UNIVERSITY OF TWENTE.

MASTER'S THESIS

Runtime Permission Checking in Concurrent Java Programs

AUTHOR:
Stijn Gijsen

SUPERVISOR:
prof.dr. M. Huisman

COMMITTEE:
dr. C.M. Bockisch
dr. S.C.C. Blom

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

in the

Formal Methods and Tools group,
department of Computer Science,
faculty of Electrical Engineering, Mathematics and Computer Science.

21st August 2015

Abstract

The development of concurrent software is one of the key ways for software developers to benefit from the increasing number of processor cores found in computers and embedded devices. Through multithreading, multiple processors can be used to speed up computations or to improve user experiences.

Developing concurrent programs is more difficult than developing sequential programs because of concurrency bugs such as thread interference and data races, which occur when threads operate on one or more pieces of shared memory concurrently. Due to the non-deterministic order in which multiple threads may be executed, these bugs are often hard to find.

Permission specifications have been introduced to reason about shared memory in concurrent programs. By introducing a concept of permissions, these specifications make explicit which memory locations may be read from or written to by individual threads. A number of static verification solutions have been implemented for verifying programs against permission specifications, but no runtime checking solutions for permission specifications exist, despite the fact that concurrency is also often used in software that is not easily checked statically, such as user interfaces.

In this master's thesis, we will discuss ways to track and check permission specifications in a concurrent Java program at runtime. The specification language we use is the annotation language of VerCors, a static verification tool for permission specifications. We extend VerCors with a prototype for runtime checking that instruments Java source code with permission accounting and permission checks. We will also discuss various approaches for developing a production-ready runtime permission checker.

Acknowledgements

Before delving into the topic of runtime checking concurrent software, I would like to express my appreciation for the people that made my master's project possible and supported me during my studies.

Dr. Marieke Huisman supervised the project and taught me most of the things I know about software verification and checking. I'm grateful for all the feedback she has given me and for the friendly atmosphere during our meetings. Thank you for making time to meet and discuss my progress, despite your busy schedule.

Thank you Dr. Christoph Bockisch for supervising my software engineering specialization during most of my time at the University of Twente and for being the lecturer on some of my favourite courses. Thank you for putting me in contact with Marieke and for staying on as a co-supervisor even after moving on to another university; your alternate point of view has been invaluable, often providing alternative solutions or angles to investigate further.

Dr. Stefan Blom has also been a key figure in enabling my research project. My extension of VerCors would not have been possible without his help as the main developer of VerCors, often being quick to resolve issues that impeded my progress and even prioritizing the implementation of functionality that benefited my project.

I am also grateful to my friends and family for supporting me throughout my studies. My mother's and father's support has been unending, without which my master's studies would undoubtedly not have been possible. To my sister, Merel: your inquiries into my progress have been more important and motivational than you may realize. And to my brother in law Andrew: thank you for your frequent encouragement and your interest in my work.

My friends Roel and Rogier: Thanks for helping me achieve most of my extracurricular goals during my time at the university, for putting things in perspective, and, in the case of Rogier, for accompanying me to so many metal concerts.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem statement . . . 2
1.3 Contribution . . . 2
1.4 Related work . . . 3
1.5 Outline . . . 3

2 Problem domain **5**
2.1 Formal specifications . . . 5
2.2 Statically verifying software behaviour . . . 6
2.2.1 Limitations of static verification . . . 7
2.3 Runtime assertion checking . . . 8
2.3.1 Limitations of runtime checking . . . 8
2.4 Concurrent software . . . 8
2.4.1 Concurrency bugs . . . 9
2.4.2 Verification of concurrent software . . . 11
2.4.3 Runtime checking concurrent behaviour . . . 12

3 Permission specifications **13**
3.1 Separation logic . . . 13
3.2 Fractional permissions . . . 15
3.3 Symbolic permissions . . . 16
3.4 Abstract predicates . . . 17

4 Runtime checking permission specifications **19**
4.1 Tracking and storing permissions . . . 19
4.1.1 Thread-local permission accounting . . . 20
4.1.2 Global, static map . . . 21
4.1.3 Per field . . . 23
4.1.4 A note on garbage collection . . . 24

4.2 Checking permissions . . . 24
4.2.1 Dereferences . . . 25
4.2.2 Assignments . . . 25
4.2.3 Constructors . . . 26
4.2.4 Checking method specifications . . . 26
4.3 Exchanging permissions . . . 27
4.3.1 Thread forking . . . 27
4.3.2 Start and join tokens . . . 29
4.3.3 Thread joining . . . 30
4.3.4 Locks . . . 30
4.4 Locking in permission accounting . . . 31

5 Prototype implementation 33
5.1 Permissions accounting . . . 33
5.2 Code transformation . . . 34
5.2.1 Passes . . . 34
5.2.2 The `DynamicCheckInstrumentation` class . . . 35
5.2.3 The `ForkJoinInstrumentation` class . . . 37
5.2.4 Checking and exchanging permissions . . . 38
5.3 Test case: Shared buffer . . . 39
5.4 Limitations of the prototype . . . 40
5.4.1 Checking on the statement level . . . 40
5.4.2 Visibility of fields . . . 44
5.4.3 Support for partially specified programs . . . 44
5.5 Missing features/checks . . . 45
5.5.1 Resource invariants . . . 45
5.5.2 Abstract predicates . . . 45
5.5.3 Detailed error messages . . . 46

6 Optimizations and future work 47
6.1 Reducing redundant checks . . . 47
6.2 Permissions for array elements . . . 48
6.3 Byte-code transformation and JVMTI . . . 49

7 Conclusion 51

A Test case source code 53

Chapter 1

Introduction

During software development, bugs are often introduced by accident. Hence, a major part of developing software is verifying that the developed program behaves correctly. For sequential programs, there are many ways to do this, including various kinds of testing (e.g. unit tests, acceptance tests, etc.), analysing the program statically (*static analysis* or *static verification*), or inspecting a program while it is running (*runtime assertion checking* or *runtime verification*).

This master's thesis details the results of our research into runtime assertion checking of concurrent software. Concurrency in software can introduce dangerous bugs that can be hard to find. We have researched ways in which concurrency in programs can be checked at runtime in order to detect these kinds of bugs, and have implemented a prototype of such a runtime checker.

This introductory chapter describes the motivation for the research project and provides an outline for the rest of the thesis.

1.1 Motivation

The performance of a computer program can be greatly improved through concurrency, as it is a way for a program to benefit from the multiple processors in a computer system that have become commonplace over the past decade.

As the importance of concurrent software has grown, researchers have investigated ways to ascertain the correctness of a concurrent program's behaviour. This has led to the creation of various static, formal verification solutions for concurrent programs. As part of these developments, specification languages have been introduced that allow software developers to formally define the correct behaviour of a concurrent program. These specification languages often use a concept of permissions to guard a program's memory from unsafe (i.e. buggy) operations.

Programs can be checked against these permission by automated tools to prove that a program behaves correctly. While research led to the development of multiple static verification tools (such as VerCors [1] and Chalice [12]), no runtime checking tools for permission specifications have been introduced. Some classes of software, such as programs with large state spaces, are not easily verified statically or within a practical amount of time. Developing a runtime checking solution for permission specifications may allow these kinds of programs to be verified at runtime rather than statically.

1.2 Problem statement

Permission specifications are already being used for static verification. The research goal is to develop a runtime checking solution for concurrent Java programs, which can check whether programs behave correctly according to their permission specifications. If possible, the runtime checker should use the same specification languages as those used by existing static checking solutions, to enable the re-use of specifications, and to build on previous work that proved the usefulness and correctness of these specification languages. Research must be done to determine ways in which permissions may be tracked and checked at runtime.

1.3 Contribution

In this thesis, we present:

- our evaluation of ways in which permissions may be tracked and checked at runtime within a concurrent Java program;
- implementation details of our prototype implementation of a tool that instruments Java source code with runtime permission checking code;
- ways in which to implement features missing from the prototype;
- ideas for implementing a production-ready runtime permission checker for Java.

1.4 Related work

Kandziora [10] introduces a way for the OpenJML runtime assertion checker to be free of interference, by performing the runtime checks on a copy-on-write snapshot of memory that cannot be overwritten by other threads. While this not let OpenJML check the correctness of concurrent behaviour, it does make checking functional specifications safe in a concurrent environment.

1.5 Outline

Chapter 2 describes some of the key aspects of concurrent software verification, providing short introductions to concepts such as specifications, static verification using formal methods, runtime checking, and the problems of concurrent programming.

In Chapter 3, we provide a quick introduction to the use of permissions in specifications to describe correct behaviour of concurrent software, i.e. safe memory manipulations in a concurrent context.

Chapter 4 describes the challenges of checking permission specifications in programs at runtime, and offers possible solutions to these challenges.

Implementation details for our runtime checker prototype can be found in Chapter 5, along with possible implementation approaches for some of the features that are missing from the prototype.

Finally, Chapter 6 describes a number of possible optimizations and avenues for future research and implementation work, including some ways to create a production-ready runtime checker for permissions.

Chapter 2

Problem domain

This chapter details some of the key concepts of the problem domain of checking the concurrent behaviour of software.

2.1 Formal specifications

A program's *specification* is a description of the required behaviour of a program. A specification can be a document meant for the developers and other stakeholders, or it can be a formal, declarative description written in a formal specification language, to be processed by automated tools. For instance, a specification may describe the properties of a program that should hold before and after a particular piece of code (e.g. a function or statement) is executed. If these *pre-* or *postconditions* do not hold in the final program, the program behaves in an unspecified and unexpected way. In other words: a bug occurs.

An example of such a specification using pre- and postconditions written in the Java Modelling Language (JML) [11] can be seen in Listing 2.1. The specification declares the pre- and postconditions for a Java method that calculates the average value of the integers in an array. The example specifies the requirement that the length of the input array must be greater than zero (in order to prevent division by zero errors) and gives the guarantee that, if the precondition has been met, the method will return the average of the integers in the array. JML specifications are embedded within the source code of a Java program, using special annotation comments starting with `//@` or `/*@`.

Listing 2.1: Example of a JML specification for a method that calculates the average value of an array of integers.

```
//@requires nums.length > 0
/*@ensures \result ==
    (\sum int i; 0 <= i && i < nums.length; nums[i]) /
    (float) nums.length; */
static float average(int nums[]) {
  // ...
}
```

2.2 Statically verifying software behaviour

A program can be analysed statically (i.e. without running it) in order to determine whether certain properties hold for it. For instance, compilers perform static analysis in order to check type safety (in case of statically typed languages), to check for common bugs such as the use of uninitialized variables, or to warn about common programmer errors such as using the assignment operator (`=`) instead of the equals operator (`==`) in an `if`-condition.

Static verification is the process of statically analysing a program and determining whether it is correct, according to its given specification. Static verification is often done using formal methods, for instance by generating a mathematical model of the program (e.g. a state machine) and checking this model against the specification. A *sound* formal verification method explores the entire state space of the model, thereby verifying that the program adheres to its specification in all scenarios. Thus, a sound technique can prove the absence of bugs. *Unsound* formal verification methods only explore a subset of the possible executions of the program, trading the conclusiveness of its findings for finishing more quickly.

An example of a formal verification method using model checking is to translate the specification into properties that must hold for the entire model. These properties can then be checked for the model using a model checker such as NuSMV [3]. If the checker finds properties that do not hold in a particular state of the model, this is indicative of unspecified behaviour in the program. The model checker generates a counter-example to show which inputs cause the properties to be violated at a particular state in the model. Static analysis tools such as Goanna [5] can translate programs written in high-level programming languages into a formal model to be checked by a model checker and can map generated counter-examples back to the relevant variable values and statements in the concrete program, to allow the developer to inspect and correct the error.

2.2.1 Limitations of static verification

A number of issues can make static verification methods impractical for some kinds of software.

Because sound formal verification methods explore the entire state space of a program, they do not scale well with increased complexity of the program (for instance through nested loops, recursion, or non-determinism) as this causes a state space explosion, greatly impacting the time required to come to a formal proof of the program's correctness. For programs with large state spaces, the time required to come to a conclusion about the program may be impractical for software development. The increased complexity may also make it harder (i.e. more time-intensive) to generate a correct formal model for the program.

Another problem with static verification is a direct result of the halting problem, famously proven by Alan Turing to be undecidable over Turing machines [16]. For instance, an infinite loop or recursion may prevent a program from ever terminating, but the undecidability of the halting problem means that it is impossible to predict for all loops whether or not they may loop infinitely. It may therefore also be impossible to determine whether the static verification tool will ever finish exploring the state space, as the state space may be infinite.

To circumvent the implications of this undecidability, unsound verification techniques may make compromises by approximating the program's behaviour. This can lead to false positives (detection of bugs that are not present in the concrete program) or false negatives (bugs in the concrete program that go undetected by the static analysis method). Another solution is the extension of the program's specification with guarantees that a loop will terminate, such as loop invariants [6], which are properties that must hold before and after each iteration of the loop. Writing these annotations is not always trivial, and the undecidable nature of the halting problem means that these annotations cannot always be found automatically.

In practice, formal verification methods are generally only used for critical software systems and for hardware designs, where bugs have a major economical impact or may endanger lives. For non-critical systems that may crash and recover without major consequences (such as user interfaces), less conclusive alternatives that scale better, such as testing or *runtime checking*, are often more practical.

2.3 Runtime assertion checking

Instead of- statically analysing a program, it is also possible to check assertions that a program adheres to its specification whilst the program is running. Ways to do this include inserting assertion checking code into programs that validates the state of the program against the specification at given moments in time (such as before and after functions), or by validating the program using external tools that inspect its state through a debugging interface.

Runtime checking can also be used for monitoring, by checking properties of the program in order to warn for potential problems before they cause the program to crash or to log the properties for later inspection by the developers.

Runtime checking frameworks include OpenJML [4] for Java and CodeContracts [13] for .NET programs.

2.3.1 Limitations of runtime checking

Because checks are performed on the actual runtime state of the program, runtime checking can only verify the correctness of the current execution of the program. This makes runtime checking considerably faster than static verification, making it viable for complex software which cannot easily be verified statically, but it also means that runtime checking cannot guarantee the absence of bugs outside of the executed path. Unlike static verification, runtime checking also incurs a runtime overhead, since checking the properties requires CPU time and may require extra memory space.

2.4 Concurrent software

Traditionally, computer programs are *sequential*, consisting of a sequence of instructions that are executed by the CPU. This sequence of instructions is executed in order, in a so-called *thread* of execution.

A *concurrent* program, also called a multithreaded program, is a program in which multiple parts of the program execute simultaneously, with each part being executed in a separate thread. If a system has multiple processors, as is now common, multiple threads may be executed simultaneously, in *parallel*. However, each processor can only execute a single thread at once, and if there are more running threads on a system than there are processors, threads must wait for a processor to become available. The scheduler, which is typically a component of the operating system, manages the execution of threads, occasionally pausing an executing thread so that a waiting thread may be executed. In practice, this happens many times per second, as modern desktop computer systems typically have hundreds of threads executing at once, with only two to six processors to execute them on. Schedulers also make it possible for multiple threads to run on systems with only a single processor, giving the illusion of parallel execution.

Concurrency can be used to speed up time-intensive algorithms and computations, for instance by partitioning the work load and spreading it across multiple processors. Concurrency can also be used to improve the user experience of a program, for instance by running the user interface and time-intensive operations (such as complex computations or blocking I/O operations) in separate threads, allowing the UI to stay responsive to the user's actions.

2.4.1 Concurrency bugs

In a sequential program, the order in which instructions are executed is predetermined, as the instructions can only be executed in the order in which they appear in the program. When such a program is executed concurrently using multiple threads, this is no longer the case: As threads are paused and activated by the scheduler, or as threads are executed in parallel, the instructions in the program become interleaved in a way that cannot be predicted during development. When these interleaved threads access and manipulate the same resources (e.g. data in memory) without consideration of each other, *thread interfere* may occur: concurrently running threads interfering with the others' execution. This may lead to unexpected behaviour of the program.

Threads in a multithreaded program have individual stacks, but heap memory is typically accessible to all threads. This enables communication between threads, but can also lead to unexpected output or behaviour when memory is shared between threads carelessly. The focus of this thesis is the detection of potential *data races*, which are a kind of bug that occurs when threads read or write memory that is also in use by other threads. Data races can cause some thread to affect the outcome of another thread's computations, which may in turn cause that thread to take branches they otherwise would not have.

An example of a program that may have data races is shown in Listing 2.2. The simple `Counter` class counts the number of times the *increase* method has been called.

The byte-code for the `increase` method as generated by the OpenJDK compiler

Listing 2.2: A Java program that may contain data races

```
class Counter {
  int count = 0;

  void increase() {
    this.count += 1;
  }
}
```

Listing 2.3: Bytecode for the `increase` method of Listing 2.2

```
aload_0      // Push the 'this' reference to the stack
dup          // Duplicate the head of the stack
getfield #2  // Push 'this.count' value onto the stack
iconst_1     // Push integer constant 1 onto the stack
iadd         // Add the two values together on the stack
putfield #2  // Push the top of the stack to 'this.count'
return       // Return from the method
```

is shown in Listing 2.3. Note that Java programs store objects in heap memory. Simply put, the bytecode reads the value of the `count` field from the heap onto the stack, pushes the constant value 1 onto the stack, adds these two values together on the stack, and writes the new value back from the top of the stack into the `count` field in heap memory.

If two threads happen to execute the increase method concurrently, the following inter-leaving of the (simplified) instructions might occur:

Thread A	Thread B	count
read count (0) to stack A		0
increment stack A value by 1		0
	read count (0) to stack B	0
	increment stack B value by 1	0
	write stack B value (1) to count	1
write stack A value (1) to count		1

The expected value of `count` after calling `increment` twice is 2, but due to the inter-leaving of the threads, the value is incremented from 0 to 1 twice, thread A overwriting the value written by thread B. Note that this inter-leaving of the threads is serendipitous and any other inter-leaving (or no inter-leaving at all) might occur when the program is ran another time.

Data races can be avoided using locks, which threads must acquire before executing a particular sequence of instructions. Threads that attempt to acquire a lock that is already in use will be forced by the scheduler to wait for the lock to become available. In most programming languages it is up to the software developers to explicitly acquire and release these locks correctly and consistently in order to protect heap memory. However, avoiding data races using locks may itself cause another type of concurrency bug if care is not taken when using locks: *deadlocks* occur when two or more interdependent threads must wait on each other to finish (i.e. release their locks), causing them to wait indefinitely.

Like all bugs, concurrency bugs can have disastrous results. Unfortunately, concurrency bugs can often be hard to detect, as they are the result of the non-deterministic order in which threads are interleaved, meaning the bug may not occur every time the program runs, even with the same input. To complicate matters, attempting to investigate the bugs by attaching a debugger or adding extra debugging code to the program may affect the inter-leaving of the threads, which may cause the bugs to stop occurring.

Testing for concurrency bugs can also be difficult and time intensive, as it may require the orchestration of multiple threads, and it may be required to test many possible thread inter-leavings in order to be certain that the program is correct.

Because concurrency bugs occur in code that may be correct in a sequential context, and because of their elusive nature, writing bug-free concurrent software is a difficult task.

2.4.2 Verification of concurrent software

Specification languages for correct concurrent behaviour of programs have been introduced, including languages that use permissions to guard the memory of a program against dangerous modifications, similar to the way permissions in databases and locks in file systems prevent the modification of data that is already in use by another process. Threads must acquire these permissions in order to read or write the memory locations. Note that, unlike the database and file systems examples, the permissions for programs only exist at the specification level: the concrete programs do not explicitly manipulate permissions through any API; In fact the concrete programs are unmodified and are not aware of the permission concept. They are only used to specify which behaviour is safe and intended, for the purposes of (static) verification.

Verification techniques for concurrent software are relatively new and the subject of active research. Most of this research focuses on static verification, because it is well-suited to explore the many possible inter-leavings of multiple threads. Runtime checking of concurrent software is much less common. For instance, OpenJML currently does not support functional checking of concurrent programs, not to mention checking specifications of their concurrent behaviour.

Permission specifications, particularly those of the VerCors project, are described in more detail in Chapter 3.

2.4.3 Runtime checking concurrent behaviour

Despite the lack of runtime checkers for concurrent behaviour, there are cases for which such a a runtime checking solution might prove useful. User interfaces can be hard to verify statically, because user interaction introduces a lot of non-determinism. Because concurrency is often used to improve the responsiveness of user interfaces, runtime checking may prove to be a good alternative for these cases that are hard to verify statically.

Runtime checking of concurrent software is particularly tricky because the runtime checker itself must avoid data races or other forms of interference from the multi-threaded environment.

Chapter 3

Permission specifications

Permission specifications can be used to formally specify the expected concurrent behaviour of a program. By guarding read and write access to memory locations with permissions, data races and incorrect usage of locks can be detected.

An example of a specification method using permissions is *Separation Logic with Fractional Permissions* [2]. Separation Logic was first introduced to reason about sequential programs with pointers into memory, but it was later found to be useful for concurrent software as well [1].

Concurrency bugs like interference are caused when a thread modifies data while it is concurrently being processed (read or written) by other threads. Permissions are used to protect against these scenarios, by guarding threads' memory accesses, guaranteeing that a thread only modifies data in memory when that thread has exclusive access to it.

3.1 Separation logic

Separation logic [14] is an extension of Hoare logic [7] that allows one to reason about shared memory and pointers thereto. When a program has multiple pointers to a single region of memory, and the memory is changed through one of these pointers (for instance, the data is moved elsewhere), other pointers may become invalidated, now unexpectedly pointing to whatever data now occupies that region of memory, or pointing at unallocated memory.

An example of this can be seen in Listing 3.1, which is a fragment of a program written in C. The fragment shows a function for deleting an item from a linked list and releasing its memory. The `free_item` function may be problematic if the program still has pointers to the deleted item elsewhere, as these pointers will now point to unallocated memory, or to memory that has since been re-allocated and filled with arbitrary data.

Listing 3.1: Example of problematic pointer program in C.

```
struct list_item {
    char name[20];
    int size;
    struct list_item *next;
    struct list_item *previous;
}

void free_item(struct list_item *item) {
    if (item->previous != null)
        item->previous->next = item->next;
    if (item->next != null)
        item->next->previous = item->previous;
    free(item);
}
```

Separation logic was introduced to define predicates for this kind of situation, particularly to prevent the modification of memory that has multiple pointers to it. The key innovation of separation logic is the introduction of a binary *separating conjunction* operator *, which evaluates to true if and only if the left and right hand sides of the operator are valid for disjoint parts of the program's heap. In other words, the formula $\phi * \psi$ only resolves to true if the sub-expression ϕ only references memory locations not referenced in ψ, and vice versa.

These issues with pointers are similar to the data race problems in concurrent software. However, separation logic itself is too restrictive to be used to reason about concurrent programs as it does not allow for shared read access through multiple pointers. Data cannot be shared between threads in a concurrent program, even though this is safe under the restriction that the data is not modified.

Separation logic has been extended with a notion of permissions to allow shared read access, by requiring threads to have read or write permissions for a memory region in order to access it. These permissions have the following properties:

1. Memory regions guarded by permissions may not overlap, similar to the notion of disjoint heaps in separation logic.

2. At most one thread may have a write permission for a memory region.

3. Whenever a thread has a read permission for a memory region, all other threads can at most have a read permission for it as well.

Using permissions that adhere to these rules, properties can be specified that prevent data races. The third rule guarantees that multiple threads never manipulate the same memory region simultaneously, and the second and third rules together guarantee that a thread that holds a write permission has exclusive access to the relevant memory region.

Permissions can be passed between threads when threads synchronise: on fork, on join, and through locks. A thread holding a permission may either pass the permission to another thread, thereby giving up its own permission to access the guarded memory region entirely, or it may share a read permission with the other thread, giving up write-access to the memory region if it had it. When a thread (re)acquires all the read permissions for a memory region, it has gained exclusive access to the region, causing it to (re)gain write access to it.

3.2 Fractional permissions

Separation logic with *fractional permissions* [2] introduces a variation of the permission specification that represents permissions as fractions to make it possible to determine whether a thread has gained exclusive (i.e. write) access to a given memory region without knowing about the permissions held by other threads.

With fractional permissions, permissions are not copied between two threads when they are shared, but split in half and distributed evenly. A whole permission (i.e. $\frac{1}{1}$ or just 1) represents a write permission. Any smaller fraction of a permission represents a read permission. When a write permission is converted into a read permission and shared with another thread, the permission is split into two $\frac{1}{2}$ fractions. A fractional permission can be split further arbitrarily (e.g. splitting a $\frac{1}{2}$ permission into two $\frac{1}{4}$ permissions), so that a read permission can be shared with arbitrarily many threads. When a thread has gained multiple fractions of the same permission, it can add the fractions together. If fractions add up to 1, it is guaranteed that the thread is the exclusive owner of the permission, and it follows that the thread has (re)gained write access to the permission's region of memory.

Permission specifications describe the permissions that are required from, and given to, threads when executing a particular fragment of code, and the permissions required and returned by threads when they are forked and joined, respectively. An example of such a specification (using the annotation syntax of the VerCors tool) is given in listing 3.2. The specification for the `increase` method states that

Listing 3.2: An example of a permission specification for a method

```
class Counter {
  int count = 0;

  //@ requires Perm(this.count, 1);
  //@ ensures Perm(this.count, 1);
  void increase() {
    this.count += 1;
  }
}
```

threads executing this method must have write access to the `count` field of the `Counter` object.

3.3 Symbolic permissions

Alternative permission systems for concurrent software exist, including the use of symbolic permissions [8] rather than concrete (e.g. fractional) permissions. Like fractional permissions, symbolic permissions indicate whether threads have read or write access to a particular memory location. However, a symbolic permission also tracks which threads the permission was passed from. These originators of the permission have the privilege to demand for the permission to be returned to them, and may pass this privilege to other threads. In other words, recipients of a permission are indebted to the permission's originator, and the originator may pass this debt to (or share it with) other threads.

The symbolic permissions of [8] are modelled as simple lists of threads that have access to a memory region and which threads they owe this access to. For instance, the list $[A, [B, A]]$ represents a permission for a field f shared by thread A with thread B.

This symbolic approach to permissions can be processed more efficiently (as there are no fractions or rational numbers involved in splitting and regaining permissions) and allows some permission transfer scenarios to be specified in a more intuitive way. Tracking the history of originators of a permission (i.e. the chain of threads the permission was passed from) also allows a verification tool to determine which other threads have the right to join a thread and gain its permissions. With fractional permissions, this right must be represented in some other way, for

Listing 3.3: An example of an abstract predicate to encapsulate permissions to private fields

```
class Person {
  private String name;
  private int age;

  /*@ resource personalia(frac f) =
        Perm(this.name, f) ** Perm(this.age, f); */

  //@ requires personalia(1);
  //@ ensures personalia(1);
  public void update(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

instance as a separate *join token* permission [1].

3.4 Abstract predicates

In the annotation language of VerCors, *abstract predicates* [15] (called *resources* in VerCors [1]) may be defined, grouping permissions together under a single name.

Using abstract predicates, specifications may be simplified, for instance when multiple permissions are commonly used together. They can also be used to encapsulate permissions for internal (i.e. private and protected) fields of a class, so that its API does not expose the class' internal workings, which is a key concept of object oriented design. Predicates can also be parametrized. An example of such a parametrized predicate used for encapsulation can be seen in Listing 3.3. The `personalia` predicate can be used to represent read or write permissions to the private `name` and `age` fields of the `Person` class, depending on the value of the `f` parameter. Note that the VerCors annotation language uses `**` as the separating conjunction operator, to differentiate it from the multiplication operator of Java.

Abstract predicates can also be used recursively, allowing them to be used to specify properties of recursive data structures such as linked lists.

Finally, they may also be declared without a definition (i.e. without a body). This makes them well-suited for representing token permissions such as the aforementioned join token.

Chapter 4

Runtime checking permission specifications

We wish to develop a runtime checking solution for permissions in concurrent Java programs. In order to check permissions, a number of tasks must be performed:

1. Permissions must be tracked throughout the lifetime of the program.
2. Permissions must be checked whenever heap memory is read from or written to.
3. Permissions must be exchanged at synchronisation points.

These tasks can be performed in a number of ways. We have researched solutions to these problems that may be integrated into the programs that we wish to check, as our prototype (discussed in Chapter 5) performs runtime checking by modifying the underlying program. The solutions to these three tasks are discussed in this chapter.

4.1 Tracking and storing permissions

In order to check permissions at runtime, *permission accounting* must be done, keeping track of all permissions in the program at all times. There are numerous ways to achieve this.

The permission accounting solution for a Java program needs the following capabilities:

- Permissions must guard access to static and dynamic fields of objects of primitive and composite data types.
- Permissions must be exchangeable between threads.
- The performance impact of the runtime checker should be minimized where possible, in particular:
  - Checking read or write access for the currently running thread is the most common operation and should be optimized.
  - Locks should be avoided when possible. Global locks in particular should be avoided as they may defeat the point of multithreading.

Three alternatives for permission accounting have been considered, and are discussed individually below.

4.1.1 Thread-local permission accounting

With thread-local permission accounting, all permission tracking and storage is done in memory that is only accessible by the currently running thread. As permissions will be looked up for fields, the accounting can be a simple *field* $\mapsto$ *fraction* mapping. In Java:

```
ThreadLocal<Map<Object, Fraction>> permissions;
```

This approach is a very literal implementation of fractional permissions, wherein threads have no information about the permissions of other threads. No locks are required for checking permissions, as only the currently running thread can ever access its permission accounting.

Checking for read access for a thread to field `f` is a matter of determining whether the thread's permission accounting contains any permission for the field, i.e.

```
permissions.get().get(f) != null;
```

To look up whether the thread has write access to field `f` we must check that a permission exists for the field, and that the permission equals $\frac{1}{1}$:

```
permissions.get().get(f) != null &&
    permissions.get().get(f) == Fraction.ONE;
```

After optimization (i.e. caching the result of the first `get(f)` when checking write access) these are both lookups in constant time in the best case (and typical) scenario, and linear time in the worst case scenario (depending on the hash function used).

This approach has two major downsides:

- Exchanging permissions between threads is not trivial, because a thread cannot directly access the permission accounting of some other thread in order to give it new permissions.
- Permissions for fields of primitive types are also problematic, because primitives are passed by value in Java and cannot be referenced.

The second problem may be solved in a number of ways. It is possible to use the Java reflection API to retrieve `java.lang.reflect.Field` instances for primitive fields, which might then be used instead of the object reference. However, this approach incurs the performance overhead of reflection and adds complexity to checking permissions.

Another solution to the second problem is to use an $(\mathit{objectref}, \mathit{class}, \mathit{fieldname})$ triple as the key for the `Map`. The *class* element could be a reference to a Java `Class` object or the class' fully-qualified name as a string, and is required in order to disambiguate the permission accounting in cases of inheritance, because superclasses and their subclasses may have distinct fields with the same name. As well as requiring more memory than the other approaches, the addition of the *class* element also increases the complexity of looking up permissions.

4.1.2 Global, static map

Giving threads direct access to other threads' permission accounting is an easy way to solve the problem of exchanging permissions. When threads are created, the creating thread can set up the permission accounting for the newly created thread before it is started.

This may be implemented as a globally accessible static class mapping fields to the threads that have a permission for them:

$$(\mathit{objectref}, \mathit{fieldref}) \mapsto (\mathit{thread} \mapsto \mathit{fraction})$$

or alternatively:

$$\mathit{objectref} \mapsto (\mathit{fieldref} \mapsto (\mathit{thread} \mapsto \mathit{fraction}))$$

The mapping could also be flipped, mapping threads to the fields they have access to:

$$\mathit{thread} \mapsto (\mathit{objectref} \mapsto (\mathit{fieldref} \mapsto \mathit{fraction}))$$

This is effectively equivalent to the thread-local approach, but now globally accessible. The former approach opts to keep all permissions for a given field together. In Java, these approaches may be implemented as follows (the alternative former approach is shown):

```
class P {
    public static Map<Object,
                    Map<Object,
                      Map<Thread, Fraction>>> perms;
    // ...
}
```

Looking up read access to field `f` of object `o` for thread `t` is as simple as a lookup for the thread in the map for field `f`, i.e.:

```
P.perms.get(o) != null &&
P.perms.get(o).get(f) != null &&
P.perms.get(o).get(f).get(t) != null;
```

To look up whether thread `t` has write access, we must also check that its permission equals $\frac{1}{1}$:

```
P.perms.get(o) != null &&
P.perms.get(o).get(f) != null &&
P.perms.get(o).get(f).get(t) != null &&
P.perms.get(o).get(f).get(t).equals(Fraction.ONE);
```

It's worth noting that this approach is closer to symbolic permissions than fractional permissions as all permissions for each thread are known and accessible to all other threads, and indeed there is no reason not to use symbolic permissions here. Since we know about the permissions of all the other threads, we do not need fractional permissions and can improve the space efficiency by only tracking which threads have *any* access to a field:

```
public static Map<Object,
                Map<Object, Set<Thread>>> perms;
```

This changes the read access lookup as follows:

```
P.perms.get(o) != null &&
P.perms.get(o).get(f) != null &&
P.perms.get(o).get(f).contains(t);
```

For write access lookup we must also check whether thread `t` has exclusive access:

```
P.perms.get(o) != null &&
P.perms.get(o).get(f) != null &&
P.perms.get(o).get(f).contains(t) &&
P.perms.get(o).get(f).size() == 1;
```

Note that we only save storage space with this optimization, as we must still do three to four lookups, plus a comparison for write access checks, although comparing integers is likely to be faster than comparing fractions. The typical implementation of the `Set` interface in Java, `HashSet`, is backed by a `HashMap`, and so the approach with fractions has the same complexity as the symbolic approach.

Like the thread-local approach, the global map approach cannot easily store permissions for primitive fields.

4.1.3 Per field

A third alternative takes the sets of threads that have access to a given field from the global static map and embeds them in field's objects using *ghost variables*, i.e. variables that are only visible to the runtime checker. Ghost variables of type `Map<Thread, Fraction>` are added for each field. As with the global map approach, since threads have access to the permissions for all other threads, symbolic permissions are sufficient, reducing the `Map` to a `Set<Thread>`.

An example of this approach can be seen below. For the `num` field of the `Counter` class below, the permissions for it are stored in the generated `num_permissions` field, using the `ghost` annotation syntax of JML:

```
class Counter {
    int num;
    //@ ghost Set<Thread> num_permissions;
}
```

For static fields, static ghost variables should be generated.

Similarly to the global static map approach, determining read access for thread `t` merely checks whether it has any access at all:

```
num_permissions.contains(t);
```

Checking write access also requires the assertion that the thread has *exclusive* access:

```
num_permissions.contains(t) &&
    num_permissions.size() == 1;
```

As can be seen from the example `Counter` class, the per-field permission accounting approach neatly supports storing permissions for fields of primitive types. This approach encapsulates permissions within the object they belong to, which causes the permission accounting to be neatly garbage collected along with the object when it goes out of scope.

The main downside to this approach is its invasiveness, as it changes the definition of the classes. Depending on the implementation of ghost variables, this may cause the permission accounting to be exposed if the program uses reflection or serialization of objects.

4.1.4 A note on garbage collection

The examples in this section use strong references to objects for the sake of brevity. It should be noted that the permission accounting data structures should avoid keeping strong references to objects (including threads), as this would prevent the objects from being garbage collected even after the objects and threads have gone out of scope in the concrete program. In Java, the `WeakReference`, `WeakHashMap` and `WeakHashSet` classes can be used for this purpose.

4.2 Checking permissions

In this section, we will explore which permissions must be checked in which scenario. The way to check a read or write permission depends on the chosen permission accounting and has been discussed in Section 4.1. In this section, we abstract theses implementation details away into commented out pseudo-code.

In the examples below, permission checks occur on the lines above each statement. It should be noted that the checks should actually happen within the expressions, at the very moment the fields are dereferenced. This is important for expressions such as `(bob.age < 65 || bob.spouse.age < 65)` as the `bob.spouse.age` field is never accessed if the left hand side of the logical-or operator evaluates to `true`, and so the program is correct if the thread does not have a read permission for it. Section 5.4.1 describes ways to implement permission checks within expressions.

4.2.1 Dereferences

Because data races can only occur with memory shared between threads, we only have to check permissions whenever references to fields of objects are dereferenced (i.e. assigned to or read from), including fields of the current object (i.e. `this`). Stack variables (e.g. variables local to a method) do not need to be checked.

We must check for read access whenever a field is dereferenced but not assigned to or otherwise manipulated. For example:

```
// check read access for bob.age
int i = bob.age;

// check read access for this.message
System.out.println(this.message);
```

When field references are nested, we must check for access for each of the nested references:

```
// check read access for bob.spouse
// check read access for bob.spouse.age
int i = bob.spouse.age;
```

4.2.2 Assignments

Whenever a reference is on the left-hand side of an assignment statement or expression, we must check for write access.

```
// check write access for this.value
this.value = 27;

// check write access for o.value
// check write access for this.value
this.value = o.value = 1988;
```

These same checks must be performed for the compound assignment operators: `+=`, `-=`, `/=`, `*=`, `%=`, `<<=`, etc.

Apart from the assignment operators, the unary pre- and postincrement operators `++` and `--` also modify their operands, causing them to need write permission:

```
// check write access for bob.age;
++bob.age;
// check read access for bob.spouse;
// check write access for bob.spouse.age;
++bob.spouse.age;
```

Read access to `bob.spouse` is required to prevent interference in the event that another thread updates who Bob's spouse is at the same time that the spouse's age is being incremented.

4.2.3 Constructors

During object construction, some permission checks are unnecessary, since there is no way for any other threads to have access to these fields, and permission accounting may not have been initialized yet:

- Write permission does not have to be checked for a field as it is being initialized.
- Read and write permissions do not have to be checked for fields that occur in initialization blocks or constructors, if the fields belong to the object under construction.

Permissions for read or write operations on fields of other objects must still be checked if they occur in an initialization block or constructor.

These cases are demonstrated in Listing 4.1.

4.2.4 Checking method specifications

Specifications for methods should be checked by the caller before and after method calls, in order to check whether it is allowed to call the method, and to check whether the permissions are in line with the specification after the method has returned.

Inside a method's body, permissions should also be checked in case the method was called from code that was not instrumented, for instance through a callback from an external library.

Listing 4.1: Examples of permission checks during object construction

```
class Singleton {
  public static String name;
}

class Widget {
  // no checks required
  int value = 12;
  // check read access for Singleton.name
  String name = Singleton.name;

  public Widget(int val, Widget w) {
    // no checks required
    this.value = val;

    // check write access for w.value
    w.value = val * 2;
  }
}
```

4.3 Exchanging permissions

Permissions can only be exchanged on synchronization points:

- When threads are forked
- When threads are joined.
- When locks are acquired and released.

4.3.1 Thread forking

When a new thread is started (*forked*) by a another thread (referred to as the *source* thread in this text), permissions from the source thread can be given to the forked thread. A special abstract predicate `preFork` in the specification of the forked thread defines which permissions the thread requires.

Via the `preFork` resource, threads may request a complete permission, in which case the thread requires write access, or any fraction of a permission, meaning the thread requires only read access.

When statically verifying a program's correctness it is possible to lazily evaluate whether a permission should have been split or passed when a thread was forked. With runtime checking, we must split or pass the permission at the time the fork is performed. This poses a problem when the specification is ambiguous. When a thread explicitly requests a write permission (i.e. `Perm(this.num, 1)`), it is clear that the forked thread requires write permission, and we must first check whether the source thread currently has a write permission, before passing this permission to the forked thread entirely. However, when a forked thread requests any fraction of a permission, it becomes ambiguous whether to pass or split the permission:

```
class Worker {
  int num;
  //@ resource preFork(frac f) = Perm(this.num, f);

  // ...
}
```

While it is immediately clear that the forked thread does not require write access, it is unclear whether the source thread should hold on to a fraction of its permission or pass its permission to the forked thread entirely. Passing the permission from the source thread to the forked thread entirely would satisfy the specification, even if the source thread has write access. However, it is not clear from the specification whether the source thread requires a fraction of the permission after the fork.

The following specification is not ambiguous, however:

```
class Worker {
  int num;
  //@ resource preFork(frac f) = Perm(this.num, f / 2);

  // ...
}
```

For this specification, regardless of whether the value of `f` at runtime is $\frac{1}{1}$ or a smaller fraction, the forked thread requests half of the source thread's permission, meaning that the source thread will always keep a fraction of the permission for itself.

There are two solutions to this problem:

- Extending the specification language to allow for disambiguation in these scenarios;

- Choosing a default behaviour for the ambiguous cases. Since the ambiguous requirements can be met with read access, the safest default behaviour is to always split the permission, although this may not always be the intended behaviour. Unfortunately, the specification language does not offer an easy way to explicitly request passing the permission (with the exception of requesting write permissions).

4.3.2 Start and join tokens

Though their internal representation depends on the permission accounting approach, thread objects can be seen to have two special permissions that are not associated with any fields: the start and join token permissions [1, 8].

A start token is created when a thread is instantiated, and this permission is given to the calling thread of the thread's constructor. Ownership of the start token is a prerequisite for calling `Thread.start()`. Because a thread can only be started once, the start token may not be split, but it may be passed to other threads.

When a thread is started, the source thread loses its start token and gets the join token. In the same way that the start token is required to call `Thread.start()`, the join token is required to call `Thread.join()`. However, since threads may be joined multiple times and by multiple threads, the join token may be split and passed to other threads.

The following fragment demonstrates the steps to perform surrounding thread creation and forking:

```
WorkerThread worker = new WorkerThread();
// current thread gains the worker start token
// and any other permissions returned by the constructor

// check if current thread has worker start token
// check worker preFork permissions and pass to worker
worker.start();
// lose worker start token
// gain worker join token
```

4.3.3 Thread joining

When a thread is joined, the joining threads give their (fraction of the) join token to the joined thread and receive a fraction of the permissions in the special `postJoin` predicate. These permissions are distributed amongst the joining threads based on the size of their join token, by multiplying their join token with each permission to be passed. For example, if the joined thread has a $\frac{1}{2}$ permission to a `num` field, and the joining thread has given a $\frac{1}{2}$ fraction of the join token, the joining thread is given $\frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$ of the `num` permission. If the thread has a full join token, this is equivalent to passing the permissions to the joining thread (since $\frac{1}{2} \times \frac{1}{1} = \frac{1}{2}$). When all joining threads have joined the thread (i.e. when the joined thread has been given the full join token), all the `postJoin` permissions will have been passed to the other joining threads, leaving the joined, finished thread without any remaining permissions.

From the perspective of a thread that joins a `worker` thread, the following occurs:

```
// check if current thread has worker join token
worker.join();
// split or pass permissions based on join token
// remove join token
```

4.3.4 Locks

In the annotation language of VerCors, locks can be annotated with a *resource invariant*, which is a specification of the fields that the lock guards. In this way, specifications may explcitily specify exactly which memory is protected by a given lock. When such a lock is created, the creating thread must pass these permissions to it. When a thread acquires the lock, the permissions are passed to the thread. When the lock is released, the permissions are returned to the lock. For this reason, permission accounting may have to track permissions owned by arbitrary objects.

An example of this can be seen in Listing 4.2. In this example, when a thread acquires the lock, it gains a write permission for `this.x` until it releases the lock, at which point the permission is given back to the lock. Furthermore, the thread that instantiates the lock passes the permissions of the resource invariant to the lock instance.

When checking this specification at runtime, we do not truly need to track the permissions inside the lock. When the lock is instantiated, we must check whether the thread has the required permissions and then remove them from the thread.

Listing 4.2: An example of a specification for a lock that holds permissions

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

//@ resource lockInv() = Perm(this.x, 1);
Lock/*@<lockInv>@*/ lock =
    new ReentrantLock/*@<lockInv>@*/();
```

Directly after the lock is acquired, we may check whether no other threads have access to the `this.x` field before giving write access to the acquiring thread. When the thread releases the lock, the permission is removed.

4.4 Locking in permission accounting

Care must be taken to make the runtime checking code itself thread-safe, i.e. free of thread interference. As lookups are performed, other threads may simultaneously be forking or joining, modifying the permission accounting data. Locks may be used to make permission checks and permission exchanges thread-safe.

Because a permission can only be modified (passed, split or combined with a joined thread's permission) by the thread that owns it, there is no risk of thread interference during permission checks, even when the steps involved in these checks are non-atomic. Thus, using a data structure that safely allows concurrent lookups will reduce the need for locking, only requiring locks for updates to the permissions.

Java's standard library includes a `java.util.concurrent` package containing thread-safe versions of collection classes, including `ConcurrentHashMap`, which supports full concurrency of retrievals and attempts to allow concurrent updates. This is achieved by partitioning the underlying data structure, to allow multiple threads to modify the data simultaneously under the condition that they do not touch the same partitions. When this happens, all but one of the threads are blocked to prevent interference. Using these data structures, no explicit locking by the runtime checking code is necessary.

Chapter 5

Prototype implementation

In this chapter, we will discuss the prototype implementation, which is based on the exploration of Chapter 4. For the prototype, we have chosen to add runtime checking of permissions to fully specified Java programs by transforming the Java source code. This choice has a number of implications for the ways in which permissions can be tracked and checked.

5.1 Permissions accounting

For the prototype implementation, we use the per-field permission accounting approach, in which classes are instrumented with a permission accounting ghost field for each of their fields. This approach does not require any support libraries and it does not generate any new classes for storing permissions, making it relatively simple to implement in *VerCors*, as described in more detail in Section 5.2. A downside of the use of ghost fields is that it makes permission accounting visible to the program via reflection. This scenario is outside of the scope of the prototype.

As described in Chapter 4, the per-field accounting approach allows us to use symbolic permissions. Where the symbolic permissions of Huisman and Mostowski [8] removes the need for join tokens by inferring a thread's right to join another from a permission's transfer history, our implementation does not track permission histories, opting instead to explicitly model the join token as a special token permission for subclasses of Threads. This makes updating the permission accounting simpler, reducing the processing and storage overhead of our runtime checking code.

We use a ghost field of type `Set` for each field of a class, in which we track the set of objects (i.e. threads and locks) that have access to the field. Read or write access is differentiated by the presence of other objects in the set.

5.2 Code transformation

In order to instrument a Java program with runtime checking code for permissions, the Java programs and their permission specification must be parsed.

We have chosen to extend the VerCors tools developed at the University of Twente, rather than writing our own parsers from scratch. The goal of the VerCors project is to perform static verification of permissions in multithreaded programs. At the core of VerCors is a powerful parser and translator of program code, which already parses annotated programs written in a number of languages including Java, translating them into a common abstract syntax tree representation called *Col*. VerCors also allows transforming these ASTs, using the visitor pattern. In the VerCors project, this functionality is used to translate annotated programs written in languages such as Java languages used by static verification tools for (concurrent) software, such as Silver [9] and Chalice [12].

We have implemented our runtime checker on top of VerCors by extending it with a new translation target. Running annotated Java source code through our version of VerCors using the `--checking` target performs instrumentation with runtime permission accounting and checking by transforming the input program source code. This approach allows us to use VerCors' existing capability to parse annotated Java programs as well as its capability to transform an abstract syntax tree back into well-formed Java code. Extending VerCors also allows us to benefit from any future improvements made to it.

The use of Java source code transformation imposes some limitations to the scope of the prototype. These limitations are discussed in more detail in Section 5.4.

5.2.1 Passes

The process of parsing the Java input and instrumenting the AST is performed in the eight passes listed below. Passes listed in bold are part of our prototype extension to VerCors, the other passes are part of the original VerCors tool.

1. Parsing the source Java programs into the common *Col* AST representation.
2. Resolving imports, for the purposes of type checking.

3. Standardizing the AST, standardizing a number of semantically equivalent statements.
4. Performing basic type checking.
5. **Translating else-if statements into nested if-statements.** This circumvents some of the limitations of using code transformation for runtime checking permissions, and is discussed in more detail in Section 5.4.1
6. Replacing assignments and field dereferences with special '`Set!`' and '`Get?`' operators in the AST. This allows the instrumentation pass to more easily determine where to introduce permissions checks.
7. **Instrumenting the AST with permission accounting and checks.** This pass performs the actual instrumentation work and is the main contribution of the project. Sections 5.2.2 and 5.2.3 describe how this pass was implemented.
8. Translating the transformed AST back to Java code, which can then be compiled using any standards-compliant Java compiler.

The `DynamicCheckInstrumentation` class is at the core the seventh pass and traverses the entire AST, adding ghost fields to classes when it encounters field declarations, introducing permission checks for field accesses and method calls, and adding permission instantiation to constructors; See 5.2.2. When declarations of `preFork` or `postJoin` predicates are encountered, the `ForkJoinInstrumentation` class is invoked to process these; See Section 5.2.3.

To avoid name collisions with the source program, the names of all generated ghost fields and methods are prepended with the common prefix '`__checking__`', and classes and interfaces such as `Set` are referred to using fully-qualified names. In the examples in this thesis, these have been omitted for the sake of readability.

5.2.2 The `DynamicCheckInstrumentation` class

The `DynamicCheckInstrumentation` class performs the majority of the instrumentation work by applying the visitor pattern to the entire AST of the input program. It is responsible for:

- Adding permission accounting ghost fields to classes;
- Adding accounting for start and join token permissions to subclasses of `Thread`;
- Surrounding method calls and method bodies with specification checks;

- Adding instantiation of permissions to constructors;
- Prepending each statement with permission checks for fields dereferenced therein;
- Invoking the `ForkJoinInstrumentation` visitor for handling `preFork` and `postJoin` predicates.

5.2.2.1 Adding ghost fields

Whenever the `DynamicCheckInstrumentation` visitor encounters a field declaration, an extra field declaration of type `Set<Object>` is generated and added to a list called `bookkeeping`. When the visitor reaches the end of the class, the fields in the `bookkeeping` list are appended to it, in order to keep all accounting fields together, at the end of the class definition. For example, the `__checking__count` field is generated for the `count` field of the `Counter` class below:

```
class Counter {
  private int count;
  //...

  public Set<Object> __checking__count =
    Collections.newSetFromMap(new
      ConcurrentHashMap<Object, Boolean>());
}
```

When a definition for a class that extends `Thread` (or one of its subclasses) is visited, ghost fields for the start and join permission tokens are also added to the `bookkeeping` list.

5.2.2.2 Instrumenting method declarations

When a method declaration is encountered, the `DynamicCheckInstrumentation` visitor instruments the body of the method depending on its type. Block statements are generated before and after the body of the method, to be filled with permission checks when visiting the sub-tree of the method body.

- Plain method bodies are prefixed with permission checks for its `requires` clause and suffixed with permission checks for its `ensures` clause.
- Instantiation of permissions is appended to constructors, in accordance with its `ensures` clause. Permissions for fields appearing in the `requires` clause

and fields of other classes appearing in the `ensures` clause are checked in the same way as with plain methods.

- The AST represents abstract predicate declarations (such as the `preFork` and `postJoin` predicates) as method declarations in the AST. These declarations are processed by the `ForkJoinInstrumentation` class, which creates ghost methods for checking and transfering the relevant permissions. Predicates other than `preFork` and `postJoin` are not yet supported by the prototype and are skipped; see Section 5.5.2.

5.2.2.3 Instrumenting statements

In front of each statement visited, a block statement is created in which any permission checks for it will be placed. When visiting the sub-tree of the statement, a read or write permission check is generated any time the special `Get?` and `Set!` expressions are encountered, respectively. Note that some `Get?` expressions will generate a write permission check, for instance when they appear as the operand of a pre-increment operator.

5.2.3 The `ForkJoinInstrumentation` class

When `Perm` operators appear in the `requires` or `ensures` clauses of a method, they cause checks for these permissions to be generated in the method's body. However, when these operators appear in the declarations of the `preFork` and `postJoin` resource invariants, these permissions should be checked and then transferred to another thread.

Because of this difference in semantics, the logic for processing these `resource` annotations has been separated out into the `ForkJoinInstrumentation` class. The architecture of VerCors enables multiple visitor implementations to share the same state and transform the same AST in a single pass. `ForkJoinInstrumentation` visits the sub-trees of these predicate declarations and transforms them into helper method declarations in `Thread` subclasses for checking and then transferring the permissions:

- `preFork` declarations are translated into a `preFork` ghost method, which should be called by the source thread before calling this thread's `start` method. The method splits or passes the relevant permissions from the source thread (i.e. the thread executing the method) to the starting thread (the `this`) by updating the permission accounting for each relevant field.

- `postJoin` declarations are translated into `postJoin` and `checkJoinToken` ghost methods.
  - `postJoin`, to be called by joining threads directly after joining this thread, transfers permissions from the joined thread (i.e. `this`) to the joining thread (i.e. the thread executing the method) by updating the permission accounting for each relevant field. Because we use symbolic permissions, we simply add the joining thread to the set of threads with access, and remove the joining thread's join token permission. When no more threads have a copy of the join token, the joined thread releases its permissions by removing itself from the permission accounting for each relevant field
  - `checkJoinToken` should be called by joining thread directly before joining a thread. It simply checks whether the calling thread has the right to join this thread by asserting its ownership of the join token.

5.2.4 Checking and exchanging permissions

Permissions for a field are stored as the set of objects that have access to it, using Java's `Set` interface.

When we wish to pass a permission for a field from some thread A to some other thread B, we simply add B to the set and remove A. When splitting the permission between A and B, we add B without removing A.

The prototype tool generates Java `assert` statements for permission checks. For example, in order to check whether the current thread has read access to field `this.num`, the following assert statement is generated:

```
assert this.__checking__num.contains(
                                Thread.currentThread());
```

When checking for write access, we also assert that the thread has exclusive access:

```
assert this.__checking__num.size() == 1 &&
       this.__checking__num.contains(
                                Thread.currentThread());
```

5.2.4.1 Thread-safety of the runtime checking code

As previously discussed in Section 4.4, no locks are required to perform these checks even through they are non-atomic. In between the `size()` and `contains()` calls, the permissions cannot change in any way that would interfere with the correctness of the assertion, as long as the following two conditions are met:

1. The `Set` implementation used for permission accounting must safely handle lookups even when the set is being updated concurrently.
2. When passing permissions between two threads (or objects), the source thread should be replaced in the set by the target thread atomically, or the target thread should be added to the set before the source thread is removed. If these steps are not performed atomically or in wrong sequence, there may be a window during which some third thread evaluates `size() == 1` to `true`.

The `HashSet` class of the Java standard library wraps around a `HashMap` object. Java offers a `ConcurrentHashMap` class that offers the thread-safety we require for the first condition to be met, but no `ConcurrentHashSet` class. However, the `Collections.newSetFromMap()` method can be used to create an implementation of the `Set` interface that is backed by a `ConcurrentHashMap`.

5.3 Test case: Shared buffer

The main test case for the prototype is a program with three threads, `Main`, `Source` and `Sink`, that share a single buffer. The Main thread creates the Source and Sink threads, acquiring write access to the shared buffer. When the Source thread is started, write access to the buffer is passed to it. The Source thread then fills the buffer (requiring write access) after which the Sink and Main threads join the Source thread, causing the Source thread's write permission for the buffer to be split into two read permissions. The Sink and Main threads both read the buffer (and print its value). Finally the Main thread joins the Sink thread, regaining full write access to the shared buffer. The full source code for the test case can be found in Appendix A. A diagram of the threads can be seen in Figure 5.1.

An example of a detected permission violation is shown in Listing 5.1. In this case, the Sink thread attempted to modify the value in the shared buffer, violating its read permission. It is a limitation of the prototype that the generated error messages are not very descriptive. Section 5.5.3 describes the future work to solve this.

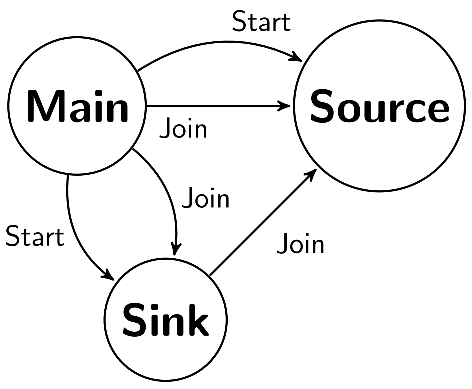

Figure 5.1: A diagram of the threads in the test case.

Listing 5.1: Example output for a permission violation detected at runtime

```
$ java -enableassertions Main
Exception in thread "Sink" java.lang.AssertionError
        at Sink.run(output.java:117)
```

5.4 Limitations of the prototype

Due to chosen implementation approach, time constraints and the limited scope of the prototype, the implemented runtime checker has a number of limitations. Some of these limitations and possible workarounds are discussed in greater detail in this section.

5.4.1 Checking on the statement level

As discussed in Section 4.2, permission checks should occur just before memory is read or written. This is easily achieved when instrumenting byte-code, where these reads and writes are atomic instructions, but requires extensive transformations of the source program when instrumenting high-level Java code. These transformations are discussed in Sections 5.4.1.1 and 5.4.1.2. However, because our research topic was runtime checking of permissions rather than runtime checker design, the prototype compromises on this front by only checking permissions before entire statements, which may cause permissions to be checked for fields that are never truly accessed.

Performing checks on the statement level may lead to false positives (i.e. erro-

Listing 5.2: An example of a limitation of checking permissions before entire statements only.

```
// <check write access to this.counter>
for (; this.counter < 10; ++this.counter) {
  // <check read access to this.counter>
  threadpool[this.counter].start();
  // <write access to this.counter possibly lost here>
}
```

Listing 5.3: A problematic Java program: There is no way to insert a permission check for fields dereferenced in `C2`

```
if (C1) {
  X();
} else if (C2) {
  Y();
} else {
  Z();
}
```

neously reported permission violations where the concrete program does not violate them). In some cases, it may also lead to false negatives (i.e. failure to detect permission violations in the concrete program), particularly when permissions to fields dereferenced in the loop's condition and iterator expressions are modified in its body. For example, Listing 5.2 shows a fragment of code that forks an array of threads by iterating over the array. If any of these threads require read or write access to the `counter` field, the currently executing thread may lose its write access to the field during the loop, causing future iterations of the loop to violate the permission, but this is not detected by the prototype.

It should be noted that the prototype performs some basic transformations to alleviate some of these problems, particularly in order to allow the prototype to insert permission checking statements into the source where they otherwise could not be. The `IfStatementRewriter` pass (the 5th pass of the instrumentation process) rewrites `else if` constructs like the one in Listing 5.3 into semantically equivalent nested `if`-statements like those of Listing 5.4 in order to allow permission checks for the nested `if`-condition to be inserted into the code.

Listing 5.4: After rewriting the if-statements, the permission checks for `C2` can be inserted.

```
if (C1) {
  X();
} else {
  // Permission checks for C2 can be performed here.
  if (C2) {
    Y();
  } else {
    Z();
  }
}
```

5.4.1.1 Rewriting complex expressions

One alternative to instrumentation on the byte-code level is to transform the Java source code more aggressively. The statements generated to check and exchange permissions cannot occur within expressions, which is the root cause of our necessity to check permissions before each statement. As a workaround, we could separate complex statements and expressions into smaller statements so that permissions checks may be performed in between them. Listing 5.5 shows a fragment that may lead to unnecessary (or even erroneous) permission checks in the prototype implementation, when `bob.age >= 65`. Listing 5.6 shows a version of the fragment which is semantically equivalent but allows for permission checks to be inserted in such a way that they are only performed when necessary. Some, but not all, of these transformations are already implemented in VerCors as the *Flatten* pass, which may be extended for this purpose.

Listing 5.5: Example of an expression wherein permissions checks should be inserted, but cannot be, due to the semantics of the Java language.

```
// Prototype implementation checks access to
// bob.age here, and erroneously checks access to
// bob.spouse.age even if bob.age >= 65.
if (bob.age < 65 || bob.spouse.age < 65) {
  // ...
}
```

Listing 5.6: Example of a logical-or expression rewritten to allow permissions checks on sub-expressions.

```
boolean answer = false;
// <check bob.age read access>
int t1 = bob.age;
if (t1 < 65) { answer = true; }
else {
  // <check bob.spouse read access>
  // <check bob.spouse.age read access>
  int t2 = bob.spouse.age;
  if (t2 < 65) { answer = true; }
}
if (answer) {
  // (bob.age < 65 || bob.spouse.age < 65) == true
}
```

Listing 5.7: Instrumented version of the expression of 5.5, using helper methods.

```
if (
     (bob.check_age_read_access() && bob.age < 65) ||
     (bob.check_spouse_read_access() &&
       bob.spouse.check_age_read_acess() &&
       bob.spouse.age < 65)) {
  // ...
}
```

5.4.1.2 Encapsulating permission checks in helper methods

Another alternative to for instrumenting high-level Java code is by generating helper methods for performing permission checks, since method bodies can contain arbitrarily many statements and method calls may occur in expressions. For instance, the fragment of Listing 5.5 could be instrumented to use helper methods to check permissions inside expressions as shown in Listing 5.7.

In order to check permissions for fields of objects returned by methods, we must cache the return value in a temporary variable, because the method may have side-effects. For example, given the expression `people.getNext().age >= 65`, we must cache the object returned by `getNext` in order to check the thread's read permission for the returned object's `age` field. However, defining variables within expressions is not allowed, further complicating our source code transformation.

Anonymous inner classes, or the lambda expressions newly introduced in Java 8, offer a way to introduce these helper methods and allow the definition of temporary variables within expressions, although this may cause a substantial performance overhead. An example of such an approach using an anonymous inner class is shown in Listing 5.8.

Listing 5.8: An example using an anonymous inner class to introduce complex permission checks within an expression

```
interface Checker<T> {
  T check();
}

// people.getNext().age >= 65
(new Checker<Person>() {
  public Person check() {
    Person p = people.getNext();
    // check read access to age field:
    p.__checking__age.contains(Thread.currentThread());
    return p;
  }
}).check().age >= 65;
```

5.4.2 Visibility of fields

Even when a field of a class is private or protected, the permission accounting for that field must be public, in order for the generated checking code to access it. This violates the OOP concepts of encapsulation and information hiding in the instrumented code. Since the generated code is not meant to be read or modified by programmers, this is only a minor issue.

5.4.3 Support for partially specified programs

The prototype does not support programs with partial specifications (such as programs that use external libraries), as the tool may erroneously generate permission checks for un-instrumented methods or classes, and external code may cause the permission accounting to be invalidated. The prototype implementation avoids generating checks for classes in the `java.*` namespace, allowing classes from the

Java standard library to be used in programs. The tool cannot generate permission checks for these classes, so care must be taken when using classes that are not documented to be thread-safe.

When instrumenting code, an annotated version of the Thread class must be supplied as input to VerCors, but must not be used when compiling the instrumented program.

5.5 Missing features/checks

A number of features of the permission specification language of VerCors are not yet supported in the prototype, due to time constraints. This section describes the major omissions and explores possible ways to implement these in the future.

5.5.1 Resource invariants

The prototype currently only supports passing permissions between threads, on fork and join. Permission exchanges when locks are acquired and released have not yet been implemented, although the permission accounting ghost code generated can track permissions owned by objects, as the ghost fields are of type `Set<Object>` rather than `Set<Thread>`.

An implementation of this feature should allow storing permissions inside a lock, passing the permissions to a thread when it acquires the lock, and passing the permissions back to the lock when it is released, according to the specification of the lock.

5.5.2 Abstract predicates

When implementing support for abstract predicates in our runtime checker, we can replace simple (i.e. non-recursive) predicates within the specifications with their constituent parts before processing the specification.

In order to check or pass the permissions for a recursive predicate at runtime, we may generate a helper method that accepts the same parameters as the predicate and evaluates the predicate, returning the outcome (i.e. `true` or `false`). By then asserting that a call to the helper method returns `true`, a recursive predicate may be checked.

5.5.3 Detailed error messages

The prototype tool currently generates simple `assert` statements for permission checks. Violations of the permission specification cause the JVM to throw a `java.lang.AssertionError` exception with a stack trace showing line numbers of the instrumented version of the program, which may differ wildly from the relevant line numbers in original source code.

We can add more descriptive error messages to the Java assert statements, containing details of the permission that were violated and the relevant line numbers in the original program, by using the alternative `assert` syntax:

```
assert Expression : error-message
```

Chapter 6

Optimizations and future work

In this chapter, we discuss some possible optimizations for a runtime checker of permissions, as well as implementation ideas for a production-ready version of such a tool.

6.1 Reducing redundant checks

Using the knowledge that the set of permissions owned by the currently executing thread can only change at specific points (fork, join and when locks are acquired and released) it is possible to significantly reduce the number of permission checks performed. After a particular permission has been checked, there is no need to check it again until such a time when the permission may have changed. An example of such a scenario is shown in Listing 6.1. Once it has been established that the thread has write access to the `seconds` field at the first pre-incrementation, there is no need to check for read and write access in the `if`-condition and the second incrementation, respectively, since there is no way for the thread to have lost its exclusive access to the permission in between these statements.

A simple way to detect redundant permissions checks at runtime is to cache (per thread) the results of permission checks and clearing this cache whenever permissions are exchanged. However, since most permission checks are only a single HashSet lookup, such a cache lookup would not reduce the runtime overhead of the runtime checker.

Redundant checks can be omitted entirely during the code generation step using a similar caching approach, by remembering which permissions checks have been inserted into the program since the last time any permission exchanges may have

Listing 6.1: Example of a program with redundant permission checks for the `seconds` field removed

```
class Timer {
  int minutes;
  int seconds;

  void step() {
    // <check write access for this.seconds>
    ++this.seconds;
    if (this.seconds == 60) {
      // <check write access for this.minutes>
      ++this.minutes;
      this.seconds = 0;
    }
  }
}
```

occurred. Care should be taken with method calls, as they may cause permissions to be changed, and loops, as previously demonstrated in Listing 5.2. A safe approach would be to clear the cache when method calls are encountered, and to always generate checks inside the body of loops, but some static analysis of the source code may allow permission checks to be omitted even in these cases, or to only clear the cache for permissions that may have been lost or split since they were last checked.

6.2 Permissions for array elements

In the current implementation, permissions for fields of array types have not been tested, and no explicit handling of arrays (such as dynamic array instantiation) has been implemented.

In the case of arrays, permissions for individual array elements (as well as the array as a whole) may be required, so that different threads can safely modify separate elements of the array. This commonly happens if an array stores input data for an algorithm that is parallelised by partitioning the problem, such as QuickSort.

6.3 Byte-code transformation and JVMTI

As previously discussed in Section 5.4.1, transforming compiled byte-code rather than Java source code is a good way to perform runtime checking instrumentation such that permissions can be checked within expressions.

Listing 6.2 shows a Java statement and its corresponding byte-code.

Listing 6.2: JVM bytecode for a java expression

```
//  boolean result = this.x > 0 && this.y > 0;
//  ------
 0: aload_0         // Push constant 0 to stack
 1: getfield   #2   // Push field x to stack
 4: ifle       18   // If 0 <= x, jump to 18
 7: aload_0         // Push constant 0 to stack
 8: getfield   #3   // Push field y to stack
11: ifle       18   // If 0 <= y, jump to 18
14: iconst_1        // Push constant 1 to stack
15: goto       19   // Jump to 19 (i.e. skip 18)
18: iconst_0        // Push constant 0 to stack
19: istore_1        // Pop stack head to 'result' variable
```

To avoid erroneously checking the permission for the `y` field in the event that $x \leq 0$, the permission checks for `y` may be inserted between the 8 and 11 instructions, making sure to leave the stack in the same state before and after the inserted permission checks.

Possible ways to perform byte-code instrumentation based on specifications found in annotated Java source code include:

- Extending the Java compiler to generate instrumented byte-code during compilation. The OpenJML runtime assertion checker operates in this way, by extending the OpenJDK Java compiler [4].
- Using a tool to parse the specifications and generate metadata about the permission checks to be performed. Instrumentation of the byte-code can then be done Just-In-Time when the compiled classes are loaded into the JVM using custom `ClassLoader`s or using an external tool that hooks into the JVM using the JVM Tool Interface (JVMTI).

Both solutions would likely require, or greatly benefit from, an alternative permissions accounting approach, in which the permission accounting is separated from the classes.

The latter approach, using the JVM Tool Interface, would make it possible to completely disable the runtime checks and the permissions accounting, as the compiled program would be completely un-instrumented.

Chapter 7

Conclusion

In this project we have successfully determined that it is possible to create a runtime checker for permissions.

The prototype we have developed shows that it is possible to integrate the required runtime checks into a Java program by transforming its source code. Due to the semantics of the permissions, checking permissions does not require any locking, which helps performance. Using the appropriate permission accounting approach, even permission transfers may often be performed without locking.

Permission specifications used for static verification can be re-used for runtime checking. However, for some ambiguous specifications we are required to choose a default behaviour. Extending the specification language with annotations that clear up these ambiguities may make implementing a runtime checker for permissions simpler or make the specifications more elegant.

Since permissions are used to guard memory locations from reads and writes, byte-code instrumentation is a perfect candidate for implementing a production-ready runtime checker, because these read and write operations are represented as individual instructions in the byte-code.

Appendix A

Test case source code

```
class Source extends Thread {
  int i;

  //@ resource preFork() = Perm(this.i, 1);
  //@ resource postJoin(frac p) = Perm(this.i, p);

  public void run() {
    i = 42;
  }
}

class Sink extends Thread {

  Source source;

  public Sink(Source source) {
    this.source = source;
  }

  //@ resource preFork(frac p) =
        this.source.joinToken(p) ** Perm(this.source, 1);
  //@ resource postJoin(frac p) =
        Perm(this.source, 1) ** Perm(this.source.i, p);

  public void run() {
    try {
      source.join();
    } catch (Exception e) {
      System.err.println(e.toString());
    }
    System.out.print("Sink: ");
    System.out.println(source.i);
  }
}

class Main {
  public static void main(String[] args) {
    Source source = new Source();
    Sink sink = new Sink(source);

    try {
      source.start();
      sink.start();

      source.join();
      System.out.print("Main: ");
      System.out.println(source.i);
      sink.join();
    } catch (Exception e) {
      System.err.println(e.toString());
    }

    // Verify that we have regained write access.
    source.i = 1988;
    System.out.println(source.i);
  }
}
```

Bibliography

[1] Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Verification of Concurrent Systems with VerCors. In Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer, editors, *Formal Methods for Executable Software Models*, volume 8483 of *Lecture Notes in Computer Science*, pages 172–216. Springer International Publishing, 2014. ISBN 978-3-319-07316-3. doi: 10.1007/978-3-319-07317-0_5. URL `http://dx.doi.org/10.1007/978-3-319-07317-0_5`.

[2] John Boyland. Checking Interference with Fractional Permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40325-6. URL `http://dl.acm.org/citation.cfm?id=1760267.1760273`.

[3] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[4] David R. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-20397-8. doi: 10.1007/978-3-642-20398-5_35. URL `http://dx.doi.org/10.1007/978-3-642-20398-5_35`.

[5] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Goanna — a Static Model Checker. In *Formal Methods: Applications and Technology*, pages 297–300. Springer, 2007.

[6] Robert W. Floyd. Assigning Meanings to Programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[7] Charles Anthony Richard Hoare. An Axiomatic Basis for Computer Pro-

gramming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL `http://doi.acm.org/10.1145/363235.363259`.

[8] Marieke Huisman and Wojciech Mostowski. A Symbolic Approach to Permission Accounting for Concurrent Reasoning. In *14th International Symposium on Parallel and Distributed Computing, ISPDC 2015, Limassol, Cyprus, June 29 - July 2, 2015*, pages 165–174, 2015. doi: 10.1109/ISPDC.2015.26. URL `http://dx.doi.org/10.1109/ISPDC.2015.26`.

[9] Uri Juhasz, Ioannis T. Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. Technical report, ETH Zurich, 2014.

[10] Jorne Kandziora. Runtime Assertion Checking of Multithreaded Java Programs. Master's thesis, Universiteit Twente, 2014.

[11] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, pages 404–420, 1998.

[12] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of Concurrent Programs with Chalice. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03828-0. doi: 10.1007/978-3-642-03829-7_7. URL `http://dx.doi.org/10.1007/978-3-642-03829-7_7`.

[13] Francesco Logozzo. Practical Verification for the Working Programmer with CodeContracts and Abstract Interpretation. In *Proceedings of the 12th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'11)*. Springer Verlag, January 2011. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=141092`.

[14] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In Laurent Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42554-0. doi: 10.1007/3-540-44802-0_1. URL `http://dx.doi.org/10.1007/3-540-44802-0_1`.

[15] Matthew Parkinson and Gavin Bierman. Separation Logic and Abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 247–258, New York, NY, USA,

2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040326. URL `http://doi.acm.org/10.1145/1040305.1040326`.

[16] Alan Mathison Turing. On Computable Numbers, with an Application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.