }
        @Override
        public void m2() {
                m(2);
        }
}
```

```
public class C7 implements TM {
        private int x; //new field
        public void m(int caller){
                x = 0;
                for(int i = 0; i < 100000; i++);
                x++;
                System.out.println(caller+ ": I am
                        exiting with x =" + x);
        }
        @Override
        public void m1() {
                m(1);
        }
        @Override
        public void m2() {
                m(2);
        }
}
```

Original Refactored

Convert Local Variable To Field introduces races on x.

In the listing of the Convert Local Variable To Field the original code always prints 1. However, in the refactored version, there is no synchronisation between the two threads on how to access x, consequently, there is a race on x and it is possible that the code prints also 0 or 2.

Chapter 4

Synchronised Data Flow Graph

The main challenge in constructing refactoring tools for real programming languages is to capture the semantics for the complete language in a faithful manner. A programming language like Java contains high level features such as classes and inheritance that enhance code maintainability and readability but complicate the semantic analysis of programs. This motivates the mapping of the Java language to an intermediate format which captures the semantics of the program at a lower level and enables the refactoring tool to verify behaviour preservation.

4.1 Capturing JMM with an Intermediate Language

Under the assumption that the data dependencies between variable accesses and the synchronisation dependencies defined by JMM preserve behaviour, we simplify Java code to an intermediate format that captures these dependencies.

The intermediate language we created is called SDFG and it is inspired by OFG [1]. This language strips Java from the structural features and maintains a set of memory access, method calls and data and synchronisation dependencies that reflect the sequence of changes made by the code to the memory.

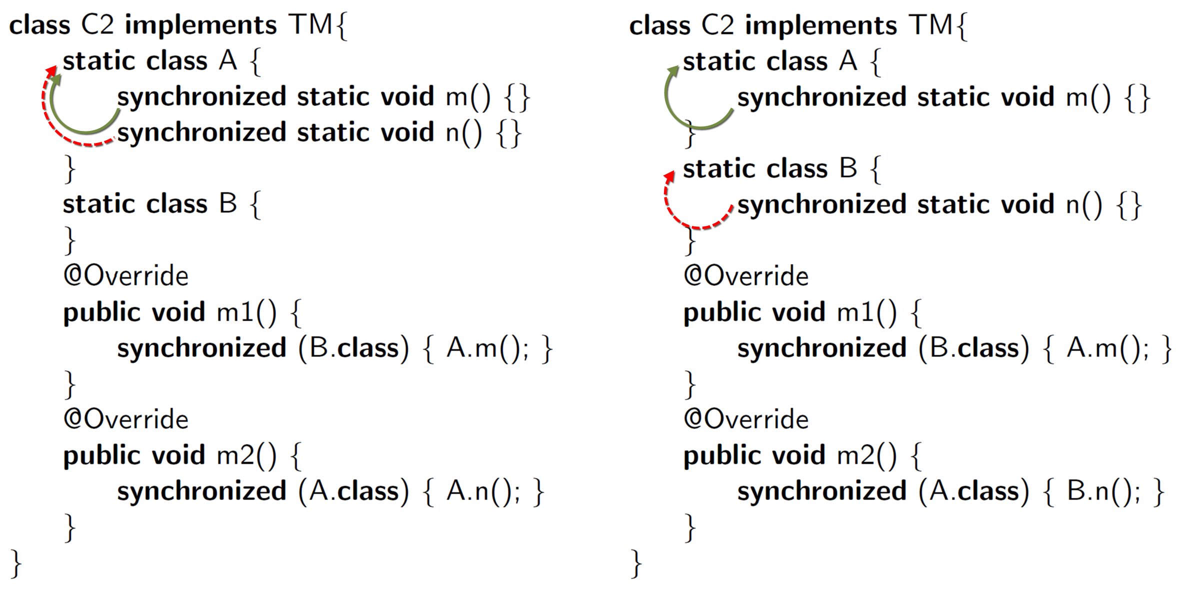
As we saw in Figure 1.1, we only keep reads and writes to memory locations along with the dependencies between them, similarly to a PDG. Figures 4.1 and 4.2 use arrows to demonstrate the hidden dependencies that are made explicit by SDFG, consequently, it is easier to detect the changed dependency that is responsible for the different behaviour of the refactored program. The dashed arrows show the dependencies that are not preserved and indicate why the refactored program exhibits new behaviour.

4.2 Preserving Behaviour with SDFG Mappings

Considering the inline local refactoring, Figure 4.3 illustrates the anticipated changes between the original and refactored SDFG code in dotted lines. In Figure 4.4 that mapping is applied the dashed arrows indicate the changed dependencies that caused the change of behaviour.

The idea is to shift the proof of behaviour preservation to proving the correctness of the inference rules. Given that a dependency is a transitive relation between two `Stmts` we require that the indirect dependencies of the original SDFG program are preserved. To check that, we use the inference rules to map the refactored program to an inferred original SDFG program. Then, we compare the original SDFG program with the inferred one. If the inferred original program is a subset of the original one, we guarantee that no unexpected dependencies are introduced and if the inference rules are correct, no unexpected dependencies are not lost. The anticipated changed dependencies are caught by the inference rules and their correctness lies on proving the correctness of the inference rules.

Consequently, proving the correctness of the refactoring lies on proving the correctness of the mapping that models the anticipated changes. Figure 4.5 illustrates this proof outline. The rounded rectangles represent the parts of the proof that are refactoring specific. The proof outline will be



```
class C2 implements TM{
    static class A {
        synchronized static void m() {}
        synchronized static void n() {}
    }
    static class B {
    }
    @Override
    public void m1() {
        synchronized (B.class) { A.m(); }
    }
    @Override
    public void m2() {
        synchronized (A.class) { A.n(); }
    }
}
```

```
class C2 implements TM{
    static class A {
        synchronized static void m() {}
    }
    static class B {
        synchronized static void n() {}
    }
    @Override
    public void m1() {
        synchronized (B.class) { A.m(); }
    }
    @Override
    public void m2() {
        synchronized (A.class) { B.n(); }
    }
}
```

Figure 4.1: The acquire and release dependencies from method *n()* to *A.class* is replaced with new dependencies to *B.class*.

completed by adding the proof of correctness of all the inference rules per refactoring.

For example, the mapping in Figure 4.3 preserves code behaviour, when Inline Local is applied, because:

- every value assignment to the variable has the original read dependencies which means that the original values are read and used to calculate the assigned value, and
- every variable reads the value of the original assignment.

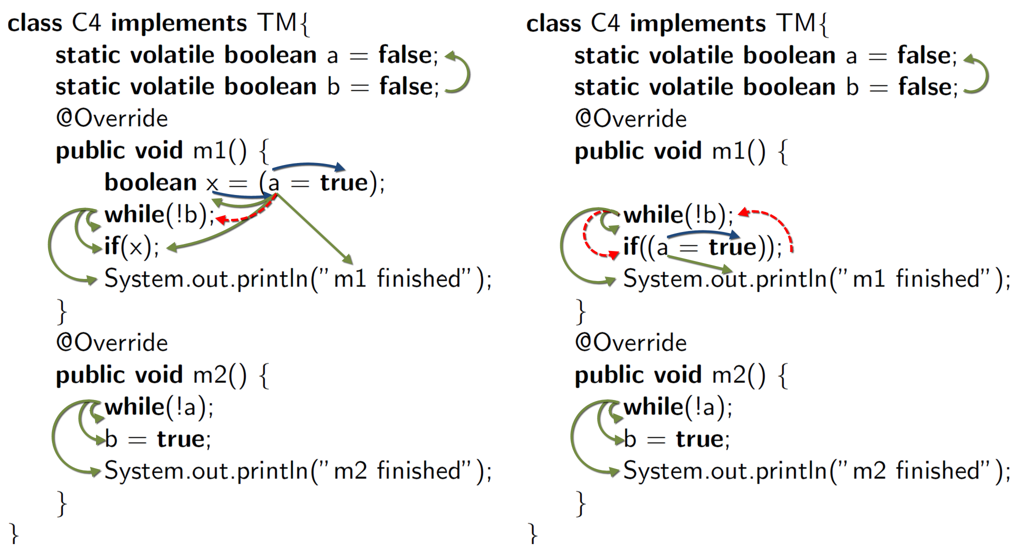
4.3 SDFG in CARR

We discussed how the abstraction of SDFG can be used to compare the memory traces of two different programs and decide if they are equivalent or not. In Figure 1.2 we saw how the SDFG is used inside the CARR algorithm. CARR uses the SDFG transformation of the original and the refactored program as the constraint checker of the refactoring. The verification process of the two SDFG programs is configured by a mapping function. Consequently, CARR uses the SDFG as the common basis to verify the refactorings but uses a different mapping function to specify the process and to satisfy the needs of each refactoring instead of requiring the programs to be identical.

In this way, we overcome the limitations found in the work of Schäfer et al. [25], who used memory trace preservation to prove the correctness of Move Method, which requires that memory trace stays intact after applying the refactoring. Consequently, it is too strict to be extended to other refactorings such as Inline Local; for instance, if the inlined expression contains a field, which is shared memory and corresponds to a JMM action, the memory trace is going to be different due to the relocation of accesses to this field.

From the perspective of the first research question *"When does the current implementation of the targeted refactorings change the behaviour of the refactored program?"*, the SDFG equivalent of the original and the refactored version of the examples in Chapter 3 points out the dependencies that were not taken into account and changed the program's behaviour. For example in the Figures 4.1 and 4.2 we can see the changed dependencies illustrated by dashed arrows.

Furthermore, to answer the last research question, *"Can we prove the correctness of the fixed version?"*, we formalise the mapping function as inference rules (see Chapter 5) and we use separation logic on SDFG `Stmts` in the proof outline (see Chapter 6).



```
class C4 implements TM{
    static volatile boolean a = false;
    static volatile boolean b = false;
    @Override
    public void m1() {
        boolean x = (a = true);
        while(!b);
        if(x);
        System.out.println("m1 finished");
    }
    @Override
    public void m2() {
        while(!a);
        b = true;
        System.out.println("m2 finished");
    }
}
```

```
class C4 implements TM{
    static volatile boolean a = false;
    static volatile boolean b = false;
    @Override
    public void m1() {

        while(!b);
        if((a = true));
        System.out.println("m1 finished");
    }
    @Override
    public void m2() {
        while(!a);
        b = true;
        System.out.println("m2 finished");
    }
}
```

Figure 4.2: The acquire dependency from the `volatile` read of *a* is lost and two new dependencies are added, a release dependency from `volatile` write to *a* to the `volatile` read of *b* and an acquire dependency from `volatile` read of *a* to the `volatile` read of *b*.

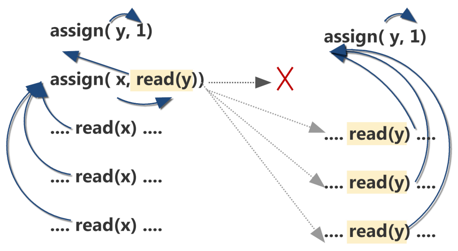


Figure 4.3: The variable x is the variable to be inlined, the actions *assign* and *read* and regular that represent the dependencies can be thought of as the invariant. The grey dotted arrows express the mapping function.

4.4 SDFG Language

A program defined in SDFG consists of a set of the declarations of the fields and methods of the Java program and a set of the reachable statements (based on control flow analysis) and their dependencies. In general, an SDFG program captures the PDG and the JMM of a program. The language is defined in the following Rascal listing.

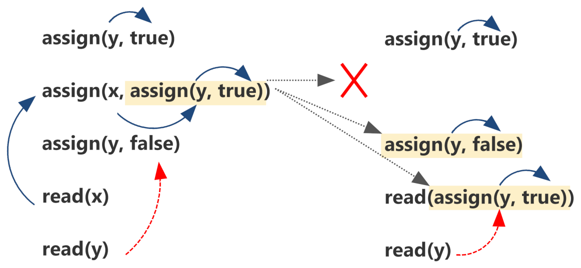


Figure 4.4: The invariant preservation that was demonstrated in Figure 4.3 is violated by this refactoring due to the changed dependency illustrated by the dashed arrow.

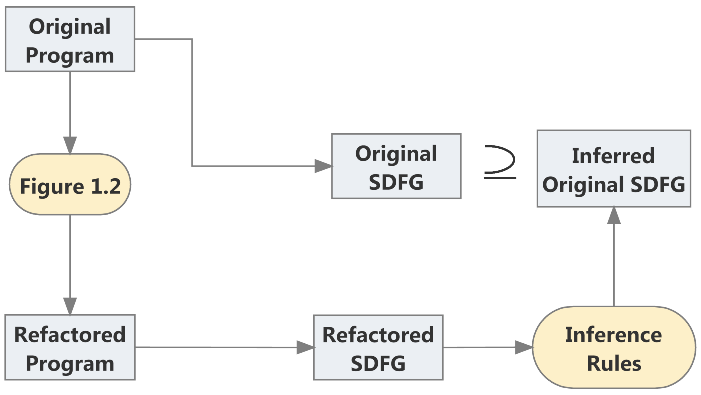


Figure 4.5: For every program, we require that the inferred SDFG program is a subset of the original one. The rounded rectangles represent the parts of the proof that are refactoring specific.

```
data Program = program(set[Decl] decls, set[Stmt] stmts);

data Decl
    = attribute(loc id, bool volatile)
    | method(loc id, list[loc] formalParameters, loc lock)
    | constructor(loc id, list[loc] formalParameters)
    ;

data Stmt
    = read(loc id, loc variable, loc depId) | assign(loc id, loc variable, loc depId)
    | change(loc id, loc typeDecl, loc dataDepId)

    | acquireLock(loc id, loc lock, loc depId) | releaseLock(loc id, loc lock, loc depId)

    | create(loc id, loc constructor, loc actualParameterId)
    | call(loc id, loc receiver, loc method, loc actualParameterId)

    | entryPoint(loc id, loc method) | exitPoint(loc id, loc method)
    ;
```

All `Stmts` are identified by their location in the original Java source code. Additionally, all entities, such as variables, fields, classes and methods are fully identified by the qualified name of their declaration.

SDFG Semantics

The `Stmts` of the SDFG language reflect the dependencies between memory accesses from the original Java code. The dependencies include the data and synchronisation dependencies from the JMM but they are applied to local memory in addition to the shared one, as well as other dependencies that reflect memory allocation and the beginning and ending of methods.

Each `Stmt` is a binary relation between two `Stmts` or between a `Stmt` and a value expressing a direct dependency. The values represent independent data values such as numbers or strings.

Data Dependencies

A `read`(·) represents reading the value of the variable defined by the variable declaration. It has data dependency to a visible write or to another `read`(·) that is needed to access this memory location. For example, the `read`(·) of an array element will have a dependency edge on the `read`(·) of the index.

An `assign`(·) represents writing on the variable defined by the variable declaration. An `assign`(·) has a data dependency to either (i) a `read`(·) that is part of the expression to be assigned or (ii) an independent value or (iii) a `read`(·) that is needed to access this memory location.

A `change`(·) represents the change of a class as a side effect of a method call or a field change. The JMM defines reads and writes on memory locations which completely defines data dependencies for primitives. However, aliasing in Java can hide data dependencies from the JMM abstraction [9, 28].

Synchronisation Dependencies

An `acquireLock`(·) represents either a `volatile` read or the entrance in a `synchronized` block. The `acquireLock`(·) is connected with the identifier of a subsequent `Stmt`.

A `releaseLock`(·) represents either a `volatile` write or the exit from a `synchronized` block. The `releaseLock`(·) is connected with the identifier of a `Stmt` that proceeded its occurrence.

Other Dependencies

`call`(·) & `create`(·) represent a method call and a constructor call respectively. They have data dependency to the `read`(·) of an actual parameter. The `call`(·) also depends on the `read`(·) of the receiver.

`entryPoint`(·) & `exitPoint`(·) represent the beginning and the end of a method. They can be referenced by a synchronisation dependency to ensure that no synchronisation edges are lost even if the method has no code except for the synchronisation structure.

The order of the `Stmts` is solely defined by the dependencies between them. The data and control flow are taken into account at the conversion step and are fully reflected by the dependencies. The analysis is inter-procedural.

In the following listing we see how implicit dependencies from a snippet of Java code from class $C4$ are made explicit in SDFG.

```
while(!b);
if((a = true));
```

```
read(r1, b, _), acquireLock(r1, b, r2), acquireLock(r1, b, a2),
read(r2, a, a2), assign(a2, a, true), releaseLock(a2, a, r1)
```

Java SDFG

The variables a and b are volatile so there are implicit synchronisation dependencies in Java code. On the other hand, the synchronisation dependencies are explicitly marked by `acquireLock`(·) and `releaseLock`(·).

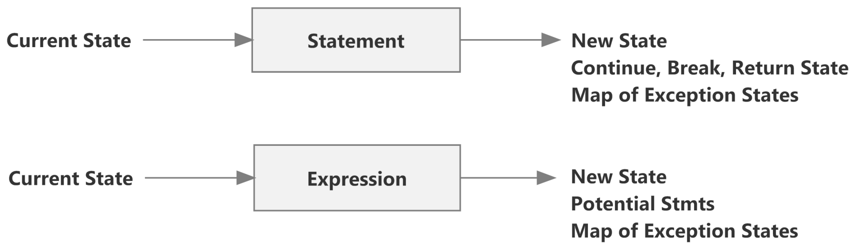


Figure 4.6: To extract the dependencies, the algorithm needs the current state at this point, and then it returns the updated state, the exception states and the potential `Stmts` or the state for redirection (**continue**, **break** and **return**) in case of expressions and statements respectively.

Dependency Extraction

The most challenging part of the converter from Java to SDFG was the extraction of the dependencies. To extract the implicit dependencies of every statement and expression we needed to maintain a program state that represents the dependencies and the gathered `Stmts` at that specific node of the AST. The state contains:

- A set of the `Stmts` that were gathered in that specific path.
- A map with the identifiers of the last visible `assign`(·) or `change`(·) of a variable declaration.
- A second map that contains the identifiers of the last change of a class which corresponds at the last visible `change`(·).
- A relation that contains a tuple with the lock declaration and the identifier of an acquire action.

The converter traverses the AST and in every node it carries the current state which is updated and returned. Along with the updated state, depending on the type of the AST node additional information is returned as it is illustrated by Figure 4.6.

The two types of AST nodes that the converter visits are the expressions and the statements. Both types return the exception state, which maps an exception with the current state and keeps it until a **catch** statement is found. In case of an expression, the converter returns a set of potential `Stmts` which refer a potential read of the current variable. Finally, in case that a statement is **continue**, **break** or **return** the current state is saved and returned to its parent node until it is time to be used. For instance, a state stored by finding a **break** statement is stored until exiting a loop or a **case** from a **switch** statement.

4.5 Advantages & Disadvantages

One of the advantages of this language is its simplicity, it reflects all the dependencies, data and synchronisation, using binary relations between two `Stmts` and the ordering of the `Stmts` is solely defined by the dependencies.

Additionally, it is an independent programming language allowing to save or to further manipulate the SDFG program independently of the original Java program.

Furthermore, SDFG includes all the information from a control and data flow analysis needed for sequential refactorings enhanced by the dependencies introduced by the JMM actions needed for the concurrency awareness. For this reason, it is suitable for the analysis of programs that requires to respect both aspects.

However, the SDFG performs inter-procedural analysis. As a result, the dependencies that are captured by the SDFG are limited to the method's scope. Although information about the method invocation is kept, the dependencies are not resolved. For example, in the Inline Local refactoring, if the inlined expression contains a method call we need to check if the method has synchronisation dependencies because then the acquire actions defined in that method have dependencies with the statements after the method call, and the release actions have dependencies with the statements that were executed before the method call. In this case and since the Inline Local does not allow new acquire or release dependencies, as it is defined by the inference rules 5.21 introduced in Chapter 5, the refactoring is rejected.

Secondly, the simplicity of the binary relations between the `Stmts` can also become a disadvantage since the number of the `Stmts` increases quickly. For example, an assignment that depends on reading values from five variables will be represented by five different `assign`(·) each one of them referring to a different `read`(·).

The increased number of `Stmts` and the decision to keep the `Stmts` in a set also increases the search time making it difficult to perform tasks on large SDFG programs. However, indexing the `Stmts` by their identifier before applying a task speeds up the search and make the analysis more scalable. We use this tactic on the Inline Local refactoring in which we need to match a sequence of `Stmts` to another one as shown by the inference rule 5.18.

Finally, although `read`(·) and `assign`(·) are enough to fully express the dependencies between primitives, they do not capture the dependencies of objects in Java. This occurs because the aliasing and the mutability of objects enables different variables to point to the same object and the changes that are performed on that object through one variable are read by accessing the object from the other. In order to make SDFG safer we introduced `change`(·) which captures the changes on a class type and propagates the changes to all the instances of this class. The drawback of this approach is that the SDFG records more dependencies than they really exist. However, resolving the dependencies in a context aware way was out of the scope of this work due to its complexity [9].

4.6 Limitations

As we mentioned in the previous paragraph aliasing in Java hides dependencies from SDFG and although the SDFG uses a pessimistic way to calculate the dependencies by adding dependencies to every instance of a class, it is still possible for the SDFG to miss dependencies. This happens because the analysis is inter-procedural, as a result, changes on classes inside a method do not create data dependencies to accesses from the environment from which it was called. To fix this without changing the inter-procedural analysis, we can keep a set of the changed types from all the versions of a method and when this method is called to add a `change`(·) for every type. This way we do not lose the dependencies but we over-estimate them to be safe.

Another issue of the current implementation is exception handling. Although the code handles exceptions, it can only account for the exceptions that are known. Currently, only the exceptions from **throw** and method declarations are gathered. To fix that, the algorithm needs to extract the exceptions thrown by all the methods that are called independently to where they were defined. Consequently, if we extract the exceptions from the jar files and the standard Java library, the correct dependencies will be extracted for all **try-catch** statements.

4.7 Claims

We claim that SDFG extracts the data and synchronisation dependencies of Java programs with respect to data and control flow analysis and the JMM. Through the small test cases in Chapter 6 we will provide evidence that special cases such as **break**, exceptions and **finally** work. Our implementation does not work with Java labels.

Furthermore, we claim that this language can be used as constraint in a behaviour preservation framework given a mapping function. This claim is supported by the extension of the tool to include also the Convert Local Variable To Field refactoring which was not part of the JRRT [25].

Chapter 5

Concurrency Aware Refactoring with Rascal

The general algorithm used in CARR implementation consists of the following steps that were illustrated in Figure 1.2, the steps 1 and 2 correspond to the circle with label **Apply the Refactoring**.

1. Check the pre-conditions described in Section 2.1
2. Apply the refactoring, generating new identifiers when needed and keep a map with the correspondence between the original and the new identifiers.
3. Convert the original and the refactored program to SDFG.
4. Apply the mapping rules on the transformed programs and either reject or accept the refactoring.

Steps 1, 2 and 4 are configured depending on the refactoring, these configurations are described in the following sections.

5.1 Move Method

There are no special requirements other than the ones described in Section 2.1 enforced by the CARR implementation of the Move Method refactoring. The algorithm is described in Rascal.

```
data MethodCase = static(loc decl, loc receiver)
                | inParameters(loc decl, loc index)
                | inFields(loc decl, loc fieldExp, loc param)
                | notTransferable();

set[Declaration] moveMethod(set[Declaration] ast, loc methodDecl, loc destinationClassDecl){
    method = getMethodFromDecl(ast, methodDecl);
    sourceClass = getClassFromDecl(ast, extractClassName(methodDecl));
    destinationClass = getClassFromDecl(ast, destinationClassDecl);

    methodConfig = getMovedMethodConfiguration(sourceClass, destinationClass, targetMethod);

    if(notTransferable(methodConfig)){
        println("The refactoring cannot be applied");
        return ast;
    }

    method = desugarSynchronizedMethod(method);
    method = desugarFieldAccess(method);
    method = adaptMethodsCode(methodConfig, method);

    refactoredAst = visit(ast){
        case c:class(_,_,_,_):{
            if(c@decl == sourceClass@decl){
                insert removeMethodFromClass(c, methodDecl);
            }
            else if(c@decl == destinationClass@decl)
                insert addMethodToClass(c, targetMethod);
        }
    }
    refactoredAst = visit(refactoredAst){
        case m:methodCall(_,_,_,_):{
            if(m@decl == methodDecl)
                insert adaptMethodCall(methodConfig, m);
        }
    }

    //Convert to SDFG
    p = convertSDFG(ast);
    pR = convertSDFG(refactoredAst);

    if(checkTheInvariant(p,pR, methodconfig, methodDecl)){
        println("Refactoring Move Method successful!");
        return refactoredAst;
    }
    else{
        println("Refactoring failed!");
        return ast;
    }
}
```

The refactoring algorithm first determines if and how a method can be moved. According to the case the necessary information is kept in the configuration such as the parameter name, the new method declaration, etc.

Before applying the refactoring we preprocess the code of the method by desugaring **this**, which means that all the accesses to fields and method calls of the enclosing class will be referenced by the identifier **this** or the qualified name if the method is static. Although this is not mandatory it helps in the constraint check. Then, to capture the case when the method was `synchronized` the keyword is desugared in order to be correctly updated in the next step.

After the desugaring steps, the code of the method is updated to match the new method declaration, for example, all the accesses to fields are now accessed through the qualified name, if they are static, or through the new parameter. Then, the method is moved. After that, the AST is revisited and the method calls to this method are changed to match the new method declaration. Finally, the SDFG of the refactored and the original program are matched based on the mapping rules defined in the following section.

Limitations. When the Move Method refactoring uses parameter swapping and field access to move the method there is no guarantee that the parameter or the field is not **null**. This issue is found in the implementations of both CARR and Eclipse. Although we can check if the parameter used to access the method is actually the keyword **null**, we cannot provide any stronger guarantee. As a partial solution for this problem, we suggest providing the option of adding assertions before every method call to reassure that the developer is notified when this bug occurs or adding the annotation *@NotNull*. However, this simple feature is not yet implemented.

5.1.1 Mapping Rules

The Move Method refactoring first applies a transformation at the method code and then applies another transformation to the rest of the code and the updated method code. Consequently, the rules concerning the refactored code of the method are:

- There should be no `assign(·)` that refers to **this** or to the field of the class used as receiver in case of parameter swap and field access respectively.
- The fields should have read dependencies to the new parameter or the qualified name.
- In case that the destination class was a parameter, the dependencies to the parameter should be replaced by dependencies to **this**.
- The receiver of a method call should not be the value **null**.

Then, CARR requires that every method call to this method is updated to match its new declaration.

The previous rules are formalized in the following inference rules. The inference rules model how the `Stmt`s from the refactored SDFG program can be used to generate the original program. The inference rules will help us answer the third research question *"Can we prove the correctness of the fixed version?"*.

Provided that the second research question is answered by CARR we need to prove that this implementation is correct, specifically, we need a way to prove that the refactored program does not change the behaviour of the original one. To accomplish that, we start from the refactored program and we invert the refactoring using the inference rules. If the inferred program matches the original one then the refactoring is semantically safe.

In the following rules *methodDecl* is the original method declaration and *newMethodDecl* is the refactored one. The next rules apply to all the cases. The inference rules in which the premise and conclusion are the same are ommited. RP and IOP stand for Refactored Program and Inferred Original Program respectively.

$$\frac{\text{RP} \models \text{entryPoint(id, newMethodDecl)}}{\text{IOP} \models \text{entryPoint(id, methodDecl)}}, \frac{\text{RP} \models \text{exitPoint(id, newMethodDecl)}}{\text{IOP} \models \text{exitPoint(id, methodDecl)}} \tag{5.1}$$

Static Methods

Inference rules for the method code:

$$\frac{\text{RP} \models \text{read(id, destinationClassId, dep)}}{\text{IOP} \models \text{read(id, sourceClassId, dep)}} \tag{5.2}$$

Inference rules for the whole SDFG program:

$$\frac{\text{RP} \models \text{call(id, destinationClassId, newMethodDecl, argId)}}{\text{IOP} \models \text{call(id, sourceClassDecl, methodId, argId)}} \tag{5.3}$$

Swap Parameters

The following inference rules refer to the method code, where *newPar* refers to the new parameter of the *sourceClass* and *parDecl* to the declaration of the original parameter. The command **abort** represents the immediate rejection of the refactoring transformation due to an invalid `Stmt` with respect to Java semantics, for instance, an assignment to **this**.

$$\frac{\text{RP} \models \text{read(id, } \textbf{this}\text{, dep)}}{\text{IOP} \models \text{read(id, par, dep)}} \tag{5.4}$$

$$\frac{\text{RP} \models \text{assign(id, } \textbf{this}\text{, dep)}}{\text{IOP} \models \textbf{abort}} \tag{5.5}$$

$$\frac{\text{RP} \models \text{read(id, newPar, dep)}}{\text{IOP} \models \text{read(id, } \textbf{this}\text{, dep)}} \tag{5.6}$$

$$\frac{\text{RP} \models \text{acquireLock(id, } \textbf{this}\text{, dep)}}{\text{IOP} \models \text{acquireLock(id, par, dep)}}, \frac{\text{RP} \models \text{releaseLock(id, } \textbf{this}\text{, dep)}}{\text{IOP} \models \text{releaseLock(id, par, dep)}} \tag{5.7}$$

$$\frac{\text{RP} \models \text{acquireLock(id, newPar, dep)}}{\text{IOP} \models \text{acquireLock(id, } \textbf{this}\text{, dep)}}, \frac{\text{RP} \models \text{releaseLock(id, newPar, dep)}}{\text{IOP} \models \text{releaseLock(id, } \textbf{this}\text{, dep)}} \tag{5.8}$$

Inference rules for the whole SDFG program, where *rec* is the original receiver of the method, *arg* is the parameter being swapped and *a* all the other arguments of the method call:

$$\frac{\text{RP} \models \text{call(id, argId, newMethodDecl, recId)}}{\text{IOP} \models \text{call(id, recId, methodDecl, argId)}} \tag{5.9}$$

$$\frac{\text{RP} \models \text{call(id, argId, newMethodDecl, aId)}}{\text{IOP} \models \text{call(id, recDecl, methodDecl, aId)}} \tag{5.10}$$

Field Access

Inference rules for the method code, where *newPar* refers to the new parameter of the *sourceClass* and *f* refers to the field used as the receiver of the class:

$$\frac{\text{RP} \models \text{read(id, newPar, dep)}}{\text{IOP} \models \text{read(id, } \textbf{this}\text{, dep)}} \tag{5.11}$$

$$\frac{\text{RP} \models \text{read(id, newPar, \_), assign(fId, f, id)}}{\text{IOP} \models \textbf{abort}} \tag{5.12}$$

Inference rules for the whole SDFG program, where *f* is the field that is used as a receiver of the method call:

$$\frac{\text{RP} \models \text{read(fId, f, recId), call(id, fId, newMethodDecl, aId), call(id, fId, newMethodDecl, recId))}}{\text{IOP} \models \text{call(id, recId, methodDecl, aId)}} \tag{5.13}$$

5.1.2 Motivating Example Revisited

Considering the mapping rules and how the `synchronized` keyword works we can answer the first research question concerning the example of MOVE METHOD from Chapter 3. As we saw in Figure 4.1 the deadlock is introduced because the synchronisation dependencies `acquireLock`(·) and `releaseLock`(·) from the refactored SDFG have different lock declarations than the ones from the original SDFG, as a result the original graph cannot be reproduced by the inference rules.

Schäfer et al. [25] use the desugaring step to fix the implementation. The same approach is also used by CARR, the result of our tool is shown in the following listing. A similar case is examined in Chapter 6.

```
class C2 implements TM {
    static class A {
        synchronized static void m() {}
    }
    static class B {
        static void n() {
            synchronized(A.class){
            }
        }
    }
    @Override
    public void m1() {
        synchronized (B.class) { A.m(); }
    }
    @Override
    public void m2() {
        synchronized (A.class) { B.n(); }
    }
}
```

Move Method, the example refactored by CARR. By desugaring the `synchronized` keyword before applying the refactoring the right lock is determined and preserved through the refactoring.

In the listing above we see the effect of the desugaring step. By desugaring the `synchronized` keyword the `synchronized` block is explicitely protected by *A.class*. While applying the refactoring the lock stays unaffected since it has no dependencies with the class that the method is defined in. After moving the method, the lock from the `synchronized` block is not the same with the class object and the `synchronized` keyword cannot be "resugared". However, the expected result is accomplished and the synchronisation dependencies are preserved since the block is `synchronized` explicitly on *A.class* and not implicitly on the receiver class.

5.2 Inline Local

In order to perform the Inline Local refactoring, CARR requires that the declaration of the local variable is inside a block to perform this refactoring. As we previously mentioned, this implementation is broader than the implementation of Eclipse. By broadening this refactoring, we demonstrate that SDFG allows the safe refactoring of cases that would have been rejected from pre-conditions.

At first, CARR inlines every occurrence of the local variable with a *potential* expression without checking anything apart from moving `synchronized` methods, then the constraint check determines if the refactoring is valid.

The characterisation of the inlined expression as "potential" is needed because if the assignments of the local variable exist in different paths that later merge, both assignments could be inlined. In these cases, one expression is chosen and then the verification process checks if both expressions were needed or not. If both expressions are visible by a specific read then the refactoring is rejected.

```
set[Declaration] inlineLocal(set[Declaration] ast, loc localDecl){
    methodDecl = getMethodDeclFromVariable(localDecl);
    inlineExpr; //keeps the expression to be inlined
    replacementIds; //maps the original ids with the new ones

    p = convertSDFG(ast);
    synchronizedMethods = gatherMethodsWithsynchronisationEdges(p);

    //Apply the refactoring
    refactoredAst = visit(ast){
        case m:method(_, _, _, _, body):{
            if(m@decl == methodDecl){
                insert inlineLocal(m, local);
            }
        }
    }

    //Convert to SDFG
    pR = convertSDFG(refactoredAst);

    if(checkTheInvariant(p, pR, local, replacementIds)){
        println("Refactoring Inline Local successful!");
        return refactoredAst;
    }
    else{
        println("Refactoring failed!");
        return ast;
    }
}

Declaration inlineLocal(Declaration m:method(_, _, _, _, body), loc local){
    loc innerBlock = findInnerBlock(body,local);
    visit(body){
        case b:block(_):{
            if(innerBlock == b@src)
                insert inlineLocal(b, local, Expression::null());
        }
    }
}
```

The algorithm first locates the method and the most inner block that contains the declaration of the local variable and visits the statements.

The statement visitor respects the control flow of the program using ideas similar with the ones used at the SDFG. In this case, instead of the state of the program, we keep the expression to be inlined. Attention should be paid on the statements that convert an expression to a statement because these statements may contain the assignments of the local variable that are omitted from the refactored program.

The expression visitor is simpler; depending on the type of access of the variable, the visitor either replaces the variable with the current expression, or updates the current expression.

Limitations. Since the algorithm of this refactoring is similar to the SDFG converter algorithm, it shares the same limitations. If an exception being thrown is not associated with a **throw** statement or a method declared in the program the algorithm cannot detect correctly the expression to be inlined; consequently, the refactoring fails.

The rules discussed in the following section ensure that if the type of the local variable is a primitive it will evaluate in the same value. However, in case that the expression contains objects we cannot guarantee that another thread or the inner code of a method call will not change the value of this field, resulting in the inlined expressions evaluating to different values even if the data flow is preserved. This problem is a limitation of the SDFG analysis as it was listed in Section 4.6.

5.2.1 Mapping Rules

The code in the previous paragraph applies the refactoring without applying any pre-conditions, this means that the correctness of the refactoring solely lies on the SDFG constraint check. Although code rearrangement refactorings change the code of a method and, consequently, the SDFG of the original and the refactored code cannot be mapped in an one to one relationship, we can use the transitive property of dependencies and require that the indirect dependencies are preserved. Consequently, regarding the INLINE LOCAL refactoring, we enhance the rules introduced from *dependence edge preservation* [25] and we require that:

- All `assign`(·) `Stmts` that refer to the local variable are removed.
- `Stmts` that did not refer to the local variable or to an assignment in the inlined expression should not refer to any of the new identifiers.
- All `Stmts` of the inlined expression should either refer to the same identifier of the original program or to an identifier that replaced the original identifier in this specific inlined expression.

The only issue we cannot check using SDFG as a constraint is the synchronisation dependencies that change when a method is moved and that is a limitation of the inter-procedural analysis of the SDFG. For this reason, we gather the `synchronized` methods using the invocation graph from the *m3* model of Rascal[1] and extracting the directly `synchronized` methods from the SDFG representation and then we check if a method contains synchronisation dependencies while performing the refactoring.

The following inference rules express formally what we discussed in this section. The declaration of the local variable is represented by x and all the other declarations are represented by the letters v and r for variables and m and c for methods and constructors respectively.

The function *f(id)* maps the identifiers of the `Stmts` in the original and the refactored program; as input it gets the identifier from the refactored program and returns the original one. The function domain does not contain the identifiers from the assignments of the local variable, so it is not defined on these values.

Furthermore, we will use another helping function *inlinedFor(id)* which takes as input a new identifier and returns the identifier of the `read`(·) of the inlined local variable that it has replaced. To ensure that a new identifier has the correct dependencies we introduce a predicate *verifyDependency(id, dep)* that takes two identifiers and evaluates to true if the dependency is acceptable, which means that there is no dependency between two different inlined expressions.

$$\frac{\text{RP} \models \text{read(id, v, depId), verifyDependency(id, depId)}}{\text{IOP} \models \text{read}(f\text{(id), v, }f\text{(depId))}} \tag{5.14}$$

$$\frac{\text{RP} \models \text{assign(id, v, depId)}}{\text{IOP} \models \text{assign}(f\text{(id), v, }f\text{(depId))}} \tag{5.15}$$

$$\frac{\text{RP} \models \text{call(id, rId, m, argId)}}{\text{IOP} \models \text{call}(f\text{(id), }f\text{(rId), m, }f\text{(argId))}} \tag{5.16}$$

$$\frac{\text{RP} \models \text{create(id, c, argId)}}{\text{IOP} \models \text{create}(f\text{(id), c, }f\text{(argId))}} \tag{5.17}$$

$$\frac{\text{RP} \models \text{assign(uId, u, depId)}}{\text{IOP} \models \text{assign}(f\text{(uId), u, xRId), read(xRId, x, xAId), assign(xAId, x, }f\text{(depId))}} \tag{5.18}$$

$$\frac{\text{RP} \models \text{call(mId, rId, m, depId)}}{\text{IOP} \models \text{call}(f\text{(mId), }f\text{(rId), m, xRId), read(xRId, x, xAId), assign(xAId, x, }f\text{(depId))}} \tag{5.19}$$

$$\frac{\text{RP} \models \text{create(cId, c, depId)}}{\text{IOP} \models \text{create}(f\text{(cId), c, xRId), read(xRId, x, xAId), assign(xAId, x, }f\text{(depId))}} \tag{5.20}$$

[1] https://github.com/cwi-swat/rascal/blob/master/src/org/rascalmpl/library/lang/java/m3/Core.rsc

Since the function *f(·)* is not defined on x's identifiers the following rules produce all the synchronisation dependencies apart from the x's ones. However, since the function is not applied on the identifier of the `Stmt`, if a dependency is lost or introduced it will introduce an inconsistency between the two versions and the refactoring will be rejected.

$$\frac{\text{RP} \models \text{acquireLock(id, l, depId)}}{\text{IOP} \models \text{acquireLock(id, l, } f\text{(depId))}}, \frac{\text{RP} \models \text{releaseLock(id, l, depId)}}{\text{IOP} \models \text{releaseLock(id, l, } f\text{(depId))}} \tag{5.21}$$

$$\frac{\text{RP} \models \text{acquireLock(id, l, depId)}}{\text{IOP} \models \text{acquireLock(id, l, xId), read(xid, x, xAId), assign(xAId, x, } f\text{(depId))}}, \tag{5.22}$$

$$\frac{\text{RP} \models \text{releaseLock(id, l, depId)}}{\text{IOP} \models \text{releaseLock(id, l, xId), read(xid, x, xAId), assign(xAId, x, } f\text{(depId))}} \tag{5.23}$$

5.2.2 Motivating Example Revisited

Considering the synchronisation dependencies we can answer the first research question regarding the examples of Inline Local from Chapter 3.

The first example that introduces a livelock is not correct because the `acquireLock`(·) that expresses an acquire dependency between b and a and the `releaseLock`(·) that expresses a release dependency between a and b are generated by the inference rules and they do not exist in the original graph. Also the `acquireLock`(·) that expresses the dependency between a and b is lost in the refactored program. In Figure 4.2 these `Stmts` are illustrated by the dashed arrows.

The second example contains an inconsistent read, the incositency is detected from the constraint check because the rule 5.14 produces a `read`(·) for line 7 that has a data dependency with the `assign`(·) of y at line 6 instead of the one at line 5 (see Figure 4.4).

The answer to the second question concerning this refactoring is that we cannot do something to refactor these examples and preserve the program's semantics. However, we claim that CARR works correctly since it rejects the refactorings.

5.3 Convert Local Variable To Field

This refactoring introduces new shared state, as a result, the refactoring itself needs to restrict the accesses at that field to be executed without interferences from other threads. We implemented this refactoring following the algorithm below, a requirement of this implementation is that the declaration of the local variable is inside a block.

```
AST convertLocalToField(ast, local) {
    <classDecl, methodDecl, newFieldDecl, lockDecl> = findDeclarations(local);

    refactoredAst = visit(ast) {
        case c:class(_, _, _, body):  {
            if(c@decl == classDecl) {
                body = for(m:method(r,n,p,e, block) <- body) {
                    if(m@decl == methodDecl) {
                        block = synchronize(
                                extractBlock(
                                updateDeclaration(
                                splitInitialization(
                                findInnerBlock(block, local)))));
                        insert method(r,n,p,e, block);
                    }
                }
            }
            body = [newLock]
                 + [newField]
                 + body;
            //add the class to the AST with the updated body

        }
    }

    //Convert to SDFG
    p = convertSDFG(ast);
    pR = convertSDFG(refactoredAst);

    if(checkTheInvariant(p,pR, local, replacementIds)) {
        println("Refactoring Convert Local To Field successful!");
        return refactoredAst;
    }
    else{
        println("Refactoring failed!");
        return ast;
    }
}
```

The code above finds the most inner block that contains the declaration of the variable, splits the declaration from the initialization expression, if there is one, and removes the declaration. Afterwards, it extracts a block with all the accesses of the variable and changes the variable declaration of the accesses to the field declaration. Then, it adds the `synchronized` keyword to the new block associated with the new lock. The initialization of the new lock in the constructor and the modifier **final** are important because otherwise the lock could change allowing different threads to interleave. Finally, the new fields are added to the class.

This algorithm refers to non-static methods. Static methods are handled in a similar way; the differences are that the new field is also static and the lock instead of being a new field, is the class object of a new inner class defined for this purpose.

Limitations. A limitation of this refactoring lies on the fact that the new lock introduces new synchronisation dependencies. However, we argue that it cannot introduce a new deadlock, but the following example shows that it can introduce a livelock.

```
public class C8 implements TM {


    volatile boolean a;
    public void m() {

        int x = 0;
        a = false;
        while(a);
        x++;

    }
    public void n() {
        a = true;
    }
    @Override
    public void m1() {
        n();
        m();
    }
    @Override
    public void m2() {
        m();
    }
}
```

```
public class C8 implements TM {
    private final generated_lock_for_x = new Object();
    private int x;
    volatile boolean a;
    public void m() {
        synchronized(generated_lock_for_x){
            x = 0;
            a = false;
            while(a);
            x++;
        }
    }
    public void n() {
        a = true;
    }
    @Override
    public void m1() {
        n();
        m();
    }
    @Override
    public void m2() {
        m();
    }
}
```

Original Refactored

The CARR implementation of Convert Local Variable To Field introduces a livelock.

A livelock is not possible in the original $C8$ program because even if n turns a to **true** the method call that follows turns a back to **false** and both threads can move past the busy-waiting point. However, after applying Convert Local Variable To Field as it is implemented by CARR, we can observe a livelock in the following scheduling, *m2()* starts and turns a to **false**. In the meantime, *m2()* calls *n()* which turns a back to **true**. Then *m()* busy-waits on a and *m1()* is blocked by the new lock and waits for *m2()* to release the lock.

On the other hand, a deadlock cannot be introduced because the new lock is used only in one place and is either nested in another synchronisation block or it contains another synchronisation block, consequently, the order of the lock acquisition is the same as in the original program. For example, if the new `synchronized` block is contained in another `synchronized` block, and there was no deadlock, a second thread would be first blocked by the first lock, and not by the new lock, since the only code that could be blocked on the new lock also needs the first lock.

5.3.1 Mapping Rules

In this category of refactorings an one to one mapping between the original and the refactored SDFG is also not possible because we introduced a new `synchronized` environment that locks on a new field that we introduce and is not used anywhere else. Consequently, the rules we require in order to accept this refactoring are:

- The only new synchronisation dependencies concern the new lock.
- Every node that contained an access of the local variable will be mapped in a new node with the same id but with the declaration of the field instead of the declaration of the variable.
- All access of the new field should have both an acquire and a release dependency with the new lock.
- No synchronisation dependencies should be lost.

We formalize the previous guidelines to the following inference rules, in which the declaration of the local variable is represented by x and the declaration of the field by x_f, the declaration of the lock is $lock_x$, all the other declarations are represented by the letter v for variables and m and c for methods.

$$\frac{\text{RP} \models \text{acquireLock(lId, } lock_x\text{, id), read(id, } x_f\text{, depId), releaseLock(lId, } lock_x\text{, id)}}{\text{IOP} \models \text{read(id, x, depId)}} \tag{5.24}$$

$$\frac{\text{RP} \models \text{acquireLock(lId, } lock_x\text{, id), assign(id, } x_f\text{, depId), releaseLock(lId, } lock_x\text{, id)}}{\text{IOP} \models \text{assign(id, x, depId)}} \tag{5.25}$$

New synchronisation dependencies are introduced by the new lock. Dependencies are added by the `synchronized` block, so we require that the identifier of the of the synchronisation dependencies concerning the new lock is unique. Finally, it is possible that new synchronisation dependencies are added in the constructor when the new lock is initialised.

$$\frac{\text{RP} \models \text{acquireLock(lId, } lock_x\text{, \_)}}{\text{IOP} \models}, \frac{\text{RP} \models \text{releaseLock(lId, } lock_x\text{, \_)}}{\text{IOP} \models} \tag{5.26}$$

$$\frac{\text{RP} \models \text{acquireLock(\_, \_, lId)}}{\text{IOP} \models}, \frac{\text{RP} \models \text{releaseLock(\_, \_, lId)}}{\text{IOP} \models} \tag{5.27}$$

5.3.2 Motivating Example Revisited

Considering the change of the variable declaration to field declaration we can answer the first research question regarding the example of Convert Local Variable To Field from Chapter 3. The refactoring has introduced races because accesses to the field from different threads can now interleave.

To answer the second question we suggest that the refactoring should ensure that the accesses to this field are protected and always run sequentially, even in the presence of multiple threads. Using CARR to apply the refactoring we get the following code, which produces the same results as the original program.

```
public class C7 implements TM {
        private final Object generated_lock_for_x = new Object();
        private int x;
        public void m(int caller) {
                synchronized(generated_lock_for_x) {
                        x = 0;
                        for(int i = 0; i < 100000; i++);
                        x++;
                        System.out.println(caller + ": I am exiting with x =" + x);
                }
        }
        @Override
        public void m1() {
                m(1);
        }
        @Override
        public void m2() {
                m(2);
        }
}
```

Refactored by CARR.

To restrict the access of different threads to the field we need to protect the access to it by a `synchronized` block. However, using any existing object as a lock could introduce deadlocks because it might be used in other parts of the code. For this reason, we use the field *generated_lock_for_x* that is initialized only once so all the the threads that use this instance synchronise on the same object. As a result, when a thread acquires the lock, it will initialise the field and use it in the same way as it was used when it was a local variable without any other thread interleaving with it. A side effect of this change is that the code will be slower depending on the size of the code block that is being protected by the new lock.

5.4 Claims

We claim that this tool successfully performs three concurrency aware refactorings, Inline Local, Convert Local Variable To Field and Move Method, without changing the behaviour of the original program apart from the cases mentioned in the limitations of each refactoring:

- Move Method: when the receiver of the method has the value **null**.
- Inline Local: when an exception that can be thrown is not resolved from the algorithm.
- Convert Local Variable To Field: when a livelock is introduced.

.

Chapter 6

Evaluation

6.1 Research Questions & Answers

In Chapter 1 we stated our research questions and so far we answered them considering the motivating examples. Generally, we answered the research questions as follows:

When does the current implementation of the targeted refactorings change the behaviour if the refactored program runs concurrently? From our research we conclude that the current Eclipse implementation of refactorings change the behaviour of code when the dependencies that are enforced from data and control flow along with the ones from JMM are not respected. The dependencies were modeled by SDFG and the cases when the refactorings broke were detected by the tool.

Can we fix the reasons that caused the new behaviour? We claim fixing the refactoring is possible and we created a tool to demonstrate that the cases of the motivating examples can be fixed.

Can we prove the correctness of the fixed version? We claim that the inference rules from Chapter 5 can be used to prove the correctness of the refactoring tool. In this chapter, we provide further evidence to support our claim and outline a proof using separation logic and the inference rules.

6.2 Evidence

Before outlining the proof of our algorithm we provide evidence that support the claims that we discussed in Chapters 4 and 5. The decision to provide evidence in the form of use cases instead of using whole programs was made to help us test the soundness of our algorithm instead of its efficiency. We argue that the efficiency can be improved by indexing the `Stmts` of the SDFG and optimizing the visiting algorithm of CARR but this is left as future work.

However, the soundness of the algorithms is important because combined with the proof outline can be used to argue about the correctness of the algorithm. Proving the correctness of the algorithms is crucial because, as we mentioned in Chapter 1, concurrency bugs are difficult to find. Although our use cases are designed to exhibit the concurrency bugs frequently, percentages of the bugs occurrences are provided to support our point.

6.2.1 Sieve of Eratosthenes

To evaluate our tool we are going to implement the Sieve of Eratosthenes (see Appendix B). To adapt the sieve to our needs, we implemented a sieve application that feeds sieves with different upper limits to two algorithms that accept sieves and process them in parallel.

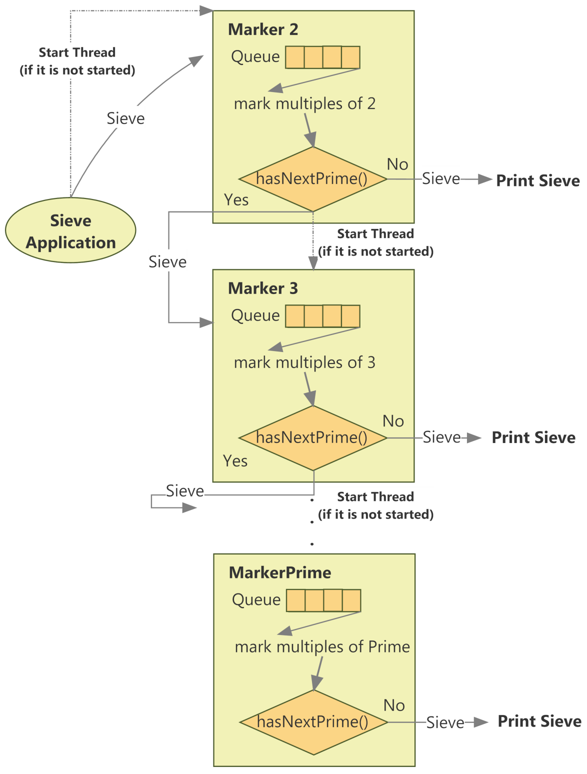


Figure 6.1: The application adds the sieves to the queue of the *Marker(2)*, after the *Marker* processes the sieve it either prints it, if there are no other prime numbers after this, or it adds it to the queue of the next *Marker*. Each *Marker* is instantiated once.

The first algorithm is shown in Figure 6.1, the class *Marker* implements *Runnable* and has a queue of sieves and a prime number. The *run()* method gets the first sieve out of the queue and marks the multiples of the prime number. Then, if there is another prime number after this one, it generates **once** the marker with the next prime and starts a new thread, it then passes the sieve to the next marker and continues to process the next sieve.

The class *Marker* also has a `volatile` field *finish*; when *finish* turns **true** the *Marker* will exit as soon as its queue is empty. The sieve application initiates this chain by initialising the first thread and the first marker with prime number the number 2. The sieve is modelled as an *ArrayList* of *Booleans* so it can be easily added in the queue of a *Marker*.

The second algorithm, shown in Figure 6.2, starts a new thread per sieve. Every thread requests a potential prime from the sieve and starts a new thread that runs the *mark()* method; this method marks the numbers that are multiples of the argument. In this algorithm it is possible that the *mark()* method is run more times than necessary since composite numbers could also be used for marking. But for our purposes we needed a specific method to be run in parallel with itself on the same object.

The sieve is modelled as a **boolean** array, this was chosen because we needed the different threads to access the same array simultaneously. The values in the array are changed only to **false**; as a result, even if two threads have a race on a position of the array if at least one of them is going to change the value to **false** it will be recorded. The code of the sieve is shown in the following listing. The method *mark()* is run in parallel with different arguments.

Move Method

The Move Method refactoring can be applied under different conditions. In this section we will examine moving both static and non static methods; these methods were chosen because the result from our implementation is different from the one from Eclipse.

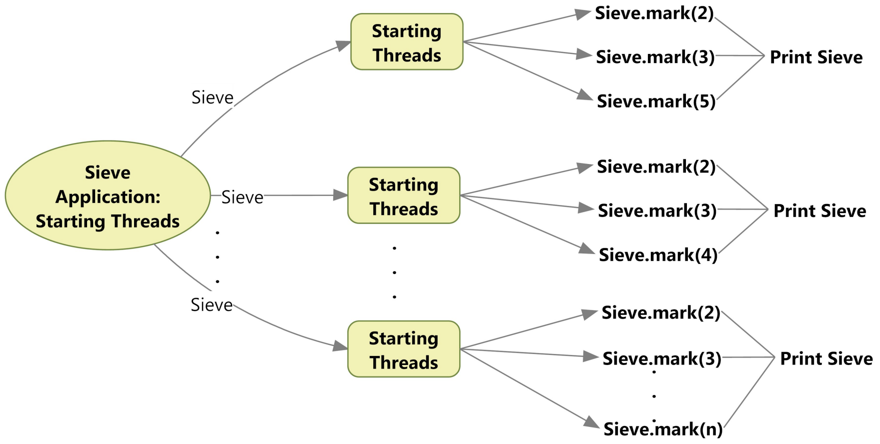


Figure 6.2: The application starts a thread per sieve. Each one of the threads requests the next potential prime number and starts another thread that marks the multiples of this number in the sieve.

Static. In the sieve application `synchronized` blocks are used to ensure that the method that prints the prime numbers and the method that prints the *Marker*'s status do not interleave resulting in inconsistent prints. Both methods are static and are declared in the *Marker*; as a result, both methods synchronise with *Marker.class*.

If we apply the Move Method refactoring as it is implemented by Eclipse, we can see that the refactoring performed by Eclipse changes the synchronisation dependencies since the inner code of the method has now a dependency on *SieveApplication.class* and not on *Marker.class*. As a result, printing the status of a marker and printing the prime numbers can interleave. Incosistent prints occurred in 5 out of 10 executions. The desugaring step of CARR ensures that the lock is not changed during the refactoring.

```
public class Marker implements Runnable {

    public static synchronized void printStatus(
         Marker m) {
        if(m.finish && m.sieves.isEmpty())
            System.out.println("Marker "+ m.prime
                 + " is finished!");
        else
            System.out.println("Marker "+ m.prime
                 + " is still processing, "+ m.sieves.
                 size() + " left...");
        if(m.nextMarker != null)
            printStatus(m.nextMarker);
    }
}

public class SieveApplication {

    public static void runSieveParallel() {
        Thread t = null;
        Marker m2 = new Marker(2);
        for(int i = 5; i < 101; i+=5) {
            Sieve s = new Sieve(i);
            if(t == null) {
                t = new Thread(m2);
                t.start();
            }
            m2.process(s);
        }
        m2.finish();
        for(int i = 0; i < 5; i++) {
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Marker.printStatus(m2);
        }
    }

    public static synchronized void print(Sieve sieve)
      {
        sieve.print();
    }

}
```

Move Method by Eclipse: **static** case.

```
public class Marker implements Runnable{

    public static synchronized void printStatus(
         Marker m) {
        if(m.finish && m.sieves.isEmpty())
            System.out.println("Marker "+ m.prime
                 + " is finished!");
        else
            System.out.println("Marker "+ m.prime
                 + " is still processing, "+ m.sieves.
                 size() + " left...");
        if(m.nextMarker != null)
            printStatus(m.nextMarker);
    }
}

public class SieveApplication {

    public static void runSieveParallel() {
        Thread t = null;
        Marker m2 = new Marker(2);
        for(int i = 5; i < 101; i+=5) {
            Sieve s = new Sieve(i);
            if(t == null){
                t = new Thread(m2);
                t.start();
            }
            m2.process(s);
        }
        m2.finish();
        for(int i = 0; i < 5; i++){
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Marker.printStatus(m2);
        }
    }

    public static void print(Sieve sieve) {
        synchronized(Marker.class) {
            sieve.print();
        }
    }
}
```

Move Method by CARR: **static** case.

Parameter Swap. The parameter swap case of the Move Method refactoring is going to be applied on the second algorithm shown in Figure 6.2. After processing a sieve the *SieveApplication* uses its `synchronized` method *print()* to print the prime numbers from each sieve.

The Eclipse refactoring tool rejects the Move Method refactoring because the method is `synchronized`. CARR applies the refactoring resulting in the code showing on the following listing. As we can see the `synchronized` keyword is desugared and the dependencies are correctly resolved.

```
public class SieveApplication {
    [...]
    private void processSieve(final BooleanSieve s) {
        int prime = 2;
        List<Thread> pool = new ArrayList<
            Thread>();
        while(prime != 0){
            final int p = prime;
            Thread t = new Thread(new Runnable()
                {
                @Override
                public void run() {
                    s.mark(p);
                }
            });
            pool.add(t);
            t.start();
            prime = s.getNextPrime(prime);
        }
        for(Thread t : pool){
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        print(s);
    }

    public synchronized void print(BooleanSieve
        sieve) {
        System.out.println("\nPrime numbers until "
            + (sieve.size() − 1));
        for(int i = 2; i < sieve.size(); i ++) {
            if(sieve.isPrime(i)) System.out.print(i + "
                , ");
        }
    }
}

public class BooleanSieve{
    [...]
}
```

Original

```
public class SieveApplication{
    [...]
    private void processSieve(final BooleanSieve s) {
        int prime = 2;
        List<Thread> pool = new ArrayList<
            Thread>();
        while(prime != 0){
            final int p = prime;
            Thread t = new Thread(new Runnable()
                {
                @Override
                public void run() {
                    s.mark(p);
                }
            });
            pool.add(t);
            t.start();
            prime = s.getNextPrime(prime);
        }
        for(Thread t : pool){
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        s.print(this);
    }
}

public class BooleanSieve{
    [...]
    public void print(SieveApplication sieve) {
        synchronized(sieve) {
            System.out.println("\nPrime numbers
                until " + (this.size() − 1));
            for(int i = 2; i < this.size(); i ++) {
                if(this.isPrime(i))
                    System.out.print(i + ", ");
            }
        }
    }
}
```

Refactored

Move Method: **Parameter Swap** case.

Inline Local

The Inline Local refactoring illustrated in the following listing is rejected by CARR because the `change`(·) that captures a potential change of the receiver of the methods in lines 13 and 15 cannot be generated by the inference rules, since the local variable is replaced by a constructor call.

In this case, the refactoring is correctly rejected because the refactored code, transformed by Eclipse, creates multiple markers instead of one. Each one of the first markers process a sieve, but they never exit because the field *finish* is never turned into **true**, and the last marker starts and finishes instantly because its queue is empty and the field *finish* turned to *true*. The bug is present in every execution.

```
public class SieveApplication {

    [...]
    public static void runSieveParallel() {
        Thread t = null;
        Marker m2 = new Marker(2);
        for(int i = 5; i < 101; i+=5) {
            Sieve s = new Sieve(i);
            if(t == null) {
                t = new Thread(m2);
                t.start();
            }
            m2.process(s);
        }
        m2.finish();
        for(int i = 0; i < 5; i++){
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Marker.printStatus(m2);
        }
    }
}
```

Original code.

```
public class SieveApplication {

    [...]
    public static void runSieveParallel() {
        Thread t = null;

        for(int i = 5; i < 101; i+=5) {
            Sieve s = new Sieve(i);
            if(t == null) {
                t = new Thread(new Marker(2));
                t.start();
            }
            new Marker(2).process(s);
        }
        new Marker(2).finish();
        for(int i = 0; i < 5; i++){
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Marker.printStatus(new Marker(2));
        }
    }
}
```

INLINE LOCAL by Eclipse.

The refactoring changed the behaviour of the code since in every location where the local is inlined it creates new instances of the objects and applies the operation on the recently created object.

Convert Local Variable To Field

The following listing contains the method *mark()* that is responsible for marking the multiples of the number that is given as argument.

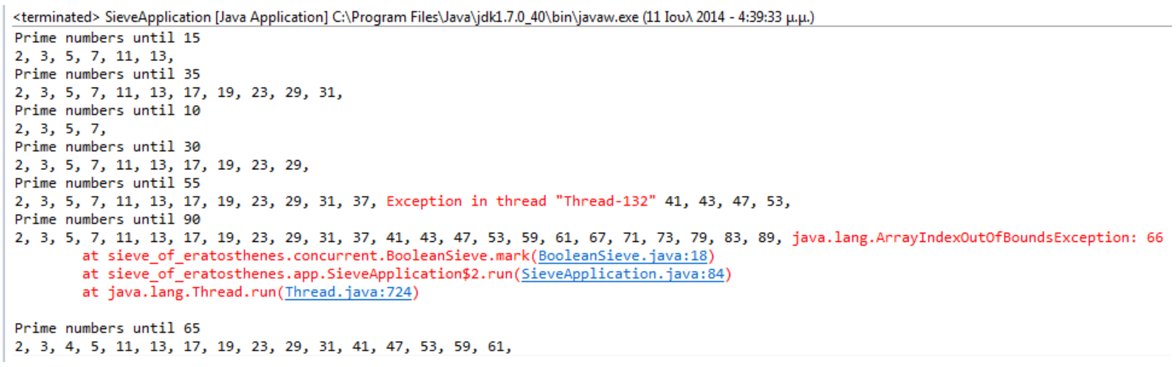
```
public class BooleanSieve {
    boolean[] sieve;

    [....]

    public void mark(int prime) {
        int i;
        for(i = 2*prime; i < sieve.length; i+=prime) {
            sieve[i] = false;
        }
    }
    [....]
}
```

The original code of the method *mark()*.

If we apply the CONVERT LOCAL VARIABLE TO FIELD refactoring on the local variable *i* of this method using both the Eclipse and CARR, the followed code is produced:



```
<terminated> SieveApplication [Java Application] C:\Program Files\Java\jdk1.7.0_40\bin\javaw.exe (11 Ιουλ 2014 - 4:39:33 μ.μ.)
Prime numbers until 15
2, 3, 5, 7, 11, 13,
Prime numbers until 35
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
Prime numbers until 10
2, 3, 5, 7,
Prime numbers until 30
2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
Prime numbers until 55
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, Exception in thread "Thread-132" 41, 43, 47, 53,
Prime numbers until 90
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, java.lang.ArrayIndexOutOfBoundsException: 66
        at sieve_of_eratosthenes.concurrent.BooleanSieve.mark(BooleanSieve.java:18)
        at sieve_of_eratosthenes.app.SieveApplication$2.run(SieveApplication.java:84)
        at java.lang.Thread.run(Thread.java:724)

Prime numbers until 65
2, 3, 4, 5, 11, 13, 17, 19, 23, 29, 31, 41, 47, 53, 59, 61,
```

Figure 6.3: Output of the program refactored by Eclipse

```
public class BooleanSieve {


    boolean[] sieve;
    private int i;

    public void mark(int prime) {

        for(i = 2*prime; i < sieve.length; i += prime
            ) {
                sieve[i] = false;
        }

    }
}
```

Convert Local Variable To Field by Eclipse.

```
public class BooleanSieve {
    private final Object generated_lock_for_i = new
            Object();
    private int i;
    boolean [] sieve;

    public void mark(int prime) {
        synchronized(generated_lock_for_i) {
            for(i=2*prime; i < sieve.length; i+=
                acquireLock
                sieve[i] = false
            }
        }
    }
}
```

Convert Local Variable To Field by CARR.

As expected the refactored code from Eclipse introduced races and consequently prints inconsistent results as shown in Figure 6.3. The output included *ArrayIndexOutOfBoundsException* or composites numbers found as primes and vice versa since i can now "jump" indexes. The concurrency bugs occurred in 3 out of 10 executions. On the other hand, the refactored program from CARR did not have any races due to the `synchronized` block.

6.2.2 Other Use Cases

The following cases were chosen because the Eclipse refactoring tool rejects both of them due to the multiple assignments. SDFG allows CARR to perform the refactoring even in cases where assignments to the local variable are inside control flow statements as long as at any read of the local variable there is only one possible expression to be inlined.

Break

The following code demonstrates how CARR inlines the local variable x even if it contains assignments inside a **while** loop. The Inline Local can be performed because before reading the value of x at any point in the code there is only one write visible, this is detected and verified by the SDFG constraint and the refactoring is not rejected.

```java
public class InlineWithBreak {
    public void m(int m) {
        int x = 0;
        int k = x ++;
        while(k < m) {
            x = 6;
            if(x < m)
                break;
        }
        x = 3;
        System.out.println(x);
    }
}
```

Original

```java
public class InlineWithBreak {
    public void m(int m) {

        int k = 0 + 1;
        while(k < m) {

            if(6 < m)
                break;
        }

        System.out.println(3);
    }
}
```

Inline Local by CARR.

Inline Local performed by CARR: every read of variable x depends on only one assignment, for this reason the refactoring can be performed successfully.

Exceptions

A similar example is demonstrated in the following listing using exceptions. In this example every **catch** has a unique path that leads to the associated block, so the reads of local variable x are uniquely defined. On the other hand, the **finally** block can be reached by three different paths; however, the assignment of x overwrites all the previous ones and the program always reads and prints the same value.

```java
public class InlineWithExceptions {
    public void m(int l) {
        int x = 0;
        try {
            if(l > 0){
                throw new NullPointerException();
            }
            x = 1;
            throw new IOException();
        }
        catch(NullPointerException e){
            System.out.println(x);
        }
        catch(IOException e){
            System.out.println(x);
        }
        finally{
            x = 2;
            System.out.println(2);
        }
    }
}
```

Original

```java
public class InlineWithExceptions {
    void m(int l) {

        try {
            if(l > 0) {
                throw new NullPointerException();
            }

            throw new IOException();
        }
        catch(NullPointerException e) {
            System.out.println(0);
        }
        catch(IOException e) {
            System.out.println(1);
        }
        finally {

            System.out.println(2);
        }
    }
}
```

Inline Local by CARR.

Inline Local performed by CARR: the **catch** blocks are reachable from only one path so there is only one expression to be inlined. In the case of **finally**, although there multiple paths, which result in more than one expressions to be inlined, the assignment overwrites all of them and the only expression to be inlined is the value *2*.

6.3 Proof

In this section we are going to outline the proof of correctness of our solution; the formalised proof is left as future work. We suggest that the correctness of the inference rules entail the correctness of CARR, consequently, we are going to use separation logic for non-blocking structures [23] to prove that the premise and conclusion of the inference rules from Chapter 5 preserve the dependencies with respect to the refactoring.

Parkinson et al. [23] suggest to use pre-conditions and post-conditions to examine the local state (store) and an invariant to examine the shared state (heap). The invariant is only used to prove the correctness of commands, in our case `Stmts`, that affect shared state.

From separation logic we have the following equation adapted to SDFG:

$$\frac{\Gamma; \mathbf{emp} \vdash \{I * P\}\ \texttt{Stmt}\ \{I * Q\}}{\Gamma; I \vdash \{P\}\ \text{atomic}(\texttt{Stmt})\ \{Q\}} \tag{6.1}$$

Given an environment Γ and a shared data invariant I, we apply the pre-condition P and post-condition Q as in Hoare logic. However, if the `Stmt` regards a shared memory location, then and only then the invariant I is added to P and Q using separating conjunction. The environment Γ can be thought of as the declarations of the SDFG and the state S consists of the store s and the heap h.

We define how the refactoring changed the environment Γ and the state S and then we examine several inference rules based on these changes.

Move Method

Considering the effect of the Move Method refactoring on the code we expect that the receiver of the method will be changed. However, the expected changes depend on the specific case of the Move Method refactoring.

The common case in all Move Method cases is that the refactoring changes the environment Γ since the declaration of the method has changed.

Lemma 6.3.1. *The environment Γ' of the refactored by* Move Method *program is the same with the original environment Γ apart from the declaration of the targeted method.*

$$\Gamma' = \Gamma - (methodDecl\ :\ address) + (newMethodDecl\ :\ address) \tag{6.2}$$

Lemma 6.3.2. *The store s' of the refactored by* Move Method *program in the case of swap parameter regarding the moved method's code differs from the original s as follows:*

$$s' = s + (newPar\ :\ s[\boldsymbol{this}],\ \boldsymbol{this}\ :\ s[par]) -\ par \tag{6.3}$$

Theorem 6.3.3. *When the* Move Method *refactoring using parameter swap is applied, the inference rule 5.1 does not introduce or loses dependencies concerning the entry or the exit of the targeted method.*

Proof. The notation $\gg$ represents a dependency and this relation is transitive.

$$\begin{gathered}\Gamma, s, h; I \vdash \{true\}\ \text{entryPoint}(\text{id},\ m),\ \text{exitPoint}(\text{id},\ m)\ \{true\} \\ \Longrightarrow \\ \Gamma', s', h; I \vdash \{true\}\ \text{entryPoint}(\text{id},\ m'),\ \text{exitPoint}(\text{id},\ m')\ \{true\}\end{gathered} \tag{6.4}$$

since due to lemma 6.3.1 the following are true:

$$\Gamma[m] = \Gamma'[m']$$

□

Theorem 6.3.4. *When the* Move Method *refactoring using parameter swap is applied, the inference rule 5.7 and 5.8 do not introduce or loses synchronisation dependencies.*

Proof. Because the Move Method swaps one parameter with the receiver of the method, the store of the method is changed as it is shown in equation 6.3.

$$\begin{gathered}\Gamma, s, h; I \vdash \{true\}\ \text{acquireLock(id, } \textbf{this}\text{, c1)}\ \{s[\textbf{this}] \gg s[\text{c1}]\} \\ \Longrightarrow \\ \Gamma', s', h; I \vdash \{true\}\ \text{acquireLock(id, newPar, c1)}\ \{s'[\text{newPar}] \gg s'[\text{c1}]\}\end{gathered} \tag{6.5}$$

$$\begin{gathered}\Gamma, s, h; I \vdash \{true\}\ \text{acquireLock(id, par, c1)}\ \{s[\text{par}] \gg s[\text{c1}]\} \\ \Longrightarrow \\ \Gamma', s', h; I \vdash \{true\}\ \text{acquireLock(id, } \textbf{this}\text{, c1)}\ \{s'[\textbf{this}] \gg s'[\text{c1}]\}\end{gathered} \tag{6.6}$$

because of lemma 6.3.2 the $s[\textbf{this}]$ and the $s'[newPar]$ evaluate to the same address, the same property holds for $s[\text{par}]$ and $s'[\textbf{this}]$. Consequently, the synchronisation dependencies are preserved because the lock has not changed. □

Theorem 6.3.5. *When the* Move Method *refactoring using parameter swap is applied, the inference rule 5.9 does not introduce or loses dependencies concerning the method call.*

Proof. The following proof refers to the inner code of the moved method. Based on the expected changes, we have:

$$\begin{gathered}\Gamma, s, h; I \vdash \{s[\textbf{this}] \neq null\}\ \text{call(id, } \textbf{this}\text{, m, par)}\ \{Q, m \gg s[\textbf{this}], m \gg s[\text{par}]\} \\ \Longrightarrow \\ \Gamma', s', h; I \vdash \{s'[\textbf{this}] \neq null\}\ \text{call(id, } \textbf{this}\text{, m', newPar)}\ \{Q, m' \gg s'[\text{newPar}], m' \gg s'[\textbf{this}]\}\end{gathered} \tag{6.7}$$

since due to 6.2 and 6.3 the following are true:

$$\Gamma[m] = \Gamma'[m'],\ s[\textbf{this}] = s'[newPar],\ s[\text{par}] = s'[\textbf{this}]$$

□

Example. In the following example we demonstrate the application of the previous theorems. Each inference rule ensures that dependencies are not lost or added. The relation between the environment and the state of the program is expressed by the following equations:

- Original: Γ = [A.m(B), A.n()], s = (**this** : addr1, par : addr2)
- Refactored: Γ' = [B.m(A), A.n()], s' = (newPar : addr1, **this** : addr2)

```
class A {

    public synchronized void m(B b) {


        m(b);


    }
}
class B {}
```

Original code in Java.

```
Program(
    { method(A.m(B))}, {
    entryPoint(e1, m),releaseLock(l1, this, e1),
    acquireLock(l1, this, c1),acquireLock(l1, this, r1),
        acquireLock(l1, this, t1),
    call(c1, t1, m(B), r1), read(r1, b, p1), read(t1, this, _),
    releaseLock(l1, this, c1),releaseLock(l1, this, r1),
        releaseLock(l1, this, t1),
    exitPoint(ex1, m),acquireLock(l1, this, ex1),
    }
)
```

Original code in SDFG.

Example original code.

Applying Hoare logic on the original method of the SDFG program we get the following steps, we assume that $l[.]$ represents the use of the object referred from this memory location as a lock:

$$\begin{aligned}
&\Gamma, s, h; I \vdash\\
\{true\}\ &\mathrm{entryPoint(e1,\ m)}\ \{true\},\\
\{true\}\ &\mathrm{releaseLock(l1,\ \mathbf{this},\ e1)}\ \{\Gamma[m] \ll l[\mathbf{this}]\},\\
\{\Gamma[m] \ll l[\mathbf{this}]\}\ &\mathrm{acquireLock(l1,\ \mathbf{this},\ t1)}\ \{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}]\},\\
\{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}]\}\ &\mathrm{acquireLock(l1,\ \mathbf{this},\ r1)}\ \{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}]\},\\
\{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}]\}\ &\\
&\mathrm{acquireLock(l1,\ \mathbf{this},\ c1)}\\
&\{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}]\},\\
\{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}]\}\ &\\
&\mathrm{read(t1,\ \mathbf{this},\ _)}\\
&\{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}]\},\\
\{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}]\}\ &\\
&\mathrm{read(r1,\ b,\ p1)}\\
\{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}], &\Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1]\},\\
\{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1]\}\ &\\
&\mathrm{call(c1,\ \mathbf{this},\ m,\ b)}\\
\{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1], &\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b]\},\\
\{\Gamma[m] \ll l[\mathbf{this}], s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1], &\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b]\}\\
&\mathrm{releaseLock(l1,\ \mathbf{this},\ t1)}\\
\{\Gamma[m] \ll l[\mathbf{this}], l[\mathbf{this}] \gg s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1], &\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b]\},\\
\{\Gamma[m] \ll l[\mathbf{this}], l[\mathbf{this}] \gg s[\mathbf{this}] \gg l[\mathbf{this}], s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1], &\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b]\}\\
&\mathrm{releaseLock(l1,\ \mathbf{this},\ r1)}\\
\{\Gamma[m] \ll l[\mathbf{this}], l[\mathbf{this}] \gg s[\mathbf{this}] \gg l[\mathbf{this}], l[\mathbf{this}] \gg s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1],\ &\\
&\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b]\},\\
\{\Gamma[m] \ll l[\mathbf{this}], l[\mathbf{this}] \gg s[\mathbf{this}] \gg l[\mathbf{this}], l[\mathbf{this}] \gg s[b] \gg l[\mathbf{this}], \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1],\ &\\
\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b]\}\ &\mathrm{releaseLock(l1,\ \mathbf{this},\ c1)}\\
\{\Gamma[m] \ll l[\mathbf{this}], l[\mathbf{this}] \gg s[\mathbf{this}] \gg l[\mathbf{this}], l[\mathbf{this}] \gg s[b] \gg l[\mathbf{this}], l[\mathbf{this}] \gg \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1],\ &\\
&\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b]\},\\
\{\Gamma[m] \ll l[\mathbf{this}], l[\mathbf{this}] \gg s[\mathbf{this}] \gg l[\mathbf{this}], l[\mathbf{this}] \gg s[b] \gg l[\mathbf{this}], l[\mathbf{this}] \gg \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1],\ &\\
\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b]\}\ &\mathrm{acquireLock(l1,\ \mathbf{this},\ ex1)}\\
\{\Gamma[m] \ll l[\mathbf{this}], l[\mathbf{this}] \gg s[\mathbf{this}] \gg l[\mathbf{this}], l[\mathbf{this}] \gg s[b] \gg l[\mathbf{this}], l[\mathbf{this}] \gg \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1],\ &\\
&\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b], \Gamma[m] \gg l[\mathbf{this}]\},\\
\{\Gamma[m] \ll l[\mathbf{this}], l[\mathbf{this}] \gg s[\mathbf{this}] \gg l[\mathbf{this}], l[\mathbf{this}] \gg s[b] \gg l[\mathbf{this}], l[\mathbf{this}] \gg \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1],\ &\\
\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b], \Gamma[m] \gg l[\mathbf{this}]\}\ &\mathrm{exitPoint(ex1,\ m)}\\
\{\Gamma[m] \ll l[\mathbf{this}], l[\mathbf{this}] \gg s[\mathbf{this}] \gg l[\mathbf{this}], l[\mathbf{this}] \gg s[b] \gg l[\mathbf{this}], l[\mathbf{this}] \gg \Gamma[c1] \gg l[\mathbf{this}], s[b] \gg [p1],\ &\\
&\Gamma[c1] \gg s[\mathbf{this}], \Gamma[c1] \gg s[b], \Gamma[m] \gg l[\mathbf{this}]\}
\end{aligned}$$

```
class A {
}
class B {
    public void m(A b) {
        synchronized(b){

            this.m(b);

        }
    }
}
```

Refactored code in Java.

```
Program(
    { method(B.m(A)) }, {

    entryPoint(e1, m'),releaseLock(l1, b, e1),
    acquireLock(l1, b, c1), acquireLock(l1, b, t1),
        acquireLock(l1, b, r1),
    call(c1, t1, m(A), r1), read(t1, this, _), read(r1, b, _),
    releaseLock(l1, b, c1), releaseLock(l1, b, t1),
        releaseLock(l1, b, r1),
    exitPoint(ex1, m'),acquireLock(l1, b, ex1),
})
```

Refactored code in SDFG.

Example refactored code.

Applying Hoare logic on the refactored code of the SDFG program we can see that the dependencies are completely preserved and we confirm the claim that the Move Method refactoring is correct.

$$
\begin{gathered}
\Gamma', s', h; I \vdash \\
\{true\}\ \text{entryPoint(e1, m')}\ \{true\},\ \text{[theorem 6.3.3]} \\
\{true\}\ \text{releaseLock(l1, b, e1)}\ \{\Gamma'[m'] \ll l'[b]\},\ \text{[theorem 6.3.4]} \\
\{\Gamma'[m'] \ll l'[b]\}\ \text{acquireLock(l1, b, t1)}\ \{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b]\}, \\
\{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b]\}\ \text{acquireLock(l1, b, r1)}\ \{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b], s'[\mathbf{this}] \gg l'[b]\}, \\
\{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b], s'[\mathbf{this}] \gg l'[b]\} \\
\text{acquireLock(l1, b, c1)} \\
\{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b], s[\mathbf{this}] \gg l'[b], \Gamma'[c1] \gg l'[b]\}, \\
\{\Gamma[m'] \ll l'[b], s'[b] \gg l'[b], s'[b] \gg l'[b], \Gamma'[c1] \gg l'[b]\} \\
\text{read(r1, } \mathbf{this}\text{, \_)} \\
\{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b], s'[\mathbf{this}] \gg l'[b], \Gamma'[c1] \gg l'[b]\}, \\
\{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b], s'[b] \gg l'[b], \Gamma'[c1] \gg l'[b]\} \\
\text{read(t1, b, p1)} \\
\{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b], s'[\mathbf{this}] \gg l'[b], \Gamma'[c1] \gg l'[b], s'[b] \gg [p1]\}, \\
\{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b], s[\mathbf{this}] \gg l'[b], \Gamma'[c1] \gg l'[b], s[b] \gg [p1]\} \\
\text{call(c1, r1, m', t1)} \\
\{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b], s'[b] \gg l'[b], \Gamma'[c1] \gg l'[b], s'[b] \gg [p1], \Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[\mathbf{this}]\}, \\
\{\Gamma'[m'] \ll l'[b], s'[b] \gg l'[b], s'[\mathbf{this}] \gg l'[b], \Gamma'[c1] \gg l'[b], s'[b] \gg [p1], \Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[\mathbf{this}]\} \\
\text{releaseLock(l1, b, t1)} \\
\{\Gamma'[m'] \ll l'[b], l'[b] \gg s'[b] \gg l'[b], s'[\mathbf{this}] \gg l'[b], \Gamma'[c1] \gg l'[b], s'[b] \gg [p1], \Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[\mathbf{this}]\}, \\
\{\Gamma'[m'] \ll l'[b], l'[b] \gg s'[b] \gg l'[b], s'[\mathbf{this}] \gg l'[b], \Gamma'[c1] \gg l'[b], s'[b] \gg [p1], \Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[\mathbf{this}]\} \\
\text{releaseLock(l1, b, r1)} \\
\{\Gamma'[m'] \ll l'[b], l'[b] \gg s'[b] \gg l'[b], l'[b] \gg s'[\mathbf{this}] \gg l'[b], \Gamma'[c1] \gg l'[b], s'[b] \gg [p1], \\
\Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[\mathbf{this}]\}, \\
\{\Gamma'[m'] \ll l'[b], l'[b] \gg s'[b] \gg l'[b], l'[b] \gg s'[\mathbf{this}] \gg l'[b], \Gamma'[c1] \gg l'[b], s'[b] \gg [p1], \\
\Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[\mathbf{this}]\}\ \text{releaseLock(l1, b, c1)} \\
\{\Gamma'[m'] \ll l'[b], l'[b] \gg s'[b] \gg l'[b], l'[b] \gg s'[\mathbf{this}] \gg l'[b], l'[b] \gg \Gamma'[c1] \gg l'[b], s'[b] \gg [p1], \\
\Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[\mathbf{this}]\}, \\
\{\Gamma'[m'] \ll l'[b], l'[b] \gg s'[b] \gg l'[b], l'[b] \gg s'[\mathbf{this}] \gg l'[b], l'[b] \gg \Gamma'[c1] \gg l'[b], s'[\mathbf{this}] \gg [p1], \\
\Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[\mathbf{this}]\}\ \text{acquireLock(l1, b, ex1)} \\
\{\Gamma'[m'] \ll l'[b], l'[b] \gg s'[b] \gg l'[b], l'[b] \gg s'[\mathbf{this}] \gg l'[b], l'[b] \gg \Gamma'[c1] \gg l'[b], s'[b] \gg [p1], \\
\Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[\mathbf{this}], \Gamma'[m'] \gg l'[b]\}, \\
\{\Gamma'[m'] \ll l'[b], l'[b] \gg s'[b] \gg l'[b], l'[b] \gg s'[\mathbf{this}] \gg l'[b], l'[b] \gg \Gamma'[c1] \gg l'[b], s'[b] \gg [p1], \\
\Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[b], \Gamma'[m'] \gg l'[b]\}\ \text{exitPoint(ex1, m')} \\
\{\Gamma[m'] \ll l'[b], l'[b] \gg s'[b] \gg l'[b], l'[b] \gg s'[\mathbf{this}] \gg l'[b], l'[b] \gg \Gamma'[c1] \gg l'[b], s'[b] \gg [p1], \\
\Gamma'[c1] \gg s'[b], \Gamma'[c1] \gg s'[b], \Gamma'[m'] \gg l'[b]\}
\end{gathered}
$$

Inline Local

Considering the effect of the Inline Local refactoring on the code, the refactored environment is expected to be the same with the original. The refactored store should differ from the original only for missing the entry of the local variable. As far as the heap is concerned, the Inline Local does not remove any mappings; however, it might add multiple copies of an object if an inlined expression contains a constructor.

Lemma 6.3.6. *The refactored heap h' is a super set of the original h and the refactored store s' of* *the refactored by* Inline Local *program differs from the original s as follows:*

$$s' = s - local,\ h' \supseteq h \tag{6.8}$$

Theorem 6.3.7. *When the* Inline Local *refactoring is applied, the inference rule 5.19 preserves the indirect dependencies of the original program:*

$$\begin{aligned} \Gamma, s, h; I \vdash \{true\}&assign(uId,\ u,\ xRId) \\ &read(xRId,\ x,\ xAId), \\ &assign(xAId,\ z,\ depId)\{S[uId] \lll S[depId]\}, \\ &\Longrightarrow \\ \Gamma, s', h'; I \vdash \{true\}&assign(f(uId), u, f(depId))\{S'[f(uId)] \gg S'[f(depId)]\} \end{aligned} \tag{6.9}$$

Proof. To prove theorem 6.3.7 we will use two axioms from Hoare logic, the consequence rule and the composition rule:

$$\frac{P_1 \rightarrow P_2\{P_2\}C\{R_1\}, R_1 \rightarrow R_2}{\{P_1\}C\{R_2\}}, \frac{\{P\}C_1\{Q_1\}, \{Q_2\}C_2\{R\}}{\{P\}C_1, C_2\{R\}}$$

The consequence rule consists of the strengthening of the pre-condition and of the weakening of the post-condition. Applying these rules on the conclusion of the inference rule 5.18 we end up to the premise.

$$\begin{aligned} \Gamma, s, h; I \vdash \{true\}&\ \text{assign(uId, u, xRId)}\{S[\text{uId}] \gg S[\text{xRId}]\}, \\ \{true\}&\ \text{read(xRId, x, xAId)}\{S[\text{xRId}] \gg S[\text{xAId}]\}, \\ \{true\}&\ \text{assign(xAId, z, depId)}\{S[xRId] \gg S[depId]\}, \\ \text{Strengthening of Pre-conditions}\ &\Longrightarrow \\ \Gamma, s, h; I \vdash \{true\}&\ \text{assign(uId, u, xRId)}\{S[\text{uId}] \gg S[\text{xRId}]\}, \\ \{S[\text{uId}] \gg S[\text{xRId}]\}&\ \text{read(xRId, x, xAId)}\{S[\text{uId}] \gg S[\text{xRId}], S[\text{xRId}] \gg S[\text{xAId}]\}, \\ \{S[\text{uId}] \gg S[\text{xRId}], S[\text{xRId}] \gg S[\text{xAId}]\}&\ \text{assign(xAId, z, depId)} \\ &\{S[\text{uId}] \gg S[\text{xRId}], S[\text{xRId}] \gg S[\text{xAId}], S[xRId] \gg S[depId]\} \\ \text{Rule of Composition}\ &\Longrightarrow \\ \Gamma, s, h; I \vdash \{true\}&\ \text{assign(uId, u, xRId), read(xRId, x, xAId), assign(xAId, z, depId)} \\ &\{S[\text{uId}] \gg S[\text{xRId}], S[\text{xRId}] \gg S[\text{xAId}], S[xRId] \gg S[depId]\} \\ \text{Weakening of Post-condition}\ &\Longrightarrow \\ \Gamma, s, h; I \vdash&\ \text{assign(uId, u, xRId), read(xRId, x, xAId), assign(xAId, z, depId)}\{S[xRId] \gg S[depId]\} \\ \text{Based on 6.8}\ &\Longrightarrow \\ \Gamma, s', h'; I \vdash \{true\}&\ \text{assign(f(uId), u, f(depId))}\{S'[f(\text{uId})] \gg S'[f(\text{depId})]\} \end{aligned}$$

We will examine the previous steps from the perspective of two examples. One example will demonstrate why a refactoring was correctly rejected and the other why another example was accepted.

Example #1. The first example we will examine is the *C6* example from Chapter 3. Based on the example we define the environment and the state of the original and the refactored program as follows:

- Original: Γ = [C6.m1()], s = (y : _, x : _)
- Refactored: Γ' = [C6.m1()], s' = (y : _)

```
class C6 {

    public void m1() {
        boolean y = true;
        boolean x = (y = true);
        y = false;
        System.out.println(x);
        System.out.println(y);
    }
}
```

```
Program(
    { method(C6.m1()) }, {
    entryPoint(e1, m),
    assign(a1, y, true),
    assign(a2x, x, r2y), read(r2y, y, a2y), assign(a2y, y, true),
    assign(a3, y, false),
    call(c4, out, println(), r4), read(r4, x, a2x),
    call(c5, out, println(), r5), read(r5, y, a3),
    exitPoint(e2,m)
})
```

Original code in Java. Original code in SDFG.

Example #1, original code.

Applying Hoare logic on the original SDFG program we get the following steps and we extract the pre-condition and post-condition of every `Stmt`.

$$
\begin{aligned}
&\Gamma, s, h; I \vdash \\
\{true\}\ &\text{assign(a1,y, true)}\{s[\text{y}] \gg true\}, \\
\{s[\text{y}] \gg true\}\ &\text{assign(a2y, y, true)}\{s[\text{y}] \gg true\}, \\
\{s[\text{y}] \gg true\}\ &\text{read(r2y, y, a2y)}\{s[\text{y}] \gg true\}, \\
\{s[\text{y}] \gg true\}\ &\text{assign(a2x, x, r2y)}\{s[\text{y}] \gg true, s[\text{x}] \gg s[y]\}, \\
\{s[\text{y}] \gg true, s[\text{x}] \gg true\}\ &\text{assign(a3, y, false)}\{s[\text{y}] \gg false, s[\text{x}] \gg true\}, \\
\{s[\text{y}] \gg false, s[\text{x}] \gg true\}\ &\text{read(r4, x, a2x)}\{s[\text{y}] \gg false, s[\text{x}] \gg true\}, \\
\{S[out] \neq null, s[\text{y}] \gg false, s[\text{x}] \gg true\}\ &\text{call(c4, out, println(), r4)}\{s[\text{y}] \gg false, s[\text{x}] \gg true\}, \\
\{s[\text{y}] \gg false, s[\text{x}] \gg true\}\ &\text{read(r5, y, a3)}\{s[\text{y}] \gg false, s[\text{x}] \gg true\}, \\
\{S[out] \neq null, s[\text{y}] \gg false, s[\text{x}] \gg true\}\ &\text{call(c5, out, println(), r5)}\{s[\text{y}] \gg false, s[\text{x}] \gg true\}
\end{aligned}
$$

Then we examine the outcome from the Eclipse refactoring tool which is shown in the following listing:

```
class C6 {

    public void m1() {
        boolean y = true;

        y = false;
        System.out.println((y=true));
        System.out.println(y);
    }
}
```

```
Program(
    { method(C6.m1()) }, {
    entryPoint(e1, m),
    assign(a1, y, true),

    assign(a3, y, false),
    call(c4, out, println(), r2y'),read(r2y', y, a2y'),assign(a2y',y,true),
    call(c5, out, println(), r5), read(r5, y, a2y'),
    exitPoint(e2, m)
})
```

Refactored code in Java. Refactored code in SDFG.

Example #1, refactored code.

Applying Hoare logic on the refactored SDFG program we get the following steps and we extract the pre-condition and post-condition for every `Stmt`.

$$
\begin{array}{rl}
 & \Gamma, s', h'; I \vdash \\
\{true\} & \text{assign(a1,y, true)}\{s'[\text{y}] \gg true\}, \\
\{s'[\text{y}] \gg true\} & \text{assign(a3, y, false)}\{s'[\text{y}] \gg false\}, \\
\{s'[\text{y}] \gg false\} & \text{assign(a2y', y, true)}\{s'[\text{y}] \gg false\}, \\
\{s'[\text{y}] \gg true\} & \text{read(r2y', y, a2y')}\{s'[\text{y}] \gg true\}, \\
\{S'[out] \neq null, s'[\text{y}] \gg true, \} & \text{call(c4, out, println(), r2y')}\{s'[\text{y}] \gg true\}, \\
\{s'[\text{y}] \gg true\} & \text{read(r5, y, a2y')}\{s'[\text{y}] \gg true\}, \\
\{S'[out] \neq null, s'[\text{y}] \gg true, \} & \text{call(c5, out, println(), r5)}\{s'[\text{y}] \gg true\}
\end{array}
$$

Comparing the two post-conditions with respect to the Inline Local refactoring, we accept the missing predicate that refers to x since this is the inlined variable. However, we can see that the data dependency from the $r5$ `read(·)` has changed from $a3$ to $a2y'$, and these identifiers are not connected by the mapping function *f*, for this reason CARR rejects this refactoring. This rejection is confirmed by comparing the two post-conditions of the `read(·)` with identifier $r5$ since the refactored one ensures that the value of y is **true** whereas the original value was **false**.

Example #2. As a second example we will use a slightly changed version of the class $C6$. We will reorder the two prints which will make the refactored code accepted by CARR.

```
class C6Accepted {

    public void m1() {
        boolean y = true;
        boolean x = (y = true);
        y = false;
        System.out.println(y);
        System.out.println(x);
    }
}
```

Original code in Java.

```
Program(
    { method(C6Accepted.m1()) }, {
    entryPoint(e1, m),
    assign(a1, y, true),
    assign(a2x, x, r2y), read(r2y, y, a2y), assign(a2y, y, true),
    assign(a3, y, false),
    call(c5, out, println(), r5), read(r5, y, a3),
    call(c4, out, println(), r4), read(r4, x, a2x),
    exitPoint(e2, m)
})
```

Original code in SDFG.

Example #2, original code.

The SDFG version of the original program preserved the identifiers as they were assigned in the previous example so it will be easier to compare the two versions of the program. Applying Hoare logic on the original SDFG program we get the following steps.

$$
\begin{array}{rl}
 & \Gamma, s, h; I \vdash \\
\{true\} & \text{assign(a1,y, true)}\{s[\text{y}] \gg true\}, \\
\{s[\text{y}] \gg true\} & \text{assign(a2y, y, true)}\{s[\text{y}] \gg true\}, \\
\{s[\text{y}] \gg true\} & \text{read(r2y, y, a2y)}\{s[\text{y}] \gg true\}, \\
\{s[\text{y}] \gg true\} & \text{assign(a2x, x, r2y)}\{s[\text{y}] \gg true, s[\text{x}] \gg s[y]\}, \\
\{s[\text{y}] \gg true, s[\text{x}] \gg true\} & \text{assign(a3, y, false)}\{s[\text{y}] \gg false, s[\text{x}] \gg true\}, \\
\{s[\text{y}] \gg false, s[\text{x}] \gg true\} & \text{read(r5, y, a3)}\{s[\text{y}] \gg false, s[\text{x}] \gg true\}, \\
\{S[out] \neq null, s[\text{y}] \gg false, s[\text{x}] \gg true\} & \text{call(c5, out, println(), r5)}\{s[\text{y}] \gg false, s[\text{x}] \gg true\}, \\
\{s[\text{y}] \gg false, s[\text{x}] \gg true\} & \text{read(r4, x, a2x)}\{s[\text{y}] \gg false, s[\text{x}] \gg true\}, \\
\{S[out] \neq null, s[\text{y}] \gg false, s[\text{x}] \gg true\} & \text{call(c4, out, println(), r4)}\{s[\text{y}] \gg false, s[\text{x}] \gg true\}
\end{array}
$$

Then we examine refactored code which is the same from both refactoring tools and it is shown in the following listing:

```
class C6Accepted {

    public void m1() {
        boolean y = true;

        y = false;
        System.out.println(y);
        System.out.println((y=true));
    }
}
```

Refactored code in Java.

```
Program(
    { method(C6Accepted.m1()) }, {
    entryPoint(e1, m),
    assign(a1, y, true),

    assign(a3, y, false),
    call(c5, out,println(), r5), read(r5, y, a3)
    call(c4, out,println(), r4), read(r2y', y, a2y'), assign(a2y', y,true),
    exitPoint(e2, m)
})
```

Refactored code in SDFG.

Example #2, refactored code.

Applying Hoare logic on the refactored SDFG program we get the following steps and we extract the pre-condition and post-condition for every `Stmt`.

$$
\begin{aligned}
&\Gamma, s', h'; I \vdash \\
\{true\}\ &\text{assign(a1,y, true)}\{s'[\text{y}] \gg true\}, \\
\{s'[\text{y}] \gg true\}\ &\text{assign(a3, y, false)}\{s'[\text{y}] \gg false\}, \\
\{s'[\text{y}] \gg true\}\ &\text{read(r5, y, a2y')}\{s'[\text{y}] \gg false\}, \\
\{S'[out] \neq null, s'[\text{y}] \gg false\}\ &\text{call(c5, out, println(), r5)}\{s'[\text{y}] \gg false\}, \\
\{s'[\text{y}] \gg false\}\ &\text{assign(a2y', y, true)}\{s'[\text{y}] \gg true\}, \\
\{s'[\text{y}] \gg true\}\ &\text{read(r2y', y, a2y')}\{s'[\text{y}] \gg true\}, \\
\{S'[out] \neq null, s'[\text{y}] \gg true\}\ &\text{call(c4, out, println(), r2y')}\{s'[\text{y}] \gg true\}
\end{aligned}
$$

Comparing the two post-conditions with respect to the Inline Local refactoring, we accept the missing predicate that refers to x as we discussed previously. Furthermore, we can see that the data dependency from the $r5$ `read`$(\cdot)$ has not changed in this version. Consequently, we move on the last post-condition of the method where we see that the value of y is different than the original.

However, in this example the variable y is local, as a result, the different value of the variable is not used and the code behaviour is not affected. On the other hand, if the variable is a field the change could be detected by other parts of the code. Since the change is not detectable inter-procedurally SDFG cannot detect this change in dependencies, this limitation was reported in Section 5.2.

□

Convert Local Variable To Field

Considering the effect of the Convert Local Variable To Field refactoring on the code, we expect that the environment will change since the refactored program contains two additional field declarations. The same is true for the store since the targeted variable is removed. Finally, the heap also changes since the object declaration has changed. The field is accessed through the object, so it is stored in the heap. Also there is a new mapping between the new lock and an instance of the object class.

Lemma 6.3.8. *The differences of the refactored program by* Convert Local Variable To Field *in comparison with the original program are described by the following equations:*

$$
\begin{aligned}
\Gamma' &= \Gamma + f + lock_f, \\
s' &= s - x, \\
h' &= h + (f : s[x]) + (l : alloc(Object()))
\end{aligned}
\tag{6.10}
$$

Theorem 6.3.9. *When the* Convert Local Variable To Field *refactoring is applied, the inference rules 5.24 and 5.25 ensure that there are no races to the new field.*

Proof. Since we will argue about races, we will examine how the original and the refactored program behave in concurrent execution.

Due to the separation logic formula 2.2 and because these are two accesses to different instances of the variable x the one from store s_1 and the other from s_2, we have:

$$\begin{gathered}\Gamma, s_1, s_2, h; I \vdash \\ \{true\}\ \text{read(id1, x, aId1)} \mid\mid \text{assign(id2, x, rId2)}\ \{(s_1[x] \gg [aId_1]) * (s_2[x] \gg [aId_2])\}\end{gathered} \tag{6.11}$$

Using inference rule 5.24 and 5.25 we get the respective refactored `Stmts`.

$$\begin{aligned}
&\Gamma, s_1, s_2, h; I \vdash \\
\{true\}\ &\text{acquireLock(idl, l, id1), read(id1, f, aId1),releaseLock(idl, l, id1)} \\
&\{(h'[f] \gg [aId_1], h'[f] \gg h'[l], h'[f] \ll h'[l])\} \\
&\mid\mid \\
\{true\}\ &\text{acquireLock(idl, l, id2), assign(id2, f, rId2), releaseLock(idl, l, id2)} \\
&\{(h'[f] \gg [aId_1]), h'[f] \gg h'[l], h'[f] \ll h'[l])\} \\
&\text{From 6.1 and 2.2} \implies \\
\{true\}\ &\text{atomic(read(id1, f, aId1))} \mid\mid \text{atomic(assign(id2, f, rId2))}\ \{(h'[f] \gg [aId_1]) * (h'[f] \gg [aId_2])\}
\end{aligned} \tag{6.12}$$

under the assumption that all accesses to f are in the same `synchronized` block. □

6.4 Claims

In this chapter we presented a proof outline consisting of lemmas, theorems and proof traces. The proof traces contain also an example that demonstrates how expect the proof to work.

Instead of proving behaviour preservation for the whole program, we suggest proving that the inference rules preserve behaviour. We claim that this simplification will lead to a complete proof under the assumption that dependency preservation leads to behaviour preservation because the inference rules capture all the changed dependencies. However, since our proof outline is based on the SDFG language, the correctness of converting from Java to SDFG is also required for the proof to hold.

Proving the correctness of the converter and then that every inference rule preserves behaviour is left as future work. This would complete the proof of correctness of CARR.

Chapter 7

Conclusion

In this work we presented three refactorings implemented by Eclipse that introduce new behaviour when the refactored code is run concurrently, Move Method, Convert Local Variable To Field and Inline Local. We focused on understanding when the behaviour is changed and provide a configurable formal model in the format of an intermediate language, SDFG, that can be used for semantic verification. We implemented the converter from Java to SDFG and integrated it to a refactoring tool called CARR to make the tool concurrency aware.

We formalized the configuration of the SDFG verification process as inference rules and argue that the correctness of the refactoring tool is implied by the correctness of the inference rules.

Given assumptions and limitations, we provided evidence that the SDFG and CARR work as expected. Finally, we outlined a proof of correctness of the inference rules using separation logic. The formal proof is left for future work.

The previous comprise the following contributions:

- SDFG consists a common theory that is used to argue about behaviour preservation,
- CARR is a prototype that uses the SDFG to implement concurrency aware refactorings configured by the mapping functions per refactoring,
- The proof outline suggests an idea of a proof, it puts forth a proving framework that is parameterised based on the refactoring. The proof is completed by the individual proofs of the inference rules.

To the best of our knowledge this is the first time that a configurable structure is used as a framework for refactoring verification. While there are still challenges to be solved, for instance, to make it more scalable and to resolve all potential exceptions, our work shows the value of this intermediate language by demonstrating how easily the framework can be extended to other refactorings.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, dr. Jurgen Vinju, who inspired me, helped me to put my ideas into perspective and without his guidance I would not be able to complete this project. Also I would like to thank Davy Landman for providing the OFG converter and helping me while writing this thesis. Finally, I am grateful for my friends and family who supported and believed in me, especially for Konstantinos who is always by my side.

Bibliography

[1] The Object Flow Graph. In *Reverse Engineering of Object Oriented Code*, Monographs in Computer Science, pages 21–41. Springer New York, 2005.

[2] D. Dig. A refactoring approach to parallelism. *Software, IEEE*, 28(1):17–22, Jan 2011.

[3] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 397–407, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. *SIGPLAN Not.*, 42(10):1–18, October 2007.

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[6] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[7] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java®Language Specification, Java SE 7 Edition*, chapter 17. 2013.

[8] Christian Haack and Clément Hurlin. Separation logic contracts for a Java-like language with fork/join. In Springer-Verlag, editor, *12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008)*, volume 5140 of *Lecture Notes in Computer Science*, Urbana, États-Unis, 2008. Jose Meseguer and Grigore Rosu. Mobius.

[9] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[11] C. A. R. Hoare. Proof of a program: Find. *Commun. ACM*, 14(1):39–45, January 1971.

[12] Susan Horwitz. Precise Flow-insensitive May-alias Analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997.

[13] C.B. Jones. Wanted: a compositional approach to concurrency. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, Monographs in Computer Science, pages 5–15. Springer New York, 2003.

[14] Hannes Kegel and Friedrich Steimann. Systematically Refactoring Inheritance to Delegation in Java. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 431–440, New York, NY, USA, 2008. ACM.

[15] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 0:168–177, 2009.

[16] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Easy meta-programming with rascal. In JoãoM. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin Heidelberg, 2011.

[17] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[18] Richard Mayr. Weak bisimulation and model checking for Basic Parallel Processes. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 88–99. Springer Berlin Heidelberg, 1996.

[19] Hanne Riis Nielson and Flemming Nielson. Semantics with Applications: A Formal Introduction, 1992.

[20] Emma Nilsson-Nyman, Görel Hedin, Eva Magnusson, and Torbjörn Ekman. Declarative Intraprocedural Flow Analysis of Java Source Code. *Electronic Notes in Theoretical Computer Science*, 238(5):155–171, 2009. Proceedings of the 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008).

[21] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1–3):271 – 307, 2007. Festschrift for John C. Reynolds's 70th birthday.

[22] William F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[23] Matthew Parkinson, Richard Bornat, and Peter O'Hearn. Modular Verification of a Non-blocking Stack. *SIGPLAN Not.*, 42(1):297–302, January 2007.

[24] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002.

[25] Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct Refactoring of Concurrent Java Code. In Theo D'Hondt, editor, *ECOOP 2010 — Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 225–249. Springer Berlin Heidelberg, 2010.

[26] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and Extensible Renaming for Java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 277–294, New York, NY, USA, 2008. ACM.

[27] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege Moor. Stepping Stones over the Refactoring Rubicon. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 369–393, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] Manu Sridharan, Satish Chandra, Julian Dolby, StephenJ. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 196–232. Springer Berlin Heidelberg, 2013.

[29] Friedrich Steimann and Andreas Thies. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In Sophia Drossopoulou, editor, *ECOOP 2009 — Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 419–443. Springer Berlin Heidelberg, 2009.

[30] Frank Tip. Refactoring Using Type Constraints. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, pages 1–17, Berlin, Heidelberg, 2007. Springer-Verlag.

[31] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for Generalization Using Type Constraints. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 13–26, New York, NY, USA, 2003. ACM.

[32] Viktor Vafeiadis and Matthew Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In LuÃŋs Caires and VascoT. Vasconcelos, editors, *CONCUR 2007 — Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer Berlin Heidelberg, 2007.

[33] Jaroslav Ševčík and David Aspinall. On Validity of Program Transformations in the Java Memory Model. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 27–51, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for Reentrancy. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 173–182, New York, NY, USA, 2009. ACM.

[35] Tim Wood and Sophia Drossopoulou. Refactoring Boundary. In Andrew V. Jones and Nicholas Ng, editors, *2013 Imperial College Computing Student Workshop*, volume 35 of *OpenAccess Series in Informatics (OASIcs)*, pages 119–127, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Appendix A

JastAdd Refactoring Tool

The JastAdd Refactoring Tool (JRRT) [25, 26, 27] could not be used for this thesis due to the changes currently done in JastAddJ, *"JastAddJ is intended to be an extensible compiler, however right now we are changing things in JastAddJ rapidly and breaking backward compatibility* [1]*"*.

In the following paragraphs we describe the effort to set up the JRRT to be compatible with the current version of JastAddJ. We had to set up in the eclipse workspace the following projects:

- JastAdd2 - [commit: 17eb133] [1]
- JastAddJ - [commit: aedcfa4] [2]
- ControlFlowGraph [revision: 9422] [3]
- Refactoring - [revision: r41] [4]

Setting up the projects

JastAdd2 and JastAddJ are actively maintained so their set up was easy following the instructions. However, the other two projects needed changes in order to be compatible with the new version of JastAdd2 and JastAddJ.

The first step included the update of the build.xml file to be compatible with the new folder structure of the projects and Java 7. The build.xml was fixed using the build.xml from JastAddJ for Java 7. The following changes when needed to account for the new features of Java 7 and the changes in the grammar of JastAddJ.

1. AssertStmt: The AssertStmt was changed in the grammar and its expressions were labelled as *Condition* and *Message*.

   ```
   AssertStmt : Stmt ::= first:Expr [Expr]; //old version
   AssertStmt : Stmt ::= Condition:Expr [Message:Expr]; //current version
   ```

2. MultiCatch: Added the MultiCatch attribute which was not there in Java 1.4.

   ```
   eq CatchParameterDeclaration.in() = empty();
   CatchParameterDeclaration implements CFGNode;

   eq MultiCatch.handlesAllUncheckedExceptions()
       = handles(typeRuntimeException()) && handles(typeError());
   ```

[1] https://bitbucket.org/jastadd/jastadd2
[2] https://bitbucket.org/jastadd/jastaddj
[3] http://svn.cs.lth.se/svn/jastadd-oxford/projects/trunk/JastAddExtensions/ControlFlowGraph/
[4] https://code.google.com/p/jrrt/

```
eq MultiCatch.handlesUncheckedException()
    = handles(typeRuntimeException())
    || handles(typeError())
    || getParameter().type().instanceOf(typeRuntimeException())
    || getParameter().type().instanceOf(typeError());
```

3. Set: After code generation the class named *Set* conflicts with the import *java.util.Set* which is added automatically. As a result this class was renamed to OldSet and updated other uses.

With these changes the project ControlFlowGraph was generated and compiled successfully.
The set up of the Refactoring project was not successful.

1. Sons: This is a feature of JastAdd which was removed from JastAddJ, so we commented it out from *util/Call.jadd*.

2. isImplicitConstructor: This method was generated from two different aspects (*jastaddJ/-java4/frontend/LookupConstructor.java* and *util/TypeExt* and could not be refined.

3. WithStmt: WithStmt is an attribute of the extended language [27] so the Aspect PreciseThrow, which belongs to JastAddJ, does not contain the equation for the synthesized attribute modifiedInScope() as a result it was added:

```
eq WithStmt.modifiedInScope(Variable var) {
    return getStmt().modifiedInScope(var);
}
```

4. PrettyPrinting: The aspect PrettyPrinting was missing from JastAddJ so it could not be refined.

5. setPackageDecl(): The aspect UndoRefines refine methods to facilitate the undo functionality. The methods of the aspect was temporally commented out.

6. CatchParameterDeclaration: The attribute MultiCatch is a subclass of abstract CatchClause as a result it needs to implement methods that return ParameterDeclaration as the BasicCatch does. However, MultiCatch has as child the CatchParameterDeclaration which is not compatible with it. To fix this we made a small change in the grammar of Java 7 to make the CatchParameterDeclaration a subclass of ParameterDeclaration.

7. getSuperClassAccessOpt(): The renaming of some methods in the new implementation of JastAddJ one resulted in missing methods. When it was possible the code that called these methods was refined to call the right name. Otherwise, an additional method was added with the old name that wrapped the current version.

8. InterfaceDecl and ClassDecl: In this classes, different internal methods use an iterator that return the extended or implemented interfaces respectively. This iterator returns either InterfaceDecl (subclass of ReferrenceType) or TypeDecl (subclass of ASTNode). As a result, the iterator could not be changed in a way that was compatible with both cases.

These problems could be solved if we could refine the conflicting methods. However, the refinement of methods other than getters was not possible as one of the writers of JastAdd Görel Hedin responded to our request: *"Concerning refinement of internal methods. You should be able to refine getters for children, but you can't refine other internal methods."*

Appendix B

The Sieve of Eratosthenes

This algorithm is named after Eratosthenes of Cyrene, a Greek mathematician; the sieve of Eratosthenes is a simple algorithm for finding all prime numbers up to any given limit using an array-like structure. The algorithm iteratively marks the the multiples of each prime as composite, starting with the multiples of 2.

For every prime, the algorithm marks all the prime's multiples by advancing in the array with a constant step that is equal to this prime. The absence of division is the characteristic that makes this algorithm efficient. We assume the following terms of mathematics:

Prime: is a natural number which has exactly two distinct natural number divisors, 1 and itself.

Composite: is a natural number which is not a prime.

Multiple of x: is a natural number which can be produced by multiplying x with another natural number larger than 1.

B.1 Algorithm

Listing B.1 describes the algorithm of the sieve of Eratosthenes and Figure B.1 illustrates an example execution with upper limit the number 10.

Listing B.1: "The sieve of Eratosthenes"

```
void sieveOfEratosthenes(int n){
    boolean[] sieve = new boolean[n];
    boolean[0] = false;
    boolean[1] = false;
    do{
        for(int j = 2*prime; j < n+1; j+=prime){
                    sieve[j] = false;
                }
                prime = nextPrime(sieve, prime);
        }while(prime != 0);
}

int nextPrime(boolean[] sieve, int prime){
    for(int i = prime + 1; i < sieve.length; i++){
        if(sieve[i])
            return i;
    }
    return 0;
}
```

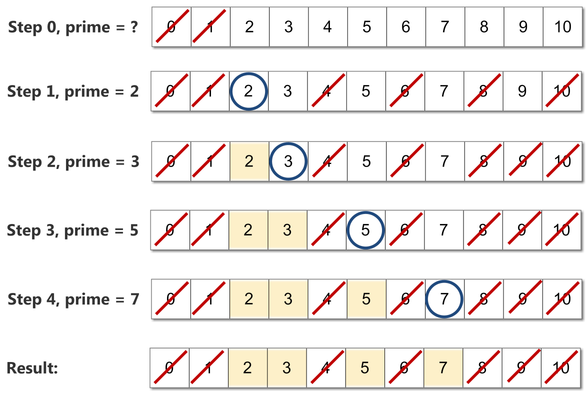


Figure B.1: The steps of the Sieve of Eratosthenes algorithm applied on a sieve of size 10.

Appendix C

SDFG Converter

For the converter implementation we used Rascal and worked on the Java Abstract Syntax Tree (AST)[1]. After extracting the declarations and taking into account the default constructors, the methods are normalized so they do not include any class declarations. An additional step is used to map the method declarations and the exceptions they throw, this is needed for the conversion of the **try-catch** statement. We used pattern matching to visit expressions and statements and extract the `Stmts`.

C.1 Java 2 SDFG

```
module lang::sdfg::converter::Java2SDFG

import IO;
import Set;
import List;
import String;
import Relation;

import lang::java::jdt::m3::AST;
import lang::java::m3::TypeSymbol;

import lang::sdfg::ast::SynchronizedDataFlowGraphLanguage;

import lang::sdfg::converter::GatherStmtFromStatements;
import lang::sdfg::converter::GatherStmtFromExpressions;

import lang::sdfg::converter::util::State;
import lang::sdfg::converter::util::Getters;
import lang::sdfg::converter::util::ExceptionManagement;
import lang::sdfg::converter::util::EnvironmentManagement;
import lang::sdfg::converter::util::TypeSensitiveEnvironment;

Program createSDFG(loc project) = createSDFG(createAstsFromEclipseProject(project, true));
Program createSDFG(set[Declaration] asts) {
    decls = getDeclarations(asts);
    stmts = getStatements(asts, decls);
    return program(decls, stmts);
}
```

[1] https://github.com/cwi-swat/rascal/blob/master/src/org/rascalmpl/library/lang/java/m3/AST.rsc

```
set[Decl] getDeclarations(set[Declaration] asts)
    = { Decl::attribute(v@decl,(volatile() in (f@modifiers ?  {}))) | /f:field(t,frags) <- asts,
v <- frags}
    + { Decl::method(m@decl, [p@decl | p:parameter(t,_,_) <- params], determineLock(m)) |
    /m:Declaration::method(_,_, list[Declaration] params, _, _)  <- asts}
    + { Decl::method(m@decl, [p@decl | p:parameter(t,_,_) <- params], determineLock(m)) |
    /m:Declaration::method(_,_, list[Declaration] params, _)  <- asts}
    + { Decl::constructor(c@decl, [p@decl | p:parameter(t,_,_) <- params]) |
    /c:Declaration::constructor(_, list[Declaration] params, _,_)  <- asts}
    // add implicit constructor

    + { Decl::constructor((c@decl)[scheme="java+constructor"] + "<name>()", []) | /c:class(name,
_, _, b) <- asts, !(Declaration::constructor(_, _, _, _) <- b)}
    ;

set[Stmt] getStatements(set[Declaration] asts, set[Decl] decls) {

    initialized = gatherInitializations(asts);
    fieldsPerClass = (c@decl.path :  {v@decl | field(t,frags) <- b, v <- frags}| /c:class(name,
_, _, b) <- asts);
    inheritingClasses = (c@decl.path :  {sc@decl.path | simpleType(sc) <- extends}| /c:class(name,
extends, _, b) <- asts);

    //Gather all methods and constructors

    allMethods
        = [ m | /m:Declaration::method(_,_,_,_,_) <- asts]
        + [Declaration::method(t, n, p, e, empty())[@decl=m@decl][@src = m@src] |
        /m:Declaration::method(Type t,n,p,e) <- asts]
        + [Declaration::method(simpleType(simpleName(n)), n, p, e,
        Statement::block((initialized[extractClassName(m@decl)] ?  []) + b))
        [@decl=m@decl][@src=m@src] | /m:Declaration::constructor(str n,p,e,  Statement::block(b))
<- asts]
        + [Declaration::method(simpleType(simpleName(n)), n, [], [], block(initialized[c@decl.path]
?  []))[@decl=(c@decl)[scheme="java+constructor"] + "<n>()"][@src = c@src] | /c:class(n, _, _,
b) <- asts, !(Declaration::constructor(_, _, _, _) <- b)]
    ;

    //Flatten nested classes
    allMethods = visit(allMethods) {
        case declarationExpression(Declaration::class(_)) => Expression::null()
        case declarationExpression(Declaration::class(_,_,_,_)) => Expression::null()
        case declarationExpression(Declaration::enum(_,_,_,_)) => Expression::null()

        case declarationStatement(Declaration::class(_)) => empty()
        case declarationStatement(Declaration::class(_,_,_,_)) => empty()
        case declarationStatement(Declaration::enum(_,_,_,_)) => empty()
    };

    set[Stmt] result = {};
    set[loc] volatileFields = {vField | attribute(vField, true) <- decls};

    //Gather exceptions
    for (m:Declaration::method(_, _, _, ex, _) <- allMethods) {
        if(ex != []){
            exceptions[m@decl] = {e@decl.path | e <- ex};
        }
    }
    for (m:Declaration::method(_, _, parameters, ex, b) <- allMethods) {
        set[Stmt] methodStmts = {entryPoint(m@src, m@decl)};

        //determine lock
        rel[loc,loc] locks = {};
        for(Decl::method(id, _, l) <- decls){
            if((id.path == m@decl.path) && (l != unlocked)){
                locks += {<m@src, l>};
                methodStmts += {read(m@src, l, emptyId)};
            }
        }
        //set up environment with parameters and fields
        map[loc, set[loc]] env = ( p@decl :  {p@src} |
        p <- parameters) + ( field :  {emptyId} | field <- fieldsPerClass[extractClassName(m@decl)]
?  {})
        + ( field :  {emptyId} | sc <- inheritingClasses[extractClassName(m@decl)] ?  {}, field
<- fieldsPerClass[sc] ?  {});
        map[loc, set[loc]] typesOfParam = index({ <getTypeDeclFromTypeSymbol(p@typ),p@decl> |
p <- parameters, isClass(p@typ)});
        map[loc,TypeSensitiveEnvironment] typesOf = ( t :  typeEnv(typesOfParam[t],{}) | t <-
typesOfParam);

        rel[loc,loc] acquireActions = locks;
        FlowEnvironment fenv;

        top-down-break visit(b) {
            case Expression e :  <methodStmts, _, env, typesOf, acquireActions, _> =
            gatherStmtFromExpressions(e, env, typesOf, volatileFields, acquireActions, methodStmts);
            case Statement s :  <methodStmts, env, typesOf, acquireActions, fenv, _> =
            gatherStmtFromStatements(s, env, typesOf, volatileFields, acquireActions, methodStmts);
        }
        //return environment
        exitSrc = m@src;
        exitSrc.offset = m@src.offset + m@src.length -1;
        for(<src, l> <- locks){
            methodStmts += addAndUnlock(methodStmts, src, l);
        }
        methodStmts += addAndLock({exitPoint(exitSrc, m@decl)},
        acquireActions + getAcquireActions(getReturnState(fenv)));
        result+= methodStmts;
    }
    return result;
}


private loc determineLock(Declaration method){
    loc lock = unlocked;
    if(synchronized() in  (method@modifiers ?  {})){
        if(static() in (method@modifiers ?  {})){
            str lockPath = extractClassName(method@decl) + ".class";
            lock = |java+class:///|+lockPath;
        }
        else{
            str lockPath = extractClassName(method@decl) + "/this";
            lock = |java+parameter:///|+lockPath;
        }
    }
    return lock;
}

private map[str, list[Statement]] gatherInitializations(set[Declaration] asts)
    = (c@decl.path :  [expressionStatement(v) | field(t,frags) <- b, v <- frags] | /c:class(name,
_, _, b) <- asts);
```

C.2 Statement Visitor

```
module lang::sdfg::converter::GatherStmtFromStatements

import IO;

import lang::java::jdt::m3::AST;

import lang::sdfg::ast::SynchronizedDataFlowGraphLanguage;
import lang::sdfg::converter::GatherStmtFromExpressions;

import lang::sdfg::converter::util::State;
import lang::sdfg::converter::util::Getters;
import lang::sdfg::converter::util::ExceptionManagement;
import lang::sdfg::converter::util::EnvironmentManagement;
import lang::sdfg::converter::util::TypeSensitiveEnvironment;

//assert(Expression expression)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
    FlowEnvironment, map[str, State]]
gatherStmtFromStatements(Statement s:\assert(exp), map[loc, set[loc]] env,
map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts)
    = gatherStmtFromStatements(\assert(exp, Expression::null()), env, typesOf, volatileFields,
acquireActions, stmts);

//assert(Expression expression, Expression message)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
FlowEnvironment, map[str, State]] gatherStmtFromStatements(Statement s:\assert(exp, message),
map[loc, set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields,
rel[loc,loc] acquireActions, set[Stmt] stmts) {
    <stmts, potential, env, acquireActions, exs> = gatherStmtFromExpressions(exp, env, typesOf,
volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireAction(potential, volatileFields);

    <stmtsM, potential, envM, acquireActionsM, exsM> = gatherStmtFromExpressions(message, env,
acquireActions, volatileFields, acquireActions, stmts);
    stmtsM += potential;
    acquireActionsM += extractAcquireAction(potential, volatileFields);
    exs = mergeState(exs, exsM);
    //the volatile access from the message are not counted since if the message appears nothing
else is going to be executed
    //The assert is a possible an exit point, in case of finally we can see it as a return
    env = merge(env,envM);
    return <stmts, env, typesOf, acquireActions, initializeReturnState(stmtsM, envM, typesOfM,
acquireActionsM), exs>;
}

//block(statements)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
FlowEnvironment, map[str, State]] gatherStmtFromStatements(Statement s:\block(sB), map[loc, set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    exs = ();
    fenv = emptyFlowEnvironment();
    for(stmt <- sB) {
        <stmts, env, typesOf, acquireActions, fenvS, exsS> = gatherStmtFromStatements(stmt, env,
typesOf, volatileFields, acquireActions, stmts);
        fenv = mergeFlow(fenv, fenvS);
        exs = mergeExceptions(exs, exsS);
        if(breakingControlFlow(stmt))
            break;
    }
    return <stmts, env, typesOf, acquireActions, fenv, exs>;
}

//break()
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\break(), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts)
    = <{}, (), (), {}, initializeBreakState(stmts, env, typesOf, acquireActions), ()>;

//break("")
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\break(""), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts)
    = <{}, (), (), {}, initializeBreakState(stmts, env, typesOf, acquireActions), ()>;

//break(str label)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\break(exp), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts) {
    if(exp == "")
        fail;
    assert false :  "Labeled statement (break) found!!!";
    return <{}, (), (), {}, initializeBreakState(stmts, env, typesOf, acquireActions), ()>;
}

//continue()
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\continue(), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts)
    = <{}, (), (), {}, initializeContinueState(stmts, env, typesOf, acquireActions), ()>;

//continue(str label)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\continue(_), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts) {
    assert false :  "Labeled statement (continue) found!!!";
    return <{}, (), (), {}, initializeContinueState(stmts, env, typesOf, acquireActions), ()>;
}

//do(Statement body, Expression condition)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\do(b, cond), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts) {

    //executed once all the reads and assigns added missing connections to itself
    <stmts, env, typesOf, acquireActions, fenv, exitExs> = gatherStmtFromStatements(b, env, typesOf,
volatileFields, acquireActions, stmts);
    stmts += getStmts(getContinueState(fenv));
    env = merge(env, getEnvironment(getContinueState(fenv)));
    typesOf = mergeTypesEnvironment(typesOf, getTypesEnvironment(getContinueState(fenv)));
    acquireActions += getAcquireActions(getContinueState(fenv));

    exitStmts += getStmts(getBreakState(fenv));
    exitEnv = getEnvironment(getBreakState(fenv));
    exitTypesOf = getTypesEnvironment(getBreakState(fenv));
    exitAcquireActions = getAcquireActions(getBreakState(fenv));

    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(cond, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);
    exitStmts += stmts;
    exitExs = mergeExceptions(exitExs, exs);
    exitEnv = merge(exitEnv, env);
    exitTypesOf = mergeTypesEnvironment(exitTypesOf, typesOf);
    exitAcquireActions += acquireActions;

    <stmts, _, _, _, fenvB, exs> = gatherStmtFromStatements(b, env, typesOf, volatileFields,
acquireActions, stmts);
    exitStmts += getStmts(getBreakState(fenvB));
    exitEnv = merge(exitEnv, getEnvironment(getBreakState(fenvB)));
    exitTypesOf = mergeTypesEnvironment(exitTypesOf, getTypesEnvironment(getBreakState(fenvB)));
    exitAcquireActions += getAcquireActions(getBreakState(fenvB));
    exitExs = mergeExceptions(exitExs, exs);
    fenv = mergeFlow(fenv, fenvB);

    return <exitStmts, exitEnv, exitTypesOf,  exitAcquireActions,
    initializeReturnState(getReturnState(fenv)), exs>;
}
```

```
//foreach(Declaration parameter, Expression collection, Statement body)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\foreach(parameter, collection, body),
map[loc, set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields,
rel[loc,loc] acquireActions, set[Stmt] stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(collection,
env, typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);

    stmts += addAndLock({Stmt::assign(parameter@src, parameter@decl, collection@src)}, acquireActions);
    env[parameter@decl] = {parameter@src};

    exitStmts = stmts;
    exitEnv = env;
    exitTypesOf = typesOf;
    exitAcquireActions = acquireActions;
    exitExs = exs;

    <stmts, envB, typesOfB, acquireActionsB, fenvB, exsB> = gatherStmtFromStatements(body, env,
typesOf, volatileFields, acquireActions, stmts);
    exitStmts += getStmts(getBreakState(fenvB));
    exitEnv = merge(exitEnv, getEnvironment(getBreakState(fenvB)));
    exitTypesOf = mergeTypesEnvironment(exitTypesOf, getTypesEnvironment(getBreakState(fenvB)));
    exitAcquireActions += getAcquireActions(getBreakState(fenvB));
    exitExs = mergeExceptions(exitExs, exsB);

    stmts += getStmts(getContinueState(fenvB));
    envB = merge(envB, getEnvironment(getContinueState(fenvB)));
    typesOfB = mergeTypesEnvironment(exitTypesOf, getTypesEnvironment(getContinueState(fenvB)));
    acquireActionsB += getAcquireActions(getContinueState(fenvB));

    //reset the parameter
    stmts += addAndLock({Stmt::assign(parameter@src, parameter@decl, collection@src)}, acquireActionsB);
    envB[parameter@decl] = {parameter@src};
    exitStmts += stmts;
    exitEnv = merge(exitEnv, envB);
    exitTypesOf = mergeTypesEnvironment(exitTypesOf, typesOfB);
    exitAcquireActions += acquireActionsB;

    <stmts, _, _, _, fenvB, exs> = gatherStmtFromStatements(body, envB, typesOfB, volatileFields,
acquireActionsB, stmts);
    exitStmts += getStmts(getBreakState(fenvB));
    exitEnv = merge(exitEnv, getEnvironment(getBreakState(fenvB)));
    exitTypesOf = mergeTypesEnvironment(exitTypesOf, getTypesEnvironment(getBreakState(fenvB)));
    exitAcquireActions += getAcquireActions(getBreakState(fenvB));
    exitExs = mergeExceptions(exitExs, exsB);

    return <exitStmts, exitEnv, exitTypesOf,  exitAcquireActions,
    initializeReturnState(getReturnState(fenvB)), exs>;

}

//for(initializers, Expression condition, updaters, Statement body)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\for(initializers, cond, updaters, body),
map[loc, set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields,
rel[loc,loc] acquireActions, set[Stmt] stmts)
    = dealWithLoopsConditionFirst(initializers, cond, updaters, body, env, typesOf, volatileFields,
acquireActions, stmts);

//for(initializers, updaters, Statement body)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\for(initializers, updaters, body), map[loc,
set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc]
acquireActions, set[Stmt] stmts)
    = dealWithLoopsConditionFirst(initializers, Expression::\null(), updaters, body, env, typesOf,
volatileFields, acquireActions, stmts);

//if(Expression condition, Statement thenB)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\if(cond, thenB), map[loc, set[loc]] env,
map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(cond, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);

    <stmts, envOpt, typesOpt, acquireActionsThen, fenv, exsC> = gatherStmtFromStatements(thenB,
env, typesOf, volatileFields, acquireActions, stmts);
    exs = mergeExceptions(exs,exsC);
    env = merge(env,envOpt);
    typesOf = mergeTypesEnvironment(typesOf, typesOpt);

    return <stmts, env, typesOf, acquireActions + acquireActionsThen, fenv, exs>;
}

//if(Expression condition, Statement thenB, Statement elseB)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\if(cond, thenB, elseB), map[loc, set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(cond, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);

    <stmtsIf,   envIf,   typesIf,   acquireActionsIf,   fenvIf,   exsIf> =
    gatherStmtFromStatements(thenB, env, typesOf, volatileFields, acquireActions, stmts);
    <stmtsElse, envElse, typesElse, acquireActionsElse, fenvElse, exsElse> =
    gatherStmtFromStatements(elseB, env, typesOf, volatileFields, acquireActions, stmts);

    env = merge(envIf,envElse);
    typesOf = mergeTypesEnvironment(typesIf, typesElse);
    fenv = mergeFlow(fenvIf, fenvElse);
    exs = mergeExceptions(exsIf,exsElse);
    return <stmtsIf + stmtsElse, env, typesOf, acquireActionsIf + acquireActionsElse, fenv, exs>;
}
```

```
//label(str name, Statement body)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\label(_, _), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts) {
    assert false:  "Labeled block";
    return <stmts, env, typesOf, acquireActions, emptyFlowEnvironment(), ()>;
}

//return(Expression expression)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\return(exp), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(exp, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields); //needed for the finally

    return <{}, (), (), {}, initializeReturnState(stmts, env, typesOf, acquireActions), exs>;
}

//return()
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\return(), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts)
    = <{}, (), (), {}, initializeReturnState(stmts, env, typesOf, acquireActions), exs>;

//switch(Expression exp, statements)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\switch(exp, body), map[loc, set[loc]]
env, map[loc,TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(exp, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);

    exitStmts = stmts;
    exitEnv = ();
    exitTypesOf = ();
    exitAcquireActions = acquireActions;
    exitFenv = emptyFlowEnvironment();

    currentStmts = stmts;
    currentEnv = env;
    currentTypesOf = typesOf;
    currentAcquireActions = acquireActions;

    list[Statement] currentCase = [];

    hasDefault = false;
    for(stmt <- body) {
        switch(stmt) {
            case \case(_):{
                <currentStmts, currentEnv, currentTypeOf, currentAcquireActions, fenv, exsC>
=
                gatherStmtFromStatements(\block(currentCase), currentEnv, currentTypesOf,
                volatileFields, currentAcquireActions, currentStmts);
                if(isEmpty(getBreakState(fenv))) {
                    currentEnv = merge(env, currentEnv);
                    currentTypesOf = mergeTypesEnvironment(typesOf, currentTypesOf);
                }
                else{
                    exitStmts += getStmts(getBreakState(fenv));
                    exitEnv = merge(exitEnv, getEnvironment(getBreakState(fenv)));
                    exitTypesOf = mergeTypesEnvironment(exitTypesOf,
                    getTypesEnvironment(getBreakState(fenv)));
                    exitAcquireActions += getAcquireActions(getBreakState(fenv));
                    fenv = updateBreak(fenv,emptyState());

                    currentStmts = stmts;
                    currentEnv = merge(env, currentEnv);
                    currentTypesOf = mergeTypesEnvironment(typesOf, currentTypesOf);
                    currentAcquireActions += acquireActions;
                }
                exitFenv = mergeFlow(exitFenv, fenv);
                exs = mergeExceptions(exs, exsC);
                currentCase = [];
            }
            case  \defaultCase():{
                hasDefault = true;
                <currentStmts, currentEnv, currentTypeOf, currentAcquireActions, fenv, exsC>
=
                gatherStmtFromStatements(\block(currentCase), currentEnv, currentTypesOf, volatileFields,
                currentAcquireActions, currentStmts);
                if(isEmpty(getBreakState(fenv))) {
                    currentEnv = merge(env, currentEnv);
                    currentTypesOf = mergeTypesEnvironment(typesOf, currentTypesOf);
                }
                else{
                    exitStmts += getStmts(getBreakState(fenv));
                    exitEnv = merge(exitEnv, getEnvironment(getBreakState(fenv)));
                    exitTypesOf = mergeTypesEnvironment(exitTypesOf,
                    getTypesEnvironment(getBreakState(fenv)));
                    exitAcquireActions += getAcquireActions(getBreakState(fenv));
                    fenv = updateBreak(fenv,emptyState());

                    currentStmts = stmts;
                    currentEnv = merge(env, currentEnv);
                    currentTypesOf = mergeTypesEnvironment(typesOf, currentTypesOf);
                    currentAcquireActions += acquireActions;
                }
                exitFenv = mergeFlow(exitFenv, fenv);
                exs = mergeExceptions(exs, exsC);
                currentCase = [];
            }
            default:{
                currentCase += [stmt];
            }
        }
    }
    <currentStmts, currentEnv, currentTypeOf, currentAcquireActions, fenv, exsC> =
    gatherStmtFromStatements(\block(currentCase), currentEnv, currentTypesOf, volatileFields,
    currentAcquireActions, currentStmts);
    exitStmts += currentStmts;
    exitStmts += getStmts(getBreakState(fenv));
    exitEnv = merge(exitEnv, currentEnv);
    exitEnv = merge(exitEnv,getEnvironment(getBreakState(fenv)));
    exitTypesOf = mergeTypesEnvironment(exitTypesOf, currentTypeOf);
    exitTypesOf = mergeTypesEnvironment(exitTypesOf, getTypesEnvironment(getBreakState(fenv)));
    exitAcquireActions += currentAcquireActions;
    exitAcquireActions += getAcquireActions(getBreakState(fenv));
    fenv = updateBreak(fenv,emptyState());
    exitFenv = mergeFlow(exitFenv, fenv);
    exs = mergeExceptions(exs, exsC);
    if(!hasDefault) {
        exitEnv = merge(exitEnv, env);
        exitTypeOf = mergeTypesEnvironment(exitTypesOf, typesOf);
    }
    return <exitStmts, exitEnv, exitTypesOf, exitAcquireActions, exitFenv, exs>;
}

//synchronizedStatement(Expression lock, Statement body)
tuple[set[Stmt], map[loc, set[loc]], map[loc,TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:synchronizedStatement(l, body), map[loc,
set[loc]] env, map[loc,TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc]
acquireActions, set[Stmt] stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(l, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);

    loc vlock;
    for(w:read(_, name, _) <- potential) {
        vlock = name;
    }
    <stmts, env, typesOf, acquireActions, fenv, exsC> = gatherStmtFromStatements(body, env, typesOf,
volatileFields, {<s@src, vlock>} + acquireActions, stmts);

    stmts += addAndUnlock(stmts, s@src, vlock);
    exs = mergeExceptions(exs, exsC);
    return <stmts, env, typesOf, acquireActions, fenv, exs>;
}

//throw(Expression exp)
tuple[set[Stmt], map[loc, set[loc]], map[loc,TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\throw(exp), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts) {
    exs = (extractClassName(exp@decl) :  state(stmts, env, typesOf, acquireActions));
    return <stmts, (), (), {}, emptyFlowEnvironment(), exs>;
}
```

```
//\try(Statement body, catchClauses)
tuple[set[Stmt], map[loc, set[loc]], map[loc,TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\try(body, catchStatements), map[loc, set[loc]]
env, map[loc,TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, exitEnv, exitTypesOf, exitAcquireActions, exitFenv, exs> = gatherStmtFromStatements(body,
env, typesOf, volatileFields, acquireActions, stmts);
    exitStmts = stmts;
    exitExs = ();
    for(cs <- catchStatements) {
        <stmtsC, envC, typesOfC, acquireActionsC, fenvC, exs, exsC> = gatherStmtFromCatchStatements(cs,
volatileFields, exs);
        exitStmts += stmtsC;
        exitEnv = merge(exitEnv, envC);
        exitTypesOf = mergeTypesEnvironment(exitTypesOf, typesOfC);
        exitAcquireActions += acquireActionsC;
        exitExs = mergeExceptions(exitExs,exsC);
        exitFenv = mergeFlow(exitFenv, fenvC);
    }
    exitExs = mergeExceptions(exitExs,exs);
    return <exitStmts, exitEnv, exitTypesOf, exitAcquireActions, exitFenv, exitExs>;
}

//\try(Statement body, catchClauses, Statement \finally)
tuple[set[Stmt], map[loc, set[loc]], map[loc,TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\try(body, catchStatements, fin), map[loc,
set[loc]] env, map[loc,TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc]
acquireActions, set[Stmt] stmts) {
    <stmts, exitEnv, exitTypesOf, exitAcquireActions, exitFenv, exs> = gatherStmtFromStatements(body,
env, typesOf, volatileFields, acquireActions, stmts);
    exitStmts = stmts;
    exitExs = ();
    for(cs <- catchStatements) {
        <stmtsC, envC, typesOfC, acquireActionsC, fenvC, exs, exsC> = gatherStmtFromCatchStatements(cs,
volatileFields, exs);
        exitStmts += stmtsC;
        exitEnv = merge(exitEnv, envC);
        exitTypesOf = mergeTypesEnvironment(exitTypesOf, typesOfC);
        exitAcquireActions += acquireActionsC;
        exitExs = mergeExceptions(exitExs,exsC);
        exitFenv = mergeFlow(exitFenv, fenvC);
    }
    exitExs = mergeExceptions(exitExs,exs);
    //Run finally for every environemnt
    //exit
    <finStmts, exitEnv, exitTypesOf, exitAcquireActions, fenv,  exsE> = gatherStmtFromStatements(fin,
exitEnv, exitTypesOf, volatileFields, exitAcquireActions, stmts);
    exitExs = mergeExceptions(exitExs, exsE);
    //continue
    <stmts, envC, typesOfC, acquireActionsC, fenvC, exsC> =
    gatherStmtFromStatements(fin, getEnvironment(getContinueState(exitFenv)),
    getTypesEnvironment(getContinueState(exitFenv)), volatileFields,
    getAcquireActions(getContinueState(exitFenv)),
    getStmts(getContinueState(exitFenv)));
    finStmts += stmts;
    fenv = updateContinue(fenv, state(stmts, envC, typesOfC, acquireActionsC));
    fenv = mergeFlow(fenv, fenvC);
    exitExs = mergeExceptions(exitExs, exsC);
    //break
    <stmts, envB, typesOfB, acquireActionsB, fenvB, exsB> =
    gatherStmtFromStatements(fin, getEnvironment(getBreakState(exitFenv)),
    getTypesEnvironment(getBreakState(exitFenv)), volatileFields,
    getAcquireActions(getBreakState(exitFenv)),
    getStmts(getBreakState(exitFenv)));
    finStmts += stmts;
    fenv = updateBreak(fenv, state(stmts, envB, typesOfB, acquireActionsB));
    fenv = mergeFlow(fenv, fenvB);
    exitExs = mergeExceptions(exitExs, exsB);
    //return
    <stmts, envR, typesOfR, acquireActionsR, fenvR, exsR> =
    gatherStmtFromStatements(fin, getEnvironment(getReturnState(exitFenv)),
    getTypesEnvironment(getReturnState(exitFenv)), volatileFields,
    getAcquireActions(getReturnState(exitFenv)), getStmts(getReturnState(exitFenv)));
    finStmts += stmts;
    fenv = updateReturn(fenv, state(stmts, envR, typesOfR, acquireActionsR));
    fenv = mergeFlow(fenv, fenvR);
    exitExs = mergeExceptions(exitExs, exsR);
    return <finStmts, exitEnv, exitTypesOf, exitAcquireActions, fenv, exitExs>;
}

//\catch(Declaration exception, Statement body)
tuple[set[Stmt], map[loc, set[loc]], map[loc,TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State], map[str, State]] gatherStmtFromCatchStatements(Statement s:\catch(except, body),
set[loc] volatileFields, map[str, State] exs) {
    env = ();
    fenv = emptyFlowEnvironment();
    exitStmts = {};
    map[str, State] exsCatch = ();
    map[loc, TypeSensitiveEnvironment] typesOf = ();
    rel[loc,loc] acquireActions = {};
    visit(except) {
        case e:simpleName(_) :  {
            <exceptionState, exs> = getAndRemoveState(exs, e@decl.path);
            if(!isEmpty(exceptionState)) {
                <stmts, envCatch, typesOfCatch, acquireActionsCatch, fenvCatch, exsC> =
                gatherStmtFromStatements(body, getEnvironment(exceptionState),
                getTypesEnvironment(exceptionState), volatileFields, getAcquireActions(exceptionState),
                getStmts(exceptionState));
                exitStmts += stmts;
                env = merge(env,envCatch);
                typesOf = mergeTypesEnvironment(typesOf, typesOfCatch);
                exsCatch = mergeExceptions(exsCatch, exsC);
                fenv = mergeFlow(fenv, fenvCatch);
                acquireActions += acquireActionsCatch;
            }
            else{
                println("Unreached exception, <e@src>");
            }
        }
    }
    return <exitStmts, env, typesOf, acquireActions, fenv, exs, exsCatch>;
}

//\declarationStatement(Declaration declaration)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement ds:\declarationStatement(d), map[loc, set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    exs = ();
    fenv = emptyFlowEnvironment();
    top-down-break visit(d) {
        case Expression e :  {
            <stmts, _, env, typesOf, acquireActions, exsE> = gatherStmtFromExpressions(e, env,
typesOf, volatileFields, acquireActions, stmts);
            exs = mergeExceptions(exs, exsE);
        }
        case Statement s :  {
            <stmts, env, typesOf, acquireActions, fenvD, exsD> = gatherStmtFromStatements(s,
env, typesOf, volatileFields, acquireActions, stmts);
            exs = mergeExceptions(exs, exsD);
            fenv = mergeFlow(fenv, fenvD);
        }
    }
    return <stmts, env, typesOf, acquireActions, fenv, exs>;
}

//\while(Expression condition, Statement body)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:\while(cond, body), map[loc, set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts)
    = dealWithLoopsConditionFirst([], cond, [], body, env, typesOf, volatileFields, acquireActions,
stmts);

//\expressionStatement(Expression stmt)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:expressionStatement(e), map[loc, set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, _, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(e, env, typesOf,
volatileFields, acquireActions, stmts);
    return <stmts, env, typesOf, acquireActions, emptyFlowEnvironment(), exs>;
}

//\constructorCall(bool isSuper, Expression expr, arguments)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:constructorCall(isSuper, exp, args), map[loc,
set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc]
acquireActions, set[Stmt] stmts) {
     exs = ();
    for(arg <- args) {
        <stmts, potential, env, typesOf, acquireActions, exsA> = gatherStmtFromExpressions(arg,
env, typesOf, volatileFields, acquireActions, stmts);
        stmts += potential;
        acquireActions += extractAcquireActions(potential, volatileFields);
        exs = mergeState(exs,exsA);
    }
    <stmts, potential, env, typesOf, acquireActions, exsE> = gatherStmtFromExpressions(exp, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);
    exs = mergeState(exs,exsE);

    return <stmts, env, typesOf, acquireActions, emptyFlowEnvironment(), exs>;
}

//\constructorCall(bool isSuper, arguments)
tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement s:constructorCall(isSuper, args), map[loc,
set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc]
acquireActions, set[Stmt] stmts)
     = gatherStmtFromStatements(constructorCall(isSuper, Expression::null(), args), env, typesOf,
volatileFields, acquireActions, stmts);


tuple[set[Stmt], map[loc, set[loc]], map[loc,TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] gatherStmtFromStatements(Statement b:empty(), map[loc, set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts)
    = <stmts, env, typesOf, acquireActions, emptyFlowEnvironment(), ()>;

default tuple[set[Stmt], map[loc, set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
FlowEnvironment, map[str, State]] gatherStmtFromStatements(Statement b, map[loc, set[loc]] env,
map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    assert false :  "Unknown statement :  <b>";
    return <stmts, env, typesOf, acquireActions, emptyFlowEnvironment(), ()>;
}

tuple[set[Stmt], map[loc, set[loc]], map[loc,TypeSensitiveEnvironment], rel[loc,loc], FlowEnvironment,
map[str, State]] dealWithLoopsConditionFirst(list[Expression] initializers, Expression cond,
list[Expression] updaters, Statement body, map[loc, set[loc]] env, map[loc,TypeSensitiveEnvironment]
typesOf, set[loc] volatileFields, rel[loc, loc] acquireActions, set[Stmt] stmts) {
    exs = ();
    for(i <- initializers) {
        <stmts, _, env, typesOf, acquireActions, exsC> = gatherStmtFromExpressions(i, env, typesOf,
volatileFields, acquireActions, stmts);
        exs = mergeExceptions(exs, exsC);
    }
    <stmts, potential, env, typesOf, acquireActions, exsC> = gatherStmtFromExpressions(cond,
env, typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);
    exs = mergeExceptions(exs,exsC);
    exitStmts = stmts;
    exitEnv = env;
    exitTypesOf = typesOf;
    exitAcquireActions = acquireActions;
    exitExs = exs;
    <stmts, envB, typesOfB, acquireActionsB, fenvB, exsB> = gatherStmtFromStatements(body, env,
typesOf, volatileFields, acquireActions, stmts);
    exitStmts += stmts;
    exitEnv = merge(exitEnv, getEnvironment(getBreakState(fenvB)));
    exitTypesOf = mergeTypesEnvironment(exitTypesOf, getTypesEnvironment(getBreakState(fenvB)));
    exitAcquireActions += getAcquireActions(getBreakState(fenvB));
    exitExs = mergeExceptions(exitExs, exsB);

    stmts += getStmts(getContinueState(fenvB));
    envB = merge(envB, getEnvironment(getContinueState(fenvB)));
    typesOfB = mergeTypesEnvironment(exitTypesOf, getTypesEnvironment(getContinueState(fenvB)));
    acquireActionsB += getAcquireActions(getContinueState(fenvB));

    for(u <- updaters) {
        <stmts, _, envB, typesOfB, acquireActionsB, exsC> = gatherStmtFromExpressions(u, envB,
typesOfB, volatileFields, acquireActionsB, stmts);
        exitExs = mergeExceptions(exitExs, exsC);
    }
    exitStmts += stmts;

    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(cond, envB,
typesOfB, volatileFields, acquireActionsB, stmts);
    stmts += potential;
    exitExs = mergeExceptions(exitExs, exs);

    exitStmts += stmts;
    exitEnv = merge(exitEnv, env);
    exitTypesOf = mergeTypesEnvironment(exitTypesOf, typesOf);
    exitAcquireActions += getAcquireActions(getBreakState(fenvB));

    <stmts, _, _, _, fenvB, exs> = gatherStmtFromStatements(body, env, typesOf, volatileFields,
acquireActions, stmts);
    exitStmts += stmts;
    exitStmts += getStmts(getBreakState(fenvB));
    exitEnv = merge(exitEnv, getEnvironment(getBreakState(fenvB)));
    exitTypesOf = mergeTypesEnvironment(exitTypesOf, getTypesEnvironment(getBreakState(fenvB)));
    exitAcquireActions += getAcquireActions(getBreakState(fenvB));
    exitExs = mergeExceptions(exitExs, exsB);

    return <exitStmts, exitEnv, exitTypesOf,  exitAcquireActions,
    initializeReturnState(getReturnState(fenvB)), exs>;
}
```

C.3 Expression Visitor

```
module lang::sdfg::converter::GatherStmtFromExpressions

import IO;
import String;
import lang::java::jdt::m3::AST;
import lang::java::m3::TypeSymbol;
import lang::sdfg::ast::SynchronizedDataFlowGraphLanguage;

import lang::sdfg::converter::util::State;
import lang::sdfg::converter::util::Getters;
import lang::sdfg::converter::util::ExceptionManagement;
import lang::sdfg::converter::util::EnvironmentManagement;
import lang::sdfg::converter::util::TypeSensitiveEnvironment;

//The functions are ordered according to the rascal/src/org/rascalImpl/library/lang/java/m3/AST.rsc


//arrayAccess(Expression array, Expression index)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:arrayAccess(ar, index), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, indexRead, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(index,
env, typesOf, volatileFields, acquireActions, stmts);
    stmts += indexRead;
    acquireActions += extractAcquireActions(indexRead, volatileFields);

    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(ar, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);
    decl = ar@decl;
    potential = addAndLock({Stmt::read(id, ar@decl, dep) | Stmt::read(id, decl,_) <- potential,
dep <- getDataDependencyIds(indexRead)}
                          +{Stmt::read(id, ar@decl, dep) | Stmt::read(id, decl,_) <- potential,
dep<- gatherValues(index)}
                           , acquireActions);

    return <stmts, potential, env, typesOf, acquireActions, exs>;
}

//newArray(Type type, dimensions, Expression init)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:newArray(_, dimensions, init), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    potential = {};
    exs = ();
    for(d <- dimensions) {
        <stmts, potentialD, env, typesOf, acquireActions, exsD> = gatherStmtFromExpressions(d,
env, typesOf, volatileFields, acquireActions, stmts);
        exs = mergeExceptions(exs,exsD);
        potential += potentialD;
        stmts += potentialD;
        acquireActions += extractAcquireActions(potentialD, volatileFields);
    }

    <stmts, potentialI, env, typesOf, acquireActions, exsI> = gatherStmtFromExpressions(init,
env, typesOf, volatileFields, acquireActions, stmts);
    exs = mergeExceptions(exs,exsI);
    stmts += potentialI;
    potential += potentialI;
    acquireActions += extractAcquireActions(potentialI, volatileFields);

    loc con = |java+constructor:///array|;
    potential = addAndLock({create(e@src, con, id) | id <- getDataDependencyIds(potential)}
                          +{create(e@src, con, id) | id <- gatherValues(dimensions)}
                          +{create(e@src, con, id) | id <- gatherValues(init)}
                          , acquireActions);
    if(potential == {})
        potential = addAndLock(create(e@src, con, emptyId), acquireActions);
    return <stmts, potential, env, typesOf, acquireActions, exs>;
}

//newArray(Type type, dimensions)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:newArray(t, dimensions), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
potential = {};
    exs = ();
    for(d <- dimensions) {
        <stmts, potentialD, env, typesOf, acquireActions, exsD> = gatherStmtFromExpressions(d,
env, typesOf, volatileFields, acquireActions, stmts);
        exs = mergeExceptions(exs,exsD);
        potential += potentialD;
        stmts += potentialD;
        acquireActions += extractAcquireActions(potentialD, volatileFields);
    }

    loc con = |java+constructor:///array|;
    potential = addAndLock({create(e@src, con, id) | id <- getDataDependencyIds(potential)}
                          +{create(e@src, con, id) | id <- gatherValues(dimensions)}
                          , acquireActions);
    return <stmts, potential, env, typesOf, acquireActions, exs>;
}

//arrayInitializer(elements)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:arrayInitializer(list[Expression] elements),
map[loc,set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc]
acquireActions, set[Stmt] stmts) {
    potential = {};
    exs = ();
    for(el <- elements) {
        <stmts, potentialC, env, typesOf, acquireActions, exsC> = gatherStmtFromExpressions(el,
env, typesOf, volatileFields, acquireActions, stmts);
        exs = mergeExceptions(exs, exsC);
        potential += potentialC;
        stmts += potentialC;
        acquireActions += extractAcquireActions(potentialC, volatileFields);
    }
    loc con = |java+constructor:///array|;
    potential = addAndLock({create(e@src, con, id) | id <- getDataDependencyIds(potential)}
                          , acquireActions);
    return <stmts, potential, env, typesOf, acquireActions, exs>;
```

```
}

//assignment(Expression lhs, str operator, Expression rhs)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:assignment(lhs,operator,rhs), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    if(operator != "=") {
        rhs = infix(lhs, "+", rhs,[])[@src = e@src][@typ = e@typ];
    }
    return resolveAssignment(lhs, rhs, env, typesOf, volatileFields, acquireActions, stmts, e@src);
}

//cast(Type type, Expression expression)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:cast(_, exp), map[loc,set[loc]] env,
map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts)
    = gatherStmtFromExpressions(exp, env, typesOf, volatileFields, acquireActions, stmts);

//newObject(Expression expr, Type type, args, Declaration class)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:newObject(expr, _, args, _), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts)
    = gatherStmtFromExpressions(Expression::newObject(expr, args)[@decl = e@decl][@typ = e@typ][@src
= e@src], env, typesOf, volatileFields, acquireActions, stmts);

//newObject(Expression expr, Type type, args)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:newObject(Expression expr, _, list[Expression]
args), map[loc,set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields,
rel[loc,loc] acquireActions, set[Stmt] stmts)
    = gatherStmtFromExpressions(Expression::newObject(expr, args)[@decl = e@decl][@typ = e@typ][@src
= e@src], env, typesOf, volatileFields, acquireActions, stmts);

//newObject(Expression expr, args, Declaration class)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:newObject(Expression expr, list[Expression]
args, _), map[loc,set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields,
rel[loc,loc] acquireActions, set[Stmt] stmts)
    = gatherStmtFromExpressions(Expression::newObject(expr, args)[@decl = e@decl][@typ = e@typ][@src
= e@src], env, typesOf, volatileFields, acquireActions, stmts);

//newObject(Expression expr, args)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:newObject(expr, args), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    potential = {};
    exs = ();
    for(arg <- args) {
        <stmts, potential, env, typesOf, acquireActions, exsA> = gatherStmtFromExpressions(arg,
env, typesOf, volatileFields, acquireActions, stmts);
        stmts += potential;
        acquireActions += extractAcquireActions(potential, volatileFields);
        exs = mergeExceptions(exs, exsA);
    }

    loc con = |java+constructor:///|;
    con.path = e@decl.path ?  "";
    potential = addAndLock({create(e@src, con, id) | id <- getDataDependencyIds(potential)}
                          +{create(e@src, con, id) | id <- gatherValues(args)}
                          , acquireActions);
    if(potential == {})
        potential = addAndLock({create(e@src, con, emptyId)}, acquireActions);
    return <stmts, potential, env, typesOf, acquireActions, exs>;
}

//qualifiedName(Expression qualifier, Expression expression)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:qualifiedName(q, exp), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(q, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);
    typesOf = addDeclOfType(typesOf, q@decl, q@typ);
    <stmts, potentialRead, env, typesOf, acquireActions, exsR> = gatherStmtFromExpressions(exp,
env, typesOf, volatileFields, acquireActions, stmts);
    potentialRead += addAndLock({read(addr, var, id) | Stmt::read(addr, var, _) <- potentialRead,
id <- getDataDependencyIds(potential)}, acquireActions);
    potentialRead -= {r | r:read(_,_,emptyId) <- potentialRead};
    return <stmts, potentialRead, env, typesOf, acquireActions, exs>;
}

//conditional(Expresion cond, Expression ifB, Expression elseB)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:conditional(cond,ifB,elseB), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> =
    gatherStmtFromExpressions(cond, env, typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);

    <stmts, potentialIf, envIf, typesIf, acquireActionsIf, exsIf> =
    gatherStmtFromExpressions(ifB, env, typesOf, volatileFields, acquireActions, stmts);
    stmts += potentialIf;
    acquireActionsIf += extractAcquireActions(potentialIf, volatileFields);

    <stmts, potentialElse, envElse, typesElse, acquireActionsElse, exsElse> =
    gatherStmtFromExpressions(elseB, env, typesOf, volatileFields, acquireActions, stmts);
    stmts += potentialElse;
    acquireActionsElse += extractAcquireActions(potentialElse, volatileFields);

    env = updateEnvironment(env,envIf);
    env = merge(env,envElse);
    exs = mergeExceptions(exs,exsIf);
    exs = mergeExceptions(exs,exsElse);
    typesOf = mergeTypesEnvironment(typesOf, typesIf);
    typesOf = mergeTypesEnvironment(typesOf, typesElse);
    return <stmts, potential + potentialIf + potentialElse, env, typesOf, acquireActionsIf +
acquireActionsElse, exs>;
}

//fieldAccess(bool isSuper, Expression expression, str name)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:fieldAccess(_,exp,_), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(exp, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);

    potential = addAndLock({Stmt::read(e@src, e@decl, writtenBy) | writtenBy <- env[e@decl] ?
{emptyId}}
                         + {Stmt::read(e@src, e@decl, getIdFromStmt(dependOn)) | dependOn <-
potential},
                         acquireActions);
    return <stmts, potential, env, typesOf, acquireActions, exs>;
}

//fieldAccess(bool isSuper, str name)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:fieldAccess(isSuper, _), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    if(!isSuper) {
        assert false :  "Field access without expression and not super!";
    }
    superSrc = e@src;
    superSrc.length = 5;
    stmts += addAndLock(
    {Stmt::read(superSrc, |java+class:///|+extractClassName(e@decl)+"/super", dep) |
    dep <- getDependenciesFromType(typesOf, |java+class:///|+extractClassName(e@decl))}, acquireActions);

    potential = addAndLock({Stmt::read(e@src, e@decl, writtenBy) | writtenBy <- env[e@decl] ?
{emptyId}}
                          +{Stmt::read(e@src, e@decl, superSrc)},
                            acquireActions);
    return <stmts, potential, env, typesOf, acquireActions, ()>;
}

//instanceof(Expression leftside, Type rightSide)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:\instanceof(lhs,_), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(lhs, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);
    return <stmts, {}, env, typesOf, acquireActions, exs>;
}

//methodCall(bool isSuper, str name, Expression arguments)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:methodCall(isSuper,name,args), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    thisSrc = e@src;
    thisSrc.offset = thisSrc.offset+1;
    return gatherStmtFromExpressions(methodCall(isSuper, Expression::this()[@src = thisSrc][@typ
= class(|java+class:///|+extractClassName(e@decl),[])], name, args)[@decl = e@decl][@typ = e@typ][@src
= e@src], env, typesOf, volatileFields, acquireActions, stmts);
}
//methodCall(bool isSuper, Expression receiver, str name,  arguments)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:methodCall(isSuper, receiver, name, args),
map[loc,set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc]
acquireActions, set[Stmt] stmts) {
    <stmts, potentialR, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(receiver,
env, typesOf, volatileFields, acquireActions, stmts);
    stmts += potentialR;
    acquireActions += extractAcquireActions(potentialR, volatileFields);

    exs = ();
    potential = {};
    for(arg <- args) {
        <stmts, potentialA, env, typesOf, acquireActions, exsA> = gatherStmtFromExpressions(arg,
env, typesOf, volatileFields, acquireActions, stmts);
        potential += potentialA;
        stmts += addAndLock(potentialA, acquireActions);
        acquireActions += extractAcquireActions(potentialA, volatileFields);
        exs = mergeExceptions(exs,exsA);
    }
    for(arg <- args) {
        <changed, env, typesOf> = propagateChanges(arg, env, typesOf, e@src);
        stmts += addAndLock(changed, acquireActions);
    }
    <changed, env, typesOf> = propagateChanges(receiver, env, typesOf, e@src);
    stmts += addAndLock(changed, acquireActions);

    loc recSrc;
    for(r:read(id, var, _) <- potentialR) {
        recSrc = id;
        if(isField(var) || isParameter(var) || isVariable(var)){
            stmts += addAndLock({change(id, getTypeDeclFromTypeSymbol(receiver@typ), e@src)},
acquireActions);
            env = updateAll(env, getDeclsFromTypeEnv(
            typesOf[getTypeDeclFromTypeSymbol(receiver@typ)]?emptyTypeSensitiveEnvironment()),
id);
            typesOf = update(typesOf, getTypeDeclFromTypeSymbol(receiver@typ), id);
            break;
        }
    }
    <changed, env, typesOf> = propagateChanges(receiver, env, typesOf, e@src);
    stmts += addAndLock(changed, acquireActions);
    for(r:create(id, _, _) <- potentialR) {
        recSrc = id;
    }
    for(r:call(id, _, _) <- potentialR) {
        recSrc = id;
    }
    potential = addAndLock(
    {Stmt::call(e@src, recSrc, e@decl, arg) | arg <- getDataDependencyIds(potential)}
    +{Stmt::call(e@src, recSrc, e@decl, arg) | arg <- gatherValues(args)}
    , acquireActions);

    //if the method call does not have any arguments
    if(potential == {})
        potential = addAndLock({Stmt::call(e@src, recSrc, e@decl, emptyId)}, acquireActions);
    stmts += potential;
    for(ex <- exceptions[e@decl] ?  {}) {
        if(ex in exs) {
            exs[ex] = merge(exs[ex],state(env,typesOf,acquireActions));
        }
        else{
            exs[ex] = state(stmts, env,typesOf,acquireActions);
        }
    }
    return <stmts, potential, env, typesOf, acquireActions, exs>;
}

//variable(str name, int extraDimensions, Expression init)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:variable(_, _, rhs), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(rhs, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);
    if(e@decl in volatileFields) {
        stmts += addAndUnlock(stmts, e@src, e@decl);
    }

    stmts += addAndLock({Stmt::assign(e@src, e@decl, id) | id <- getDataDependencyIds(potential)}
                       +{Stmt::assign(e@src, e@decl, id) | id <- gatherValues(rhs)}
                       , acquireActions);
    env[e@decl] = {e@src};
    typesOf = addDeclOfType(typesOf, e@decl, e@typ);
    return <stmts, {}, env, typesOf, acquireActions, exs>;
}

//bracket(Expression exp);
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:\bracket(exp), map[loc,set[loc]] env,
map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts)
    = gatherStmtFromExpressions(exp, env, typesOf, volatileFields, acquireActions, stmts);
```

```
//this()
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:this(), map[loc,set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts) {
    potential = addAndLock(
    {Stmt::read(e@src, |java+parameter:///| + getTypeDeclFromTypeSymbol(e@typ).path + "/this",
dep) | dep <- getDependenciesFromType(typesOf, getTypeDeclFromTypeSymbol(e@typ)) }, acquireActions);
    if(potential == {}) {
        potential = addAndLock(
        {Stmt::read(e@src, |java+parameter:///| + getTypeDeclFromTypeSymbol(e@typ).path + "/this",
emptyId)}, acquireActions);
    }
    return <stmts, potential, env, typesOf, acquireActions, ()>;
}

//this(Expression exp)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:this(exp), map[loc,set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts) {
    assert false :  "Found this with expression in:  <e>!";
    return <stmts, {}, env, typesOf, acquireActions, ()>;
}

//super()
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:super(), map[loc,set[loc]] env, map[loc,
TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions, set[Stmt]
stmts) {
    assert false :  "Found super in:  <e>!";
    return <stmts, {}, env, typesOf, acquireActions, ()>;
}

//declarationExpression(Declaration d)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Declaration m , Expression e:declarationExpression(d),
map[loc,set[loc]] env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc]
acquireActions, set[Stmt] stmts) {
    exs = ();
    fenv = emptyFlowEnvironment();
    top-down-break visit(d) {
        case Expression exp :  {
            <stmts, _, env, typesOf, acquireActions, exsE> = gatherStmtFromExpressions(exp, env,
typesOf, volatileFields, acquireActions, stmts);
            exs = mergeExceptions(exs, exsE);
        }
    }
    return <stmts, {}, env, typesOf, acquireActions, exs>;
}

//infix(Expression lhs, str operator, Expression rhs, extendedOperands)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]]  gatherStmtFromExpressions(e:infix(lhs, operator, rhs, ext), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    operands = [rhs] + ext;
    <stmts, potential, env, typesOf, acquireActions, exs> = gatherStmtFromExpressions(lhs, env,
typesOf, volatileFields, acquireActions, stmts);
    stmts += potential;
    acquireActions += extractAcquireActions(potential, volatileFields);

    if(operator == "&&" || operator == "||") {
        envOp = env;
        typesOp = typesOf;
        acquireActionsOp = acquireActions;
        for(op <- operands) {
            <stmts, potentialOp, envOp, typesOp, acquireActionsOp, exsOp> = gatherStmtFromExpressions(op,
envOp, typesOp, volatileFields, acquireActionsOp, stmts);
            stmts += potentialOp;
            acquireActions += extractAcquireActions(potentialOp, volatileFields);
            env = merge(env,envOp);
            typesOp = mergeTypesEnvironment(typesOf, typesOp);
            exs = mergeExceptions(exs, exsOp);
            //The expressions are already in stmts, however we need to fill the potential for
dependencies
            potential += potentialOp;
        }
        return <stmts, potential, env, typesOf, acquireActions, exs>;
    }
    else{
        exs = ();
        dependencies = potential;
        for(op <- operands) {
            <stmts, potential, env, typesOf, acquireActions, exsOp> =
            gatherStmtFromExpressions(op, env, typesOf, volatileFields, acquireActions, stmts);
            stmts += potential;
            dependencies += potential;
            acquireActions += extractAcquireActions(potential, volatileFields);
            exs = mergeExceptions(exs,exsOp);
        }
        //the reads are not potential because there are operations done one them that cannot
be statements!
        return <stmts, dependencies, env, typesOf, acquireActions, exs>;
    }
}

//postfix(Expression operand, str operator)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]]  gatherStmtFromExpressions(Expression e:postfix(operand, operator), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    if(operator == "++" || operator == "-") {
        rhs = infix(operand, "+", number("1")[@src = e@src][@typ = e@typ],[])[@src = e@src][@typ
= e@typ];
        return resolveAssignment(operand, rhs, env, typesOf, volatileFields, acquireActions,
stmts, e@src);
    }
    else{
        return gatherStmtFromExpressions(operand, env, typesOf, volatileFields, acquireActions,
stmts);
    }
}

//prefix(str operator, Expression operand)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:prefix(operator, operand), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    if(operator == "++" || operator == "-") {
        rhs = infix(operand, "+", number("1")[@src = e@src][@typ = e@typ],[])[@src = e@src][@typ
= e@typ];
        return resolveAssignment(operand, rhs, env, typesOf, volatileFields, acquireActions,
stmts, e@src);
    }
    else{
        return gatherStmtFromExpressions(operand, env, typesOf, volatileFields, acquireActions,
stmts);
    }
}

//simpleName(str name)
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:simpleName(name), map[loc,set[loc]] env,
map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    potential = addAndLock({Stmt::read(e@src, e@decl, writtenBy) | writtenBy <- env[e@decl] ?
{emptyId}}, acquireActions);
    return <stmts, potential, env, typesOf, acquireActions, ()>;
}

//type(simpleType(_)) representing <Object>.class no check for volatile required
tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e:\type(simpleType(name)), map[loc,set[loc]]
env, map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    loc l = name@decl;
    l.path = name@decl.path + ".class";
    potential = addAndLock({Stmt::read(e@src, l, emptyId)}, acquireActions);
    return <stmts, potential, env, typesOf, acquireActions, ()>;
}

default tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] gatherStmtFromExpressions(Expression e, map[loc,set[loc]] env,
map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts) {
    return <stmts, {}, env, typesOf, acquireActions, ()>;
}

tuple[set[Stmt], set[Stmt], map[loc,set[loc]], map[loc, TypeSensitiveEnvironment], rel[loc,loc],
map[str, State]] resolveAssignment(Expression target, Expression rhs, map[loc,set[loc]] env,
map[loc, TypeSensitiveEnvironment] typesOf, set[loc] volatileFields, rel[loc,loc] acquireActions,
set[Stmt] stmts, loc dep) {
    set[loc] independentValues = gatherValues(rhs);
    <stmts, potentialReads, env, typesOf, acquireActions, exsRhs> = gatherStmtFromExpressions(rhs,
env, typesOf, volatileFields, acquireActions, stmts);
    stmts += potentialReads;
    acquireActions += extractAcquireActions(potentialReads, volatileFields);

    <stmts, potentialWrites, env, typesOf, acquireActions, exsLhs> = gatherStmtFromExpressions(target,
env, typesOf, volatileFields, acquireActions, stmts);

    if(arrayAccess(arr,_) := target) {
        decl = arr@decl;
        <changed, env, typesOf> = propagateChanges(target, env, typesOf, dep);
        stmts += addAndLock(changed
                          + {Stmt::change(id, getTypeDeclFromTypeSymbol(arr@typ), d) | Stmt::read(id,
decl,_)<- potentialWrites, d <- getDataDependencyIds(potentialReads)}
                            + {Stmt::change(id, getTypeDeclFromTypeSymbol(arr@typ), d) | Stmt::read(id,
decl,_)<- potentialWrites, d <- independentValues}
                          , acquireActions);
        env = updateAll(env, getDeclsFromTypeEnv(typesOf[getTypeDeclFromTypeSymbol(arr@typ)]
?  emptyTypeSensitiveEnvironment()), dep);
        typesOf = update(typesOf, getTypeDeclFromTypeSymbol(arr@typ), dep);
        potential = addAndLock({Stmt::read(id, decl, arr@src) | read(id, decl, _) <- potentialWrites},
acquireActions);
        return <stmts, potential, env, typesOf, acquireActions, mergeExceptions(exsLhs, exsRhs)>;
    }

    //get the variable name
    loc var;
    for(w:read(_, name, _) <- potentialWrites) {
        var = name;
    }
    <changed, env, typesOf> = propagateChanges(target, env, typesOf, dep);
    stmts += addAndLock(changed, acquireActions);
    if(var in volatileFields)
        stmts += addAndUnlock(stmts, dep, var);

    stmts += addAndLock({Stmt::assign(dep, var, id) | id <- getDataDependencyIds(potentialReads)}
                      + {Stmt::assign(dep, var, id) | id <- independentValues}
                      , acquireActions);
    env[var] = {dep};
    potential = addAndLock({Stmt::read(id, var, dep) | read(id, var, _) <- potentialWrites},
acquireActions);
    return <stmts, potential, env, typesOf, acquireActions, mergeExceptions(exsLhs, exsRhs)>;

}
```