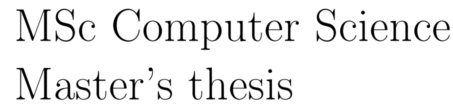

MSc Computer Science
Master's thesis

Design and Implementation of new features for Runtime Permission Verification in Concurrent Java Programs using VerCors

Dylan Damian Janssen BSc.

Supervisor:
Prof. dr. Marieke Huisman
Pieter Bos MSc.

External examiner:
Dr. Ir. Pieter-Tjerk de Boer

April, 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

Concurrent programs increase productively significantly for computers, but they can also introduce bugs, like data races and deadlocks. VerCors is a static verification tool that can verify using formal specifications that the concurrent programs behave correctly. VerCors also wants support for runtime verification for Java programs using the same formal specifications as static verification.

Gijsen implemented an initial prototype in 2015 to perform basic runtime permission checking in VerCors. However, the prototype does not work with the current version of VerCors and uses an approach that is different from the annotations from VerCors, which limits the addition of new features to the runtime checking prototype. Additionally, the prototype was only limited in its functionality, it only provides support for basic permission checking.

In this master's thesis, we will first identify how to rework the old prototype to a new prototype that does work with the current version of VerCors. Additionally, we will design and implement new features for implementing permission checking for arrays, quantifiers, loop invariants, lock invariants and predicates. We identify different options to implement the new features in the prototype and implement the mentioned features into the prototype.

Each feature is evaluated using a test case. The test cases should not throw any error if they are executed and by altering the test cases slightly, the test case should throw an error. All test cases throw the error that was expected and thus the prototype works as intended. The test cases produce a method and line coverage of 89% and 92% respectively, which shows that most of the prototype is tested thoroughly. Finally, the test cases are significantly faster for runtime verification than static verification. Runtime verification took an average of 5.039 seconds while static verification took 14.923 seconds.

Keywords: runtime verification, static verification, VerCors, runtime assertion checking, program verification

Acknowledgements

I would first like to thank the persons who helped and supported me during the creation of this master's project.

First I would like to thank Prof. Dr. Marieke Huisman for being my supervisor during my research topics and master's project. I am thankful for all the feedback and insights that you provided during the project.

Additionally I would like to thank Pieter Bos MSc. for being my second supervisor during my research topics and master's project. Due to your technical insights into VerCors and Scala, I was able to start the implementation of the prototype relatively quickly. Moreover, whenever I had a question I could contact you directly and you would reply within a couple of minutes.

I would also like to thank Dr. Ir. Pieter-Tjerk de Boer for being my external examiner. During my bachelor studies, I always enjoyed the courses that you taught me and you are a great teacher.

This master's thesis has been performed at ALTEN Netherlands B.V., which I would like to thank for their technical expertise and guidance. In particular, Hugo Logmans, Nick Eilander BSc., and Anil Özen BSc. Hugo Logomans always enabled me to have different views on certain problems and provided me with feedback on my presentational skills. Nick Eilander for providing me with all the resources I needed to accomplish this project. Finally, Anil Özen for being my coach during the project and for your keen eye for details in my report

Finally, I would like to thank my family and friends for supporting me during the master's project period. My parents, Heleen Janssen van Loon BSc. and Maarten Janssen MD. PhD, my sisters, Rachel Janssen MD. and Sharon Janssen MSc, and my girlfriend Pleun Hakvoort BSc. were always there to motivate me during the project. I would like to thank my friends for being there during my studies and providing me with technical feedback and support on the project.

Contents

1 Introduction **5**
1.1 Motivation . . . 5
1.2 Problem statement . . . 6
1.3 Research Questions . . . 6
1.4 Outline . . . 7

2 Problem domain **8**
2.1 Concurrency . . . 8
2.1.1 Bugs in concurrent programs . . . 8
2.2 Formal specifications . . . 9
2.3 Static verification . . . 11
2.3.1 Problems with static verification . . . 11
2.4 Runtime checking . . . 12
2.5 Byte code transformation . . . 13

3 Permission logic **14**
3.1 Seperation logic . . . 14
3.2 Fractional permissions . . . 15
3.3 Symbolic permissions . . . 16

4 VerCors **17**
4.1 What is VerCors? . . . 17
4.2 VerCors internal structure . . . 17
4.3 Syntax . . . 18

5 Implementation Design **22**
5.1 Basic permissions . . . 22
5.1.1 Storing permissions . . . 22
5.1.2 Checking permissions . . . 24
5.1.3 Exchanging permission . . . 26
5.1.4 Locking . . . 27
5.2 Array permission . . . 27
5.3 Quantifiers . . . 28
5.4 Loop invariants . . . 30
5.5 Predicates . . . 30
5.6 Prototype Implementation Overview . . . 33
5.6.1 Ledger . . . 33
5.6.2 Class Descriptions . . . 33

6 Testing & Results **36**
6.1 Test case Shared Buffer from Gijsen . . . 37
6.1.1 Results . . . 38
6.2 Test case Shared Buffer for arrays . . . 38
6.2.1 Results . . . 38
6.3 Quantifiers test case . . . 39
6.3.1 Results . . . 39
6.4 Loop Invariant test case . . . 39
6.4.1 Results . . . 39
6.5 Lock Invariant test case . . . 39
6.5.1 Results . . . 40
6.6 Predicates test case . . . 40
6.6.1 Results . . . 40

7 Related work **41**
7.1 Runtime verification for sequential programs . . . 41
7.1.1 Frama-C . . . 42
7.1.2 Whiley . . . 42
7.2 Runtime verification for concurrent programs . . . 42
7.2.1 Old prototype . . . 42
7.2.2 OpenJML . . . 43
7.2.3 Concurrent and Distributed systems . . . 43
7.3 Static verification tools concurrent programs . . . 43
7.3.1 VerCors . . . 43
7.3.2 VeriFast . . . 43
7.3.3 VCC . . . 43
7.3.4 Chalice . . . 44
7.3.5 GPUVerify . . . 44

8 Conclusion **45**

9 Future work **47**
9.1 New features . . . 47
9.2 Automated testing environment . . . 47
9.3 Scoping permissions per function . . . 47

A Detailed Prototype Implementation **50**
A.1 Details Helper Classes . . . 50
A.1.1 Util . . . 50
A.1.2 LedgerHelper . . . 50
A.1.3 CreateConstructor . . . 51
A.1.4 PermissionData . . . 51
A.1.5 FindBoundsQuantifier . . . 51
A.1.6 AbstractQuantifierRewriter . . . 51
A.1.7 RewriteContractExpr . . . 52
A.1.8 TransferPermissionRewriter . . . 52
A.2 Details Stage Classes . . . 52
A.2.1 CreateLedger . . . 52
A.2.2 RemoveSelfLoops . . . 57
A.2.3 RefactorGeneratedCode . . . 57

A.2.4 AddPermissionOnCreate . . . 57
A.2.5 CreatePredicateFoldUnfold . . . 58
A.2.6 CheckPermissionsBlocksMethod . . . 58
A.2.7 CreatePrePostConditions . . . 60
A.2.8 CreateLocking . . . 61
A.2.9 CreateLoopInvariants . . . 61
A.2.10 ForkJoinPermissionTransfer . . . 61
A.2.11 GenerateJava . . . 62

B Code of tests **63**
B.1 Test case for reviving code Gijsen . . . 63
B.2 Test cases for new features . . . 64
B.2.1 Arrays . . . 64
B.2.2 Quantifiers . . . 67
B.2.3 Loop Invariants . . . 68
B.2.4 Lock Invariant . . . 69
B.2.5 Predicates . . . 69

Chapter 1

Introduction

Many bugs and issues can occur when working on programs and this is often challenging to verify the behaviour of the implementation. It is important to find these bugs because they can lead to undesired program behavior and have significant consequences, such as the loss of important data. Fortunately, there are many different testing methods to do this, e.g. white box, black box, and model-based testing. Next to testing methods, there is also static verification, which verifies if the code conforms to the specifications of the program. These techniques work on sequential programs, but it is also necessary that concurrent programs are verified as well. Additionally, runtime verification is another technique to verify the behaviour of the program. In runtime verification, the program will be verified while the program is being executed compared to static verification which is being validated without executing the program. In this work, we will provide the design and implementation of a runtime verification for concurrent processes extension for VerCors [3]. VerCors is a tool that is being developed by the University of Twente in the Formal Methods and Tools group (FMT)[1]. VerCors, at the moment of writing, is a tool that statically verifies programs using specifications provided in annotations and it is specialized at concurrent software.

1.1 Motivation

Concurrent programs are used to increase the performance of a program significantly or they can give a user feedback in a User Interface (UI) while other threads perform other background tasks.

Although concurrent programs have a significantly high performance due to the use of multiple threads, they are also more sensitive to bugs and it can be more complex to understand the behaviour of the software. This is because multiple threads can access fields of objects at the same time and therefore introduce data races. To counter data races, it is possible to introduce locking. On one hand, this prevents other threads from accessing a certain field of an object simultaneously and therefore solves the data race issue. On the other hand, it also removes the purpose of creating a concurrent program if other threads have to wait till a thread is finished. Additionally, it can also introduce deadlocks, which means that two threads are waiting for each other to release a lock and thus not execute at all (see Chapter 2 for more information on data races and deadlocks). So by using con-

[1] Formal Methods and Tools group (FMT): https://www.utwente.nl/en/eemcs/fmt/

current software, data races and deadlocks can be introduced, and because of that makes it harder to ensure that the code is thread-safe.

Some tools can verify if code is thread-safe, for instance, VerCors [3] and other tools, see Chapter 7. VerCors verifies if the code is thread-safe by using function contracts that specify permission specifications for fields. Permission specifications describe how much permission a thread has for an object or field in the program. Despite using modular verification, VerCors still requires a significant time to verify larger software. Also, everything in a program needs to be specified to let the static verification succeed. A solution for this would be using a runtime checking solution, which means that the program is validated during runtime. In runtime verification, the program will not be verified statically (without executing the program) but will be validated during execution. This is done by adding assert statements to the source code, so the behaviour of the program can be verified. This is useful because it will give a fast indication if the system behaves correctly, but it can never show that it covers all the cases of the program behaviour. By combining the runtime and static verification techniques a developer can get a fast indication if the program works as intended by using runtime verification and in the end, prove that the behaviour of the program is correct by using static verification.

1.2 Problem statement

An initial prototype of an extension of VerCors for runtime checking for basic permissions was created by Gijsen. [9]. VerCors has changed significantly over the past years, resulting in the original code not being functional anymore. Moreover, the prototype only covers of a small subpart of the functionality that VerCors provides. For instance, the prototype lacks permission with locks, array permissions, quantifiers in annotations, loop invariants, and the use of predicates. Additionally, the prototype makes use of symbolic permission checking, while VerCors uses fractional permission checking. It is desirable that the runtime verification checker also uses fractional permissions because then the same annotations can be used for verifying the program statically and during runtime.

Gijsen provided a suggestion to use byte code alteration to rewrite as little source code as possible. Byte code alteration would enable the possibility to check permissions anywhere and anytime in the program. Therefore research should be done to investigate if byte code alteration is worth implementing.

Summarizing what must be done during this research is to provide a solution to revive and extend the project of Gijsen to work with the current version of VerCors and its syntax, investigate if byte code alteration is worth implementing, and introduce new features such as permission with locks, array permissions, quantifiers in annotations, loop invariants, and the use of predicates for checking concurrent Java programs.

1.3 Research Questions

To create a prototype of the runtime verifier of concurrent programs, we need to answer the following research questions:

- What are the different techniques to do runtime verification using fractional permissions?
  - How will the different techniques be applied to the prototype of Gijsen to revive the project?
  - What are the benefits of using byte code alteration?
- How can features be integrated into VerCors, and what design considerations should be taken into account?
  - How can basic permissions be integrated into VerCors?
  - How can array permissions be integrated into VerCors?
  - How can quantifiers be integrated into VerCors?
  - How can predicates be integrated into VerCors?
  - How can locks be integrated into VerCors?
  - How can loop invariants be integrated into VerCors?
- What is the performance and coverage of the prototype compared to static verification?

1.4 Outline

This thesis will have the following structure. First, we will discuss the problem domain in Chapter 2. In Chapter 3 we will discuss how permissions can be handled in verification techniques. After that, we will introduce how VerCors works internally and syntactically in Chapter 4. Using the gained knowledge from the previously mentioned chapters, we will discuss different options to implement the features into VerCors and how they are implemented in Chapter 5. To verify if the prototype works correctly we perform tests on the prototype and show the results of those tests in Chapter 6. In Chapter 7 we will discuss the related works to this thesis. Finally, Chapter 9 we will discuss the possible future work.

Chapter 2

Problem domain

In this chapter, we will discuss the problem domain of concurrent programs. This includes what concurrent programs are, the bugs that can occur, what static verification is, and what runtime checking is.

2.1 Concurrency

In the early days of computer science, it was normal to have sequential programs. A sequential program is executed on one thread. This is fine if the program is not large and only needs to perform a couple of tasks. Programs gradually require a lot more calculating power or have more tasks running in parallel. This is achieved by having the program executed on multiple threads, by dividing a computationally heavy task over multiple threads, or by running the different processes on different threads.

At the beginning of designing concurrent programs (the 1960s), most computers only had one processor core with one logical thread. This meant that the different tasks of a program would be scheduled by a scheduler to execute multiple processes at the same time on a single core. Nowadays, almost every device has a multi-core processor with sometimes multiple logical threads as well. This means that multiple processes can run on their logical thread and in parallel. To determine which logical threads a process runs is still determined by a scheduler, which is often built-in into the operating system.

2.1.1 Bugs in concurrent programs

Although the performance boost of concurrent programs is desirable, it also introduces bugs that cannot occur in sequential programs. These bugs cannot occur because in sequential programs each line of code is executed after the other, so there is no other process that can use the same resources as another thread.

The first bug that can occur in concurrent programs is data races, described by Pun et al. [21]. Data races can occur when two threads are accessing or altering the same field at the same time and at least one of the threads is writing. In Listing 2.1, you can see an example of a data race bug. In this example, we have the class *BankAccount*, which keeps track of the balance of the account. If two or more threads would invoke the *add* method at the same time, it is possible that the balance would not be correct anymore. In Table 2.1 we can see how the data race can occur. The starting balance is 100 and there is one thread that wants to add 10 to the balance value and one thread that wants to add 20 to

the balance. Every thread has its own stack, but they access the *balance* field from the heap memory. First, thread one reads the value of the balance, adds 10 to it, and stores it on its stack. Then thread two reads the value of the balance adds 20 to it and stores it back on the heap. Finally, thread one stores the value of its stack to the heap. This results in a data race, because the end value of the balance is now 110 instead of 130.

```
class BankAccount {
    int balance = 0;

    public void add(int difference) {
        balance += difference
    }

}
```

LISTING 2.1: Example of implementation that is prone to data races

Thread one	Thread two	Balance
read value balance		100
increment value in stack by 10		100
	read value balance	100
	increment value in stack by 20	100
	write stack to balance field	120
write stack to balance field		110

TABLE 2.1: Example flow data race bug

Fortunately, it is possible to avoid data races by using locks. A lock ensures that a thread will wait until the thread that is operating on a field has released the lock. However, if this is not implemented correctly by a developer it is possible that a deadlock can occur, see Pun et al. [21]. A deadlock occurs when two threads have acquired two different locks, but want to request each other's lock as well. This would mean that two threads are waiting for each other to finish, thus resulting in no execution at all. Table 2.2 shows an example of a deadlock, where two threads want to transfer money from one balance to the other. Thread one reads and locks the first balance one, then Thread two reads and locks balance two. Now they also need to know the other balance to do the transfer, so both threads attempt to get the lock of the other balance, which fails because both balances are locked already. Now both threads will wait infinitely for the lock of the other balance which is held by the other thread.

Additionally, it is possible that the misuse of locks does not prevent the data race problem. For instance, if a different lock is used for reading and writing, it is still possible that a thread can alter the value while another thread is reading the value. Thus creating a data race even though locks are used.

2.2 Formal specifications

Formal specifications define, according to Thomson et al. [23], the characteristics that a system is to embody, where each statement should be defined so that it is proveable to be correct. According to Hierons et al. [10], a variety of different formal specification

Thread one	Thread two
Read value balance one (lock balance one)	
	Read value balance two (lock balance two)
Waiting to read value balance two due to lock	
	Waiting to read value balance one due to lock
Both infinitely waiting for each other	

TABLE 2.2: Example deadlock

techniques exists. They discussed some of the more popular notations, which will be discussed briefly.

First, the *Finite State-Based Languages* defines the states of a finite set of values. This model represents what happens when certain inputs and outputs happen in a state and what the next state will be.

The second form is the *Process Algebra State-Based Languages*, which describes the system as several communicating concurrent processes. This language works similarly to the *Finite State-Based Languages*, but can also keep track of traces of inputs and outputs.

The third form is the *Hybrid Languages*, which encompasses both discrete and continuous mathematics. This is because many systems use a combination of analog and digital components.

The fourth form is the *Algebraic Languages*, which describes a system using algebraic properties. The language describes the behaviour of the systems in terms of *axioms*. This approach is useful for functional programming languages.

Finally, we have the *Model-Based Languages*, which is an approach by building a model of the intended behaviour of the program. This is often done by defining pre- and post-conditions for functions. The conditions can be proved correctly using the Hoare logic [11]. If these conditions do not hold then the program might not work as intended, which means that a bug can occur in the program. This type of formal specification we will use in this document.

Programs that use *Model-Based Languages* to prove the behaviour of a program are, for instance, the Java Modelling Language (JML) [19] or the ANSI/ISO C Specification Language (ACSL) [7]. See Listing 2.2 for an example of how JML can be used to prove the behaviour of a function that calculates the area of a square. In the specification, we require the *sideLength* to be a positive number and the result of the function is the area based on that *sideLength*.

VerCors also uses the *Model-Based Languages* to prove the behaviour. Permission-Based Separation Logic (PBSL) is used to specify the behaviour of a program. PBSL is an implementation of the fractional permissions introduced in Section 3.2. Since the static verification of VerCors uses PBSL, the prototype also uses the same PBSL annotations to prove the behaviour of programs.

```
//@requires sideLength > 0
//@ensures \result == sideLength * sideLength
public int areaSquare(int sideLength) {
    return sideLength * sideLength
}
```

LISTING 2.2: Example formal specifications using JML

2.3 Static verification

Static verification is a process where a program is verified without running it. For instance, some compilers perform static verification on the program to ensure the type safety of a program. Static verification is often done using formal methods, which are mathematical models of the code and checking if the behaviour of the model is correct using specifications. A formal specification is called sound if the program is verified then the program will not throw exceptions and will always behave to the specifications. It is also possible to do unsound verification by only traversing through a subset of the model.

2.3.1 Problems with static verification

Static verification can prove that the program works correctly, but there are also challenges with using static verification. Hähnle et al. [13] identified 24 challenges for deductive software verification. In their research, they split the challenges into technical- and non-technical. Each challenge represents a task that could be solved for further development in deductive software verification. Also, each challenge is categorized into a certain topic and this topic explains an overall current problem in deductive software verification. We will briefly explain these topics what the problems are in deductive verification and what the effect of using runtime verification is. Topics for deductive software verification are:

1. Technical:

    (a) **Specification**: All specifications must be precise to let the verification proof succeed, even if some specifications are trivial from the code and can be inferred into the specification in a (semi) automated matter. When using runtime verification it is not required that all specifications must be precisely defined, so that only one property is verified in the system.

    (b) **Integration**: Different programming languages need different tools to perform the verification. Unfortunately, there is no one-fits-all solution for verifying all types of programs. This cannot be solved in runtime verification because the runtime checks have to be implemented in the same language as the source language. Thus a new tool needs to be created for each language.

    (c) **Coverage**: Verification techniques need to take functional- and non-functional techniques into account. When new research is done to extend the deductive verification to concurrent programs, the problems need to be resolved in the other tools as well. This also holds for runtime verification.

2. Non-Technical:

    (a) **Usability**: The effectiveness of the deductive verification tools is very high, but using the tools during the existing development environment can be cumbersome. At the moment there are only a few usability studies being done, therefore more research must be done to adopt deductive verification in more programs. The effectiveness of runtime verification is lower than deductive verification because we can never prove that runtime verification has covered all paths in the program. However, the runtime verification approach can increase the use of verification, because not all specifications have to be defined and thus is easier to use.

    (b) **Funding**: Building tools that perform deductive or runtime verification is an expensive process where adding support for different industrial languages takes

a long time. Additionally, not all problems are solved, such as floating point types and weak memory models, so research is still being done which requires money and time. Funding the building of the tools and research is difficult. This also applies for runtime verification.

(c) **Industrial and Social Context**: Programming languages evolve and are not designed to take verifiability in mind. This makes formal verification and coverage a difficult task. This is also a problem in runtime verification, because if the programming language changes then the runtime verification does not work as well.

2.4 Runtime checking

Unlike the static verification process, it is possible to do the verification during runtime. Static verification verifies the program based on the code and the formal specifications, this happens without executing the program. Runtime verification validates the program on the formal specifications while the program is being executed. For instance, pre- and post-conditions of a method will be verified at the beginning and ending of the method respectively. Additionally when variables, fields or objects are used check if the thread has enough permission to access it. All the verification steps are evaluated in assert statements, which throw an error if the program does not behave correctly. This has the advantage that there is a fast indication if the program behaves correctly or not, but it can never prove if the program is working completely correctly because it might not have covered all the edge cases. Moreover, it also requires more processing power to check that the code is correct, this can lead to performance issues for running the program.

To use runtime assertion checking, a program needs to be transformed so that assertions are added to the program that are specified in the formal specifications. Listing 2.3, shows how the simple function from Listing 2.2 can be transformed into a program that can be verified by runtime assertion checking. The *setLength* method in the example will set the *length* of the class to the *newLength* that is provided as a parameter. There is one pre-condition that requires the *newLength* parameter to be larger than 0. As shown in the example the pre-condition is translated to an assert statement. Next to the pre-condition, another assertion is added when the *this.length* field is accessed, to ensure that the thread has enough permission to change this field. Runtime assertion can only check if the conditions hold for the current inputs and variables, thus it will not check all the possible inputs and variables as in static verification.

```
//@requires newLength > 0;
public void setLength(int newLength) {
    assert(newLength > 0);

    // assert check write permission this.length
    this.length = newLength;
}
```

Listing 2.3: Example formal specification into runtime assertion checking pre-condition

2.5 Byte code transformation

Gijsen [9] proposed to use byte code transformation to perform assertion checking only when variables are used. This section will discuss byte code transformation and its advantages and disadvantages.

Byte code transformation is a process where the byte code of a program is altered. Using byte code transformation it is possible to change the program when it is loading into the Java Virtual Machine (JVM). This can be used for runtime verification because no source code will be changed during the process, only byte code will be added that checks whether the code is correct. In Listing 2.4 you can see an example code that benefits from byte code transformation. According to Gijsen if you want to check permissions for property b then this would require either rewriting the source code to Listing 2.5 or checking the permission before the if statement (which leads to performance drawbacks). In byte code alteration it is possible to do the permission check before the variable is loaded into the stack and thus it can verify if the program has enough permission and the source code does not have to be altered. However, byte code transformation is a complex process and should be done with care, because it can break programs relatively quickly if it is not done correctly. Since a lot of code needs to be added to the program, therefore more byte code operations be added to perform permission checking. One mistake in a byte code operation can cause a failure in the program and thus should be implemented with care. Additionally, VerCors does not have existing tools built-in to work with byte code alteration. Thus a completely new system needs to be created for byte code alteration. Therefore it is more convenient to use the tooling that VerCors already provides and alter the program such that it can perform runtime verification.

```
public void threeOptions() {
    if(a) {
        return "a"
    } else if(b) {
        return "b"
    } else {
        return "c"
    }
}
```

Listing 2.4: Example code that would benefit from byte code alteration

```
public void threeOptions() {
    if(a) {
        return "a"
    } else {
        if(b) {
            return "b"
        } else {
            return "c"
        }
    }
}
```

Listing 2.5: Rewritten code of Listing 2.4 for checking permissions that is required instead of using byte code alteration

Chapter 3

Permission logic

This chapter discusses how permissions can be handled in verification. Therefore, we will discuss Separation logic, Fractional permissions, and Symbolic permissions.

3.1 Seperation logic

To understand the concept of separation logic [18] we have to explain Hoare logic [11] first. Hoare logic is a method to verify a computer program with the use of a set of logical rules. For this, the Hoare triple is used, which is in the form of $\{P\}C\{Q\}$ where P represents assertions for the preconditions, Q the assertions for the postconditions and C is the code that is being executed. Hoare [11] defines the Hoare triple as:

"If the assertion P is true before initiation of a program C, then the assertion Q will be true on its completion"

Listing 3.1 and 3.2 show 2 example programs to show how the Hoare logic can be used. Listing 3.1 shows a simple program that increments b as long as it is smaller or equal to a. In this example, the precondition (P) is set to *true*, because there are no constraints to the program when it is started. Then the code (C) is executed. Finally, the postcondition (Q) checks the behaviour of the program, meaning that b should now be bigger than a. Listing 3.2 shows an example of a program that has no ending. In this program, the precondition (P) is set to *true*, therefore the code (C) can be executed directly, which is a *while* loop that will run infinitely many times. The postcondition (Q) is set to false, meaning that the program cannot complete its execution.

```
P = true
C{
    a = 10;
    b = 0;
    while (b <= a){
        b = 1 + b;
    }
}
Q = b > a
```

Listing 3.1: Example Hoare triple

Separation logic is an extension of Hoare logic, where the idea of spatial conjunction is checked. Spatial conjunction checks using the binary operator $*$ (separation conjunction)

```
P = true
C{
    while (true){
        foo();
    }
}
Q = false
```

LISTING 3.2: Example Hoare triple with no ending

```
P = perm(x, 1);
C{
    x = 10;
}
Q = perm(x, 1);
```

LISTING 3.3: Example separation logic spatial conjunction

that the left and right sides of the operator are true and are in different memory locations of the heap. Therefore, getting permission for a field twice in the memory is impossible. At first, separation logic did not allow concurrent programs at all. Later on, separation logic was adapted to concurrent separation logic [5], which allowed concurrent programs, and verified that two threads could not access the same field simultaneously. Although this verifies that there are no data races, it also disallows multiple threads from reading the value of the field simultaneously.

3.2 Fractional permissions

Fractional permissions [4] is a method using separation logic and fractions, to determine permissions in a program. VerCors uses fractional permission checking to verify the behaviour of programs. In fractional permissions, each field stored on the heap has an associated fraction assigned to it per thread. The total permission for a field is 1 and therefore a thread is allowed to read and modify a field when it holds the complete permission of 1. This prevents 2 threads from accessing a field simultaneously and functions similarly to separation logic. However, in fraction permission checking, multiple threads may read the same variable simultaneously. This is allowed when the permissions for all of the threads hold the 2 conditions:

1. $\forall t \in threads : 0 < P(f) < 1$. Meaning that each thread that wants to read the field has permission between zero and one.
2. $\sum_{t}^{threads} P_t(f) \leq 1$ Meaning that the sum of all the permissions for a field is in total less or equal to one.

Listings 3.3 and 3.4 show an example of a program that uses fractional permissions. In the first listing, the permission for the field x is set to 1, meaning that the thread can read and modify the field. The second listing is the same example however now the thread only has half the permission for the field x.
Furthermore, using the mentioned conditions for fractional permission checking shows that when a thread holds a read permission, it is impossible that another thread has write

```
P = perm(x, 1/2);
C{
    x = 10; //not allowed
}
Q = perm(x, 1/2);
```

LISTING 3.4: Example separation logic spatial conjunction

```
P = perm(x, [T1]);
C{
    x = 10;
}
Q = perm(x, [T1]);
```

LISTING 3.5: Example separation logic spatial conjunction

permission. Assuming we have one thread with write permission and one thread with read permission for the same field, this would mean that the total permission of that field is bigger than one which is not allowed according to the conditions.

3.3 Symbolic permissions

Symbolic permissions [12] also represent if a thread has read or write access to a certain field of the program. Instead of keeping track of fractions, see Section 3.2, it keeps track of a list of all threads that will access the field. When a thread is the only one in the list then it has write access and is allowed to modify the value of the field. When there are multiple threads in the list then these threads have read access to the field and are not allowed to modify the value. Listing 3.5 shows an example of symbolic permission checking. In the example, the precondition holds a permission list for field x. In this list is only *T1*, which is in this case the main thread. Since *T1* is the only thread in the list it has write permission for the field x and thus is allowed to set the value of x to 10.

Using symbolic permissions makes it possible to easily check if a thread has enough permissions for a field, however proving that a thread has only read access, while the thread is the only one in the list is complicated. A workaround would be adding a dummy thread to the list to show that it only has read permission. Listing 3.6 shows an example on how the dummy thread can be used. The example has a permission list for field x. This list has two threads, namely *T1* and *Dummy*. This indicates that there are multiple threads in the list and therefore the threads are only allowed to read. Therefore it is not possible that *T1* can set the value of x to 10.

```
P = perm(x, [T1, Dummy]);
C{
    x = 10; //not allowed
}
Q = perm(x, [T1, Dummy]);
```

LISTING 3.6: Example separation logic spatial conjunction

Chapter 4

VerCors

This chapter explains what VerCors is and how it is internally structured. The complete and up-to-date explanation of how VerCors works can be found on the GitHub of VerCors[1].

4.1 What is VerCors?

VerCors is a static/deductive verifier built and maintained by the Formal Methods and Tools (FMT)[2] research group of the University of Twente. The tool is specialized in concurrent software and validates programs using fractional permission. This is done using formal specifications about how the program should work and what permissions threads have on certain fields, these annotations are called Permission-Based Separation Logic (PBSL). At the moment of writing the paper, VerCors supports the following languages: Java, C, OpenCL, and PVL.

4.2 VerCors internal structure

VerCors is written in Scala [1], which is a programming language that combines Object Oriented Programming (OOP) and Functional programming. This language works well for VerCors, because an abstract syntax tree (AST) can be easily constructed in an OOP language. Also, for transforming the AST, functional programming features can be applied.

VerCors consists of multiple stages to prove that a program is correct, see Figure 4.1. The first stage is the *Parsing* stage, which parses the source language of the program with annotations. This stage translates the program to its internal language (Col) which is a notation for an AST. This stage makes sure that there are no syntactical errors in the code and annotations.

[1] VerCors: `https://github.com/utwente-fmt/vercors`
[2] Formal Methods and Tools group (FMT): `https://www.utwente.nl/en/eemcs/fmt/`

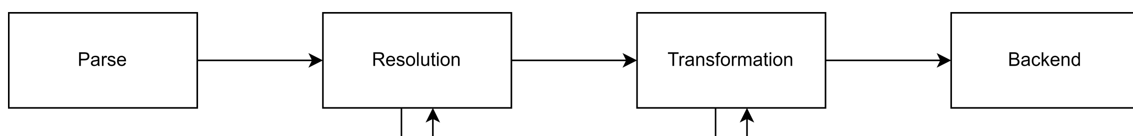

FIGURE 4.1: VerCors stages

The second stage is the *Resolution* stage, which resolves all the names of the AST and loads in classes that are present in the source code. This also loads in classes that are already predefined in the source language, for instance, it is possible to load in a shim of the *Thread* class of the *java.lang* package.

The third stage is the *Transformation* stage, this stage is responsible for repeatedly transforming the AST into nodes that the verifier supports.

The final stage is the *Backend* stage, this stage is used to verify the produced AST and see if it is (in)correct. VerCors uses Viper [17] as the backend to prove the program.

4.3 Syntax

In this section, we will discuss the specification syntax of VerCors. Only the specification syntax that is used in this project (see Section 5) is discussed. Every specification also has an example of how the keywords should be used in VerCors.

Pre- and Postconditions

In VerCors, you define pre- and postconditions using the *requires*- and *ensures* keywords respectively. Additionally, it is also possible to use the *context* keyword if the condition must hold in the pre- and postcondition. In the postcondition, it is also possible to use the result of the function by using the $\backslash result$ keyword and to get the old value of a variable by using the $\backslash old$ keyword.

```
//@ context Perm(this.x, write);
//@ ensures \result == \old(this.x) + 1;
public int increment() { /* ... */ }

//@ ensures false;
public void neverEndingMethod() {
    while(true){ /* ... */ }
}
```

Listing 4.1: Example on how to use the requires and ensures keywords

Permissions

Permissions are defined using the $Perm(o.f, frac)$ function, where the o represents the object and the f represents the field of that object. If the field is inside the object the keyword *this* can be omitted. Two permissions of the same field can be combined using the separation conjunction operator (**), but they cannot be more than one in total. Listing 4.2 shows an example of how the permission syntax can be used, where write permission for the *this.val* field is required as a pre- and postcondition of the *incr* method.

```
class Counter {
  public int val;
  /*@
    context Perm(this.val, write);
  */
  void incr(int n) { /* ... */ }
}
```

Listing 4.2: Example on how to use the syntax for defining permissions

Quantifiers

It is possible to use the $\exists$, $\forall$, and $\forall*$ quantifiers using the $\backslash exists$, $\backslash forall$, and $\backslash forall*$ keywords respectively. The $\backslash exists$ quantifier means that at least one of the conditions should be true. The $\backslash forall$ quantifier means that all the conditions must be true. The $\backslash forall*$ also means that all the conditions must be true, but all conditions are in a separating conjunction, meaning that all the permissions for each field are maximally 1. Quantifiers are often used in combination with arrays to check if a condition holds for all elements of the array. Listing 4.3 shows an example of how the quantifier keywords can be used in a program. First, we check that the *arr* is not null. Then we check if we have write permission for each element in the array. Finally, we check if there exists an element in the array that has a value of bigger than 0.

```
//@ requires arr != null;
//@ requires (\forall* int i ; 0<=i && i<arr.length ; Perm(arr[i], 1));
//@ requires (\exists int i ; 0<=i && i<arr.length ; arr[i]>0);
void foo(int[] arr) { /* ... */ }
```

LISTING 4.3: Example on how to use the syntax for defining quantifiers

Loop invariants

Loop invariants are conditions that must hold before and after each iteration of the loop. This is specified using the *loop_invariant* keyword. Loop invariants must redeclare the permissions, just like function contracts. If they are the same as the function contract, it is possible to use the *context_everywhere* keyword in the function contract. Listing 4.4, shows an example of how loop invariants can be used. The first loop invariant shows that *i* is always lower or equal to *a.length*. The second loop invariant shows that the write permission for *a[i]* is required.

```
public int mult(int[] a, int b) {
  //@ loop_invariant i <= a.length;
  //@ loop_invariant Perm(this.a[i], write);
  for (int i=0; i<a.length; i++) {
    if(a[i] == b) {
        return i;
    }
  }
  return -1;
}
```

LISTING 4.4: Example on how to use the syntax for defining loop invariants

Predicates

Predicates can be seen as a box where permissions and conditions can be put in. This makes it easy to transfer multiple permissions to a function without showing the underlying fields of that object. Thus allowing methods not to see the private fields of other objects and therefore conform to the Object Oriented Programming (OOP) paradigm. Predicates also make it possible to not have a rippling effect where the developer has to write down each individual permission every time they call another method.

To have access to fields, the predicates must be unboxed inside the method. Predicates can only be created inside the methods, but then the thread puts its permission to access the field into a "box". This box can be passed to the next method, where it can be unboxed to receive access to the required fields. In VerCors first, a definition of the predicate must be defined, this is done using the *resource* keyword. Then it is possible to "box" the predicate using the *fold* keyword and "unbox" it using the *unfold* keyword.

Listing 4.5 shows an example of how a predicate can be used in a program. The program has a class *Cell*. The class has a predicate called *state* which receives a parameter *val*. The body of the predicate checks if the write permission for *this.x* is present and if the *val* and *this.x* are the same value. The *foo* method will add *this.x* and *n* together and returns that value. The method has a pre- and postcondition that specify that an instance of the *state* predicate must be present. To read the value of *this.x* the predicate must first be unfolded and later on be folded again to satisfy the postcondition.

```
class Cell {
    //@ resource state(int val) = Perm(this.x, write) ** x == val;

    int x;

    //@ requires state(this.x);
    //@ ensures state(this.x);
    public int foo(int n) {
        //@ unfold state(this.x);
        int result = this.x + n;
        //@ fold state(this.x);
        return result;
    }
}
```

LISTING 4.5: Example on how to use the syntax for defining predicates

Locks

In VerCors it is possible to create a lock invariant, which is a special kind of predicate to determine the properties of a lock. The role of the lock invariant is a condition that should hold when the lock on an object is acquired. The condition can also exist of permissions that can be provided to the thread that locks on the object. A lock invariant is always on a class and should always hold at the end of the constructor of that class (so it is automatically folded). In the constructor, the creating thread should have all the access to the object. If that is not the case then it can never be true later. In VerCors showing that the lock is initialized is by using the keyword *commit*.

The only way to currently use the lock invariant in Java is by using a *synchronized* block. To use the lock we must first verify that the lock is initialized using the keyword *committed*. When a thread is in this block the lock invariant can be assumed of that object (so it is automatically unfolded). At the end of the block, the lock invariant should be reestablished (folding it again) to its original state. If it cannot be folded again, and thus permission gets lost, an error should be thrown, because the next thread that wants to lock the object, will not receive the permissions that the lock invariant promises to give.

Listing 4.6 shows an example of how lock invariants can be used to verify a program. The program consists of a class *Cell*. This class has a lock invariant, that holds write permission of the *this.x* field. In the constructor, the lock invariant will be established using the keyword *commit*. Finally when using the method *increment* it is required that the lock invariant is established. If it is and the thread enters the synchronized block it acquires the lock invariant.

```
//@ lock_invariant Perm(this.x, write);
class Cell  {
    int x;
    public Cell() {
        // Lock invariant is established here!
        //@ commit this;
    }

    //@requires committed(this);
    public void increment () {
        synchronized(this){
            // Assume lock
            x += 2
            // Return lock invariant
        }
    }
}
```

LISTING 4.6: Example on how to use the syntax for defining predicates

Chapter 5

Implementation Design

This chapter will consider the advantages and disadvantages of different options for implementing the prototype. Additionally, we will discuss what design choice for each feature is used in the prototype implementation with a discussion about design mistakes.

5.1 Basic permissions

The internal structure of VerCors has changed significantly, so the prototype that Gijsen created [9] does not work anymore. Additionally, the prototype used symbolic permission checking, see Chapter 3, to check if a thread has enough permission. Since we want to use fractional permissions, the code cannot be reused in this project and needs to be reworked completely.

5.1.1 Storing permissions

Fortunately, Gijsen did introduce different ideas for tracking and storing permissions.

ThreadLocal permission accounting

First, Gijsen introduced the *ThreadLocal* permission accounting. This accounting stores all the permissions which a thread has in the *ThreadLocal* (See Listing 5.1) storage and thus is only available to the owner of the thread's local storage.

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

Listing 5.1: Storing permission in the *ThreadLocal*

This approach has the following advantages and disadvantages:

On one hand, using this approach it is very easy to check if the thread has enough permission to access a field. On the other hand, exchanging permissions between threads is not trivial because the threads only know how much permission they have themselves, and not what the others have. Additionally, it is not possible to store permissions for primitive types, because primitive types are passed by value in Java.

Global permission accounting

The second approach is to have a global static map (Ledger), where the permissions are stored into a map based on the object, field, and the id of the thread, see Listing 5.2. By using a global map it is relatively easy to transfer the permission between threads. This

approach is close to a symbolic permission checking approach because every thread knows the permissions of all threads. Gijsen decided that fractions are not necessary at all and therefore the fraction can be removed from this map to save memory. Although this is true for basic permissions, this removes a core concept of VerCors, by not using fractional permission syntax and thus makes it harder to support different features other than basic permissions, for instance, predicates and quantifiers. This would also require the developer to write different annotations for static verification and runtime verification, which is not desired.

```
Map<ThreadId, Map<Object, Map<Field, Fraction>>>
```

Listing 5.2: Storing permission in global map

Per-field permission accounting

The final approach Gijsen proposed was the per-field permission checking. For this, each class holds a map that can be used to check the permission for the field of the class, see Listing 5.3. The advantage of this approach is that you can check the permission of a field relatively easily because the class is already known. A disadvantage is that the definition of the class changes, and thus when the program makes use of reflection or serialization of objects in the program, the permission accounting is exposed.

```
class Example{
    Map<ThreadId, Map<Field, Fraction>>>
}
```

Listing 5.3: Storing permission in per-field map

Implementation Choice & Discussion

At the beginning of writing the prototype, we decided to implement the per-field permission storage. At first, this was a good decision because it was relatively easy to implement the storage for basic permissions for fields, making use of internal numbers that refer to the fields of an object. However, this implementation also had its drawbacks namely:

- Every class in the prototype should create *HashMaps* to store permissions. This results in creating a large number of *HashMaps* when a significant number of objects are created. Due to the creation of *HashMaps* the implementation is scaling poorly in memory
- Accessing the permissions got significantly complicated for quantifiers and predicates and thus implementing these features got significantly harder
- Per-field permission storage is unable to determine permissions for function parameters. This is because a parameter in a function can be an *Object* or array which does not have to point to a field but can be a locally created object in another function. Due to this, we do not know where the permission should be checked and thus the permission cannot be checked entirely. This is the most important reason that per-field permission storage cannot be used in runtime assertion checking because it has cases that it simply cannot support.

Due to the mentioned reasons we decided to change the prototype to make use of the global permission storage (Ledger). This created multiple benefits:

- All objects that are created can be put into a single *HashMap* and thus save memory as much as possible
- Permissions can be requested or altered from anywhere in the code
- Predicates and join tokens can also be stored in the global storage (Ledger), making it easier to add different functionality to the code in future implementations

To store permissions we also need a *Fraction* object to store fractional permissions. Instead of implementing the *Fraction* class ourselves, we use the *Fraction* class of the *org.apache.commons.math3*[1] package. The *Fraction* class supports mathematical operations between fractions. Additionally, it also provides methods to easily compare fractions with each other.

5.1.2 Checking permissions

Gijsen discussed when permissions should be checked. According to Gijsen, permissions should be checked above each line that is referencing to a field to ensure that all expressions have the correct permissions. They mention two scenarios when permissions should be checked automatically.

The first scenario is when a field of an object is dereferenced and thus is being used in an expression. Then the permission for that field should be a read permission, see Listing 5.4.

```
// need read access for counter.count
int c = counter.count

//need read access for program.counter
//need read access for program.counter.count
int d = program.counter.count
```

Listing 5.4: Assignments that requires read permission

The second scenario is when a dereference happens on the left side of an assignment statement. Then the thread needs write permission to access that field, see Listing 5.5.

```
// need write access for counter.count
counter.count = 4
```

Listing 5.5: Assignment that requires write permission

These two approaches are correct but create a lot of unnecessary permission checks. For this, we can introduce permission checks based on blocks, where assignments and operations can be grouped so that the permissions have to be checked only once per block instead of for each statement. A block ends when a change on the heap occurs, for which there are two scenarios. The first scenario is when a method invocation occurs because we do not know what happens in that function so we must assume that the heap can change in that function. The second scenario is when an assignment to a non-primitive variable/field occurs. In this case, the heap will change and the permissions need to be rechecked. Listing 5.6 shows an example of how the permissions for blocks. In this example we create two different *Counter* objects, therefore the heap changes and thus both statements need to be in a separate block. Block three has two statements reading and altering the value of

[1] Fraction class: https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/fraction/Fraction.html

primitive fields, therefore all the permissions that are required in this block can be grouped. Block four changes a non-primitive variable by assigning *counterTwo* to *counterOne* and thus the permissions need to be rechecked in the next block. Block five creates two new integers using the *counterOne.count* variable, thus the permission for *counterOne.count* only needs to be checked once. Finally, a statement with a method invocation in block six needs to be in a separate block since we do not know if the *increment* method changes the heap or not, so we must assume that it does.

```
    public void main() {
        //block 1
        Counter counterOne = new Counter();
        //block 2
        Counter counterTwo = new Counter();
        //block 3
        int a = counterOne.count;
        counterOne.count = a * counterTwo;
        //block 4 (assignment to non-primitive variable)
        counterOne = counterTwo;

        //block 5
        int b = counterOne.count * 2;
        int c = counterOne.count + 1;
        //block 6 (method invocation, so new block)
        counterTwo.increment(b + c);
    }
```

LISTING 5.6: Permission checking per block

Gijsen also mentioned that permissions checks on the fields of a class are not necessary inside the constructor of that class, because the permissions for these fields are initialized in the constructor and thus no other thread could have accessed the field yet. Permissions of fields of an external object that are used in a constructor should be checked, see Listing 5.7. In the listing, permission checking is not required for the *this.val* field but is required for the *e.val* field.

```
public class Example{
    int val;
    public Example(Example e) {
        //no write permission required for this.val
        this.val = 0;
        //write permission required for the e.val field
        e.val = e.val + 1;
    }
}
```

LISTING 5.7: Permission checking inside the constructor

Implementation Choice & Discussion

In the prototype, we make use of the permission per block checking method. This removes redundant permission checks and thus saves performance. Checking permissions can easily be done by looking up the permission for the object in the *HashMap*. However, the prototype should also support the verification, shown in Listing 5.8, where the permission for *this.i* is checked twice for a permission of $\frac{1}{2}$ permission.

```
//@ requires Perm(this.i, 1\2) ** Perm(this.write, 1\2)
// is equivalent to:
//@ requires Perm(this.i, 1)
```

LISTING 5.8: Permission should be added up for *this.write*

This is a valid check, but due to the ** operator, the permissions need to be added up and thus the permission that needs to be checked is actually $\frac{1}{1}$. To do this, we keep track of the permission objects and add all the permissions together if they are referring to the same object. After we checked the complete condition, we additionally check that the permission of the tracked objects is not higher than the permission the thread has for that object.

5.1.3 Exchanging permission

Permission can only be exchanged when threads are forking, joining, or acquiring a lock.

Starting a thread

Gijsen used a *preFork* predicate to indicate how the permission should be transferred between threads. The thread that starts another thread first needs to check if it has enough permission to transfer the permission. If it has enough permission, it can create the *preFork* predicate and hand over the permission to the other thread. Additionally, Gijsen made use of start tokens to verify that the thread cannot be started more than once. However, this is unnecessary for the prototype because the Java Runtime Environment (JRE) will throw an *Exception* when a thread is started more than once.

Another option to know the permissions needed to be transferred to the new thread, is using the pre-conditions of the *run* method. All newly created threads have the *run* method implemented and thus can specify in annotations what permissions are required to start the thread. Before transferring the permissions, the starting thread needs to check that it has enough permission.

Joining a thread

A thread can be joined more than once and can be joined by multiple threads. To decide how much permission each thread will get from the joined thread, join tokens are used. The sum of all join tokens is maximally 1, when a thread is joining a thread it will receive:

$$join_token \times perm + old_perm$$

For this, it is possible to add a special annotation to VerCors called *postJoin(frac)*. This annotation must then be used in the runtime verifier to show how much of the permission is transferred between threads. The special annotation can then be used on the post-condition of the *run* method. The post-condition of the *run* method indicates what the final permissions of the thread are.

Gijsen made use of symbolic permission checking. This enabled storing the join tokens as a list of threads that will join the joining thread. This implementation could be used in our prototype as well and is relatively easy to implement, however, this will limit the prototype that every joining thread gets the same permission of the joined thread. Therefore this limits the functionality of the prototype and limits implementing other features as well.

Implementation Choice & Discussion

In the prototype, we use the specifications of the *run* method for starting and joining the threads. The pre-condition of the *run* method specifies how much permission the thread wants to acquire. The starting thread will have to check if it has enough permission to transfer the permissions to the newly started thread. If that is the case the started thread can assign the required permissions to itself.

Joining makes use of the post-condition of the *run* method and a special annotation for a join token using the following format: $o.postJoin(frac)$. The o indicates which thread object the post join token applies to and the $frac$ indicates the size of the join token that is used in the joining operation. The join token for each thread is stored in the global storage and is used to check if no more than the allowed join token is requested. After checking if there is enough join token left for the thread, the permissions will be transferred to the joining thread by multiplying the post-condition with the join token.

5.1.4 Locking

As explained in Section 4.3 locking is a special predicate in VerCors. Predicates will be explained in Section 5.5, but work similarly. In the prototype, we can remove the permissions that are required for the lock invariant when the object is initialized. When a thread enters the synchronized block it will receive the permission that is defined in the lock invariant. At the end of the synchronized block, the lock invariant should be re-established, if this is not possible then permission for a field has been lost or the program reached an undesired state. The advantage of this approach is that there are no extra objects created to store the permissions of the lock invariant.

Another option would be to store the predicate for the lock invariant in the object itself and when the thread acquires the predicate, it first unfolds the predicate and receives the permissions. Finally, at the end of the synchronized block, it needs to set the lock invariant of the object back to its original value. The advantage of this approach is that there is only one lock invariant and thus when a thread acquires the lock invariant we show that there are no other threads with the lock invariant predicate. The disadvantage is that an extra lock invariant object needs to be created and thus requires additional memory usage.

Implementation Choice & Design

Locking is implemented by removing the required permissions of the thread that creates the object holding a lock invariant. When a thread enters the *synchronized* block it will receive the permissions of the lock invariant. Finally, at the end of the *synchronized* block it will need to reestablish the lock invariant.

5.2 Array permission

Next to basic permission checking, array permission checking should also be supported in the prototype. This is possible by creating another field in the global storage (Ledger) for arrays specifically, where we also store the location of the array in the *Map*, see Listing 5.9. By doing this, it still works similarly to normal permission storage.
An advantage of this approach is that its permissions can easily be looked up in the *Map*. A disadvantage of this approach is that a significant number of *Maps* need to be created

```
Map<ThreadId, Map<Object, Map<Integer, Fraction>>>
```

Listing 5.9: Storing permission for array in another map

and thus be memory inefficient.

Another option is to store the permissions in the same map we use for basic permission checking but making use of *Object[]*, where we can store the array and its location and then refer to a *Fraction*, see Listing 5.10. In the listing, we create an integer array with two elements. The prototype should create a *Object[]* for each element in the array. The content of the *Object[]* is the array (a) and the location of the element in the array. An advantage of this approach is that we can reuse the same map for basic permission checking for the array permission checking. A disadvantage is that extra objects have to be created to check the permission for the arrays.

```
int[] a = new int[2];
//store permission Object[]{a, 0}
//store permission Object[]{a, 1}
```

Listing 5.10: Storing permission for array in *Object[]*

Implementation Choice & Design

Since we started with implementing permission storage per field, we also created an additional *HashMaps* that also kept track of the location in the array. However as we mentioned before, we do not always know where the array is coming from in a function when it is passed as a parameter. Therefore, we cannot use this approach for permission checking on positions in the array.

The solution is that we can use the same global *HashMap* that we are using for normal permission. We can store the location of the array in an *Object[]* and this can be stored in the *HashMap* as well. Unfortunately, *HashMaps* and *List* make use of the *equals* method on objects and thus the equals method on two *Object[]* will return false even though their content is the same. This is because the arrays both refer to different positions in the memory and thus are not equal, see Listing 5.11. Fortunately, it is possible to make a wrapper class (*DataObject*) which stores the *Object[]* but uses the *Arrays.equals* method to determine if two *Object[]* are equivalent, see Section A.2.1 for a more detailed explanation of how the *DataObject* works.

```
Object[] arr1 = new Object[]{1, 2};
Object[] arr2 = new Object[]{1, 2};
arr1.equals(arr2);
//returns false since they refer to different object even though their
   content is the same
```

Listing 5.11: Comparing 2 *Object[]* with similar content

5.3 Quantifiers

In VerCors it is possible to use the $\exists$, $\forall$, and $\forall*$ quantifiers in the pre- and postconditions and lock/loop invariants using the $\backslash exists$, $\backslash forall$, and $\backslash forall*$ keywords respectively.

For this, we can provide two options on how they can be implemented in the prototype.

The first option is relatively simple, where we create a loop that checks the condition of the quantifiers. Although this option is easy to implement, the generated code looks less clean, because of the number of for-loops in it.

The second option is to create a function that creates a for-loop to check the conditions of the quantifier. This makes the code look very clean but is relatively difficult to implement. Since the quantifiers might need variables that are defined earlier, this needs to be bubbled down into other quantifiers if there are multiple quantifiers nested, so Listing 5.12. Using the first option does not require bubbling down of the parameters because all the parameters are available in the function itself.

```
/*@ requires (\forall* int i ; 0<=i && i<arr.length ;
    (\forall* int j; 0<=j && j<i; Perm(arr[i], 1) ** arr[i] <= arr[j]
        ** arr[i] < this.val));
*/
public void main(){
    assert(quantifier_1(arr, this.val));
}

//the inputs of the variables that are needed needs to be bubbled down
    to this function
public boolean quantifier_1(int[] arr, int val) {
    for(int i=0; i < arr.length; i++){
        if(!quantifier_2(arr, val, i)){
            return false;
        }
    }
    return true;
}

//the inputs of the variables that are needed needs to be bubbled down
    to this function
public boolean quantifier_1(int[] arr, int val, int i) {
    for(int i=0; i < arr.length; i++){
        //check permission for arr[i]
        if(arr[i] >= arr[j]){
            return false;
        };
        if(arr[i] >= val) {
            return false;
        }
    }
    return true;
}
```

LISTING 5.12: Example quantifier bubbled down problem

Implementation Choice & Design

At first, we implemented the option for generating new functions for each quantifier. However, as mentioned before the bubbling down of the parameters became significantly more

difficult to implement. Thus we changed the implementation to generating for-loops for each quantifier in the code. First, the bounds of the will be looked up, if there are no minimum or maximum bounds defined we make use of the *Integer.MinValue* and *Integer.MaxValue* respectively. Second, a check is generated that the loop condition holds, if this is not the case the code will proceed to the next iteration. Finally, the loop check is performed and returns true or false based on the quantifier.

Additionally, the $\backslash forall*$ quantifier also needs to check that there are no duplicate objects in the array the quantifier is checking permissions from. This behaviour is not allowed in the static version of VerCors and thus will also not be allowed in the runtime assertion checking, see Listing 5.13 for an implementation that should not be allowed by VerCors. In the example an array with *Dummy* objects is created, where the first and second elements point to the same *Dummy* object *a*. The pre-condition of the *test* method checks if all elements have $\frac{1}{2}$ permission. It is not allowed that two elements are tested twice on permission in a $\backslash forall*$ quantifier and thus should return an error if it does occur.

```
class Dummy{
    int a;
}

public void main() {
    Dummy a = new Dummy();
    Dummy[] b = new Dummy[]{a, a};
    test(b);
}

//@ requires (\forall* int i ; 0<=i && i<arr.length; Perm(arr[i].a,
    1\2);
public void test(Dummy[] arr) { /*...*/ }
```

LISTING 5.13: Example that shows an implementation that is not allowed by VerCors

5.4 Loop invariants

Loop invariants are needed to check at the beginning and end of each loop iteration. This can be done by adding assertions to the beginning and ending(s) of a method. This is similar to the pre- and postconditions of a loop iteration. When a *continue*- keyword occurs, the loop invariant is also checked before this keyword. This is required because they will stop a loop iteration and thus the loop invariant needs to be checked since it is the end of the loop iteration.

Implementation Choice & Design

Loop invariants are implemented as described above in the prototype and no problems occurred when implementing it.

5.5 Predicates

Predicates are more difficult than the other new features since they work as a box (see Section 4.3). The minimal requirements are that we can fold and unfold the predicate and

check if the thread has the predicate in its possession. Two different options are considered for implementing predicates into the prototype.

Class per predicate

The first option is to use a separate class for each predicate. This class has a generated fold and unfold method, that can remove the permission and return the permission to the thread. This class can also be used statically to store all the predicates that the thread currently has of this predicate type. See Listing 5.14 for an example of how this option for predicates works.

```
public class Counter{
    int count;
    //@resource state(int val) = Perm(this.count, write) ** this.count
        == val;
}

public class RuntimePredicateState{
    Map<Long, List<RuntimePredicateState>> predicates;

    public void unfold(Counter count, int val){
        //adds permission to the thread and removes predicates from
            thread
    }

    public void fold(Counter count, int val){
        //check predicate condition
        //remove permissions to the thread and add predicate to the
            list
    }

    public boolean hasPredicate(Counter count, int val){
        //returns true if the thread has the predicate
    }
}
```

Listing 5.14: Predicate creating class for each predicate

The fold method will remove the permissions of the thread that are required by the predicate. First it needs to check if the thread has enough permissions for the fields. After which it will remove the permissions of the thread that are required by the predicate. Then it will create an instance of the predicate class and store it in the predicate store. To know the properties of the predicate, we also need to store the corresponding class and the parameters of the predicate.

The unfold method works the other way around than the fold method. First, the method will check if the thread has the predicate in its possession, and then it will add the permissions to the thread. Finally it will remove the predicate from the store.

The advantage of this approach is that the permissions only change when a thread accesses the predicate class. The disadvantage of this approach is that a lot of boilerplate code needs to be generated to create the predicate functionality. Additionally, when a predicate is created, a new instance of the predicate is created as well, this uses a significant amount of memory.

Predicate in Object[]

The second option is to store the predicate details in an *Object[]* and store it in a map globally. An example of this approach is shown in Listing 5.15. In this example, the predicate is folded and unfolded using the *Object[]*. In the *Object[]* is the class that is associated with the predicate, in this case *this*. Then the parameters of the predicate (*count*) and finally the name of the predicate (*"state"*). The advantage of this approach is that the predicate can easily be folded and unfolded and easily be checked that the thread has the predicate in possession. The downside of this approach is that for each predicate that is folded an extra object is created with the details of the predicate, this uses more memory. Another downside of this approach is that the transfer of permissions happens in the method with the fold/unfold statements. Thus making the code of that method long and difficult to understand. However, this is generated code so it should not be read by the developer of the program and only by the developer of the runtime assertion checker.

```
Map<Long, List<Object[]>> predicatestore

public class Counter{
    int count;
    //resource state(int val) = Perm(this.count, write) ** this.count
       == val;

    //@ requires state(count);
    //@ ensures state(count);
    public void increment(){
        assert(predicatestore.contains(new Object[]{this, count, "state
           "}));
        //unfold predicate permissions to thread
        predicatestore.remove(new Object[]{this, count, "state"});
        count += 1;
        //fold predicate permissions from thread
        predicatestore.add(new Object[]{this, count, "state"});
    }
}
```

Listing 5.15: Predicate store predicate details in *Object[]*

Implementation Choice & Desgin

The initial idea for predicates was to create a different class for each predicate that is defined. These classes will have a fold and an unfold method so that the permission can be removed and added inside of this generated class. Additionally, this class could also be the store for the predicates to check if the thread has the predicate. This idea worked but was not ideal, because of the following reasons:

- When a significant number of predicates are stated in the code also a significant number of classes are generated. This resulted in being unclear of what was happening in the code and made the code look poorly
- Each of the generated classes is coupled to a class that the resource is defined in, so there is no general store that can handle all the predicates at once
- Each generated predicate class has a predicate store to store all the predicates for each thread, thus generating more *HashMaps* for the predicate objects

Due to these reasons, we also decided to rework the predicates and store them in the global storage (Ledger) in a different *HashMap*, which has the thread id as the key and a list of all the predicates that the thread has as the value. The object that is used to refer to the predicate is, just like array permissions, an *Object[]* wrapped in a *DataObject*. In the *Object[]*, we keep track of the arguments of the predicate, the object the predicate is associated with, and the name of the predicate, see Listing 5.16. When a predicate is being folded the condition of the predicate is checked, then the permissions are removed from the thread beforehand and finally, the predicate object is created. Unfolding a predicate first checks if the thread has the predicate, then it will remove the predicate from the storage and finally will return the permission to the thread.

```
//@ resource state(int val) = Perm(this.count, write) ** this.count ==
    val
DataObject.create(new Object[]{this.count, this, "state"});
```

LISTING 5.16: Predicate transformation to *DataObject* holding predicate data

5.6 Prototype Implementation Overview

This section will discuss the generated global storage (*Ledger*) and how it works. Additionally, a brief overview of all classes that are used throughout the prototype.

5.6.1 Ledger

Figure 5.1 shows the generated *Ledger*. The *Ledger* has a store for permissions, predicates, and join tokens. The permissions and predicates make use of the *DataObject* class. The *DataObject* class is a wrapper class for the *Object[]*. Because comparing two different *Object[]* arrays always returns false even if the content is the same, we need to define a new *equals* method. The *equals* method of the *DataObject* makes use of the *Array.equals* method to compare the content of the arrays as well. The *DataObject* makes it possible for array permissions and predicate information can be stored in an *Object[]* so that they can be globally stored with other permissions and predicates.

5.6.2 Class Descriptions

Figure 5.2 shows the different classes that are used to make the prototype work. We will briefly explain how each class works and what the responsibilities are of each class. First, we have all the helper classes, which are defined on the right side of the figure.

- **Ledgerhelper**: responsible for interacting the the global storage (Ledger).
- **Util**: provides utility methods which can be used throughout multiple different stages
- **CreateConstructor**: restores the removed constructor in the *Resolution* stage. The *CreateConstructor* class is only used by the *RefactorGeneratedCode* indicated by the green line.
- **FindBoundsQuantifier**: searches through the quantifier and defines bounds for each quantifier
- **PermissionData**: data object that provides extra information for rewriting contracts and transferring permissions

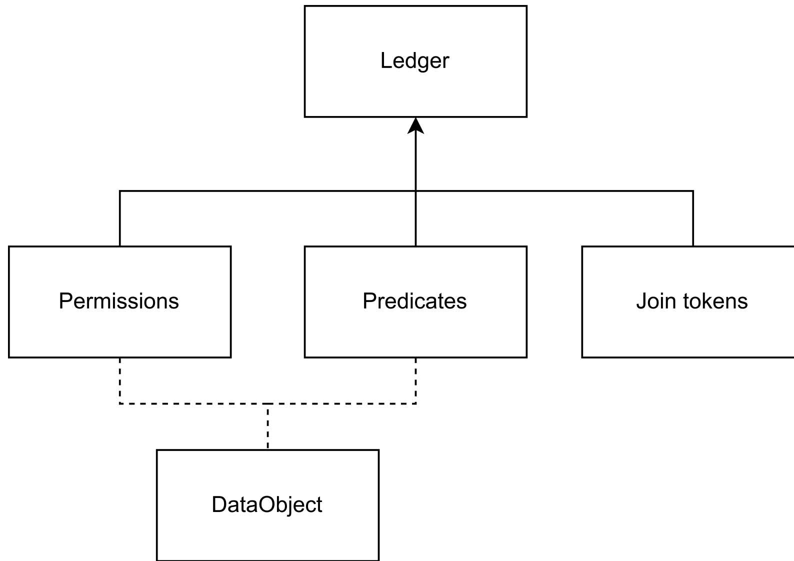

FIGURE 5.1: Ledger overview

- **AbstracQuantifier**: an abstract class that transforms a quantifier to a for-loop, so that only the loop body have to be implemented by the *RewriteContractExpr* and *TransferPermissionRewriter* rewriters
- **RewriteContractExpr**: rewrites a contract expression to multiple assert statements. So it is responsible for rewriting contracts of pre- and post-conditions, lock invariants, loop invariant, predicates, forking and joining. Stage classes that make use of the *RewriteContractExpr* class are indicated with an orange line in the figure.
- **TransferPermissionRewriter**: rewrites a contract expression so that is possible to transfer permission between two threads. This is used in forking, joining, predicate and lock invariants. Stage classes that make use of the *TransferPermissionRewriter* are indicated with a blue line.

Second, we have the stage classes, which are stages to transform the source code into the newly generated code. The stages are displayed on the right side of the figure.

1. **CreateLedger**: creates the global storage (*Ledger*) for storing permissions, predicates and join tokens.
2. **RemoveSelfLoops**: removes the self loop for each object to the *Object* class. Since each object in java extends the *Object* class, VerCors automatically adds the *Object* class as a dependency for each class. But in the generated code there can only be one extension therefore, the *Object* reference must be removed
3. **RefactorGeneratedCode**: responsible for recreating the constructor if the class is a class and not an interface
4. **AddPermissionOnCreate**: add permissions to the *Ledger* when an object is instantiated. Additionally, it initializes the permission when a new array is created.
5. **CreatePredicateFoldUnfold**: responsible for creating the fold and unfold functionality. It needs to create a check if the condition of the predicate is met. If so it

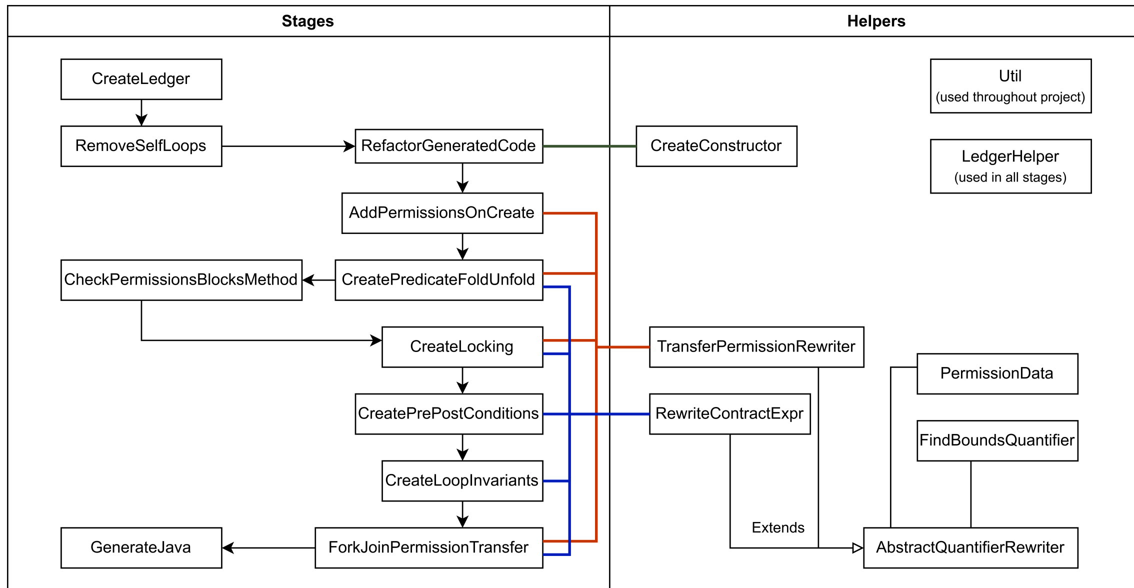

FIGURE 5.2: Prototype stages

can remove the permissions of the thread and create the predicate object and store it in the *Ledger*. When unfolding it first needs to check if the thread has a predicate of this type and can then remove it and return the permission to the thread.

6. **CreateLocking**: responsible for removing the permissions of the lock invariant when the class is instantiated in the constructor. Additionally when entering a synchronized block the permissions of the lock invariant are transferred to the thread. At the end of the synchronized block, the lock invariant is reestablished.

7. **CreatePrePostConditions**: transforms pre- and post-conditions to assert statements at the beginning and end of each method.

8. **CreateLoopInvariants**: transforms loop invariants to assert statements at the beginning and end of each loop iteration.

9. **ForkJoinPermissionTransfer**: responsible for transferring permission between threads when a fork or join occurs.

10. **GenerateJava**: the final stage, which tries to transform the generated code to code that can be executed directly. So it removes unnecessary classes and functions as much as possible, add imports and creates a package name.

Appendix A, provides a more detailed description of each class and how the classes work.

Chapter 6

Testing & Results

An important part of the project is testing and validating that the implementation of the prototype is correct. In the current version of VerCors, there are test cases that can be run to test the implementation of VerCors. Ultimately this would have been nice to use on the runtime verification as well. Unfortunately, the code that is produced by the prototype cannot be executed immediately. For instance, a dummy implementation of the *Thread* class is in the produced output, which at the moment of writing this report is not possible to remove from the produced output. Due to the small errors in the produced output, the produced code first needs to be manually changed to immediately execute it. Due to this, it is not possible to automatically test the implementation of the prototype.

In this chapter we provide the test cases that were used to test the implementation of the prototype. For each test, we first describe the test that should not throw an error when the produced output is executed. All following test descriptions will describe how the succeeding test can be changed slightly such that the test fails with the correct error. All the code of the test programs and the changes to the program so that the program fails can be found in Appendix B. To run the prototype we execute the command in Listing 6.1. The command shows that we use the VerCors program, and want to use the runtime plugin. Additionally, we need to provide a *runtime-output*, which is a file location to store the generated code by the prototype. Finally, we also need to provide an input file that needs to be validated, since there is always an input file required we do not have to insert a flag for it. After that, the generated code can be executed like a normal Java file, and assertions are enabled using the *-ea* flag.

```
vercors -p --lang java --runtime --runtime-output {output} {input}
```

Listing 6.1: Command running the prototype

The test coverage of the test cases on the prototype is 89% method- and 92% line coverage. These are gathered using the IntelliJ coverage runner [1]. These values are good and mean that most of the code is tested. The parts that are not covered by the prototype are most of the time default cases in a scala match-case statement, to not receive warnings when compiling even though we know that the prototype will never reach that line. Additionally, each stage class needs to overwrite the *RewriteBuilder* class, which requires the *desc* method to be overridden. However, the function is never used during the test coverage,

[1] IntelliJ Coverage runner: https://www.jetbrains.com/help/idea/running-test-with-coverage.html#coverage-run-configurations

thus lowering the method coverage. Finally, the method coverage of the *util* package is 82%, therefore lowering the total method coverage as well.

Another important metric is how long the runtime verification prototype took to transform and execute the tests. Table 6.1 shows the execution time for static verification and the time it took for the runtime prototype to transform and execute the program. On average the static verification took 14.923 seconds and the runtime verification took 5.039 seconds. Although runtime verification is faster, static verification guarantees that the code's behaviour is correct. Therefore runtime and static verification cannot be compared to each other. Additionally, the test case programs are relatively small and will complete their execution. If the program does not complete its execution, and therefore the program has an infinite loop, the runtime verification will also run infinitely. Static verification tools would terminate and prove that the infinite program works as intended and therefore would be faster than runtime verification in those cases. Combining runtime and static verification in verifying the behaviour is a suitable option because runtime verification can be used to get a fast indication of the program's behaviour and static verification would completely verify that the program is correct.

Test case	**Static**	**Runtime transformation**	**Runtime execution**	**Runtime total**
Shared Buffer	13.210	5.116	0.025	5.141
Shared Buffer arrays	17.443	4.854	0.022	4.876
Shared Buffer duplicate objects	14.682	4.012	0.022	4.034
Quantifiers	16.779	5.941	0.021	5.962
Loop invariant	14.412	5.422	0.020	5.442
Lock invariant	11.913	4.193	0.023	4.216
Predicates	16.023	5.583	0.022	5.605

Table 6.1: Test results on duration in seconds

6.1 Test case Shared Buffer from Gijsen

Gijsen [9] defined a test case to test their prototype for checking permissions on fields. Since checking permissions on fields is also one of the requirements of our prototype, we will test our program using the same test as Gijsen used. The test program Gijsen created was called the *Shared Buffer*. Figure 6.1 shows how the threads in the program are related to each other. The program simulates the following behaviour:

1. The *Main* thread will create the *Source* and *Sink* threads
2. When the *Source* thread is started it acquires permission for the buffer and changes its value
3. The *Sink* and *Main* thread will join the main thread, each receiving half of the permission of the buffer of the *Source* thread
4. The *Sink* can perform operations and use the buffer as a read value
5. The *Sink* thread is eventually done executing and the *Main* thread joins the *Sink* thread and thus receives the other half of the permission of the buffer

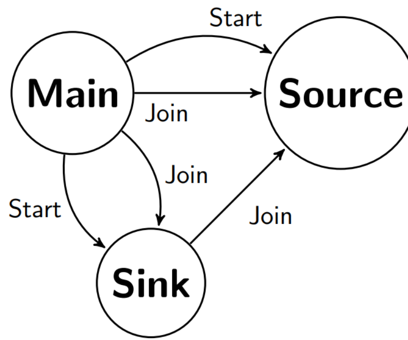

FIGURE 6.1: Diagram of the threads of the reviving code of Gijsen[9]

6. The *Main* thread now holds complete permission for the buffer and thus is allowed to modify the value of the buffer

To let the test case fail we change it so that the *Sink* thread tries to change the value of the buffer. This is not allowed since it only has half of the permission for the buffer and thus only has read permission.

6.1.1 Results

When executing the transformed program *Shared Buffer* the base case does not throw any assertion error, meaning that the test passed and thus the program executed successfully. The failing version of the *Shared Buffer* program does throw an error, that there is not enough permission for the *Sink* thread to access the buffer and points to the line in the program that fails.

6.2 Test case Shared Buffer for arrays

The *Shared Buffer* program used for testing the prototype for basic permission checking can also be used for checking permissions for arrays. In this version of the *Shared Buffer*, we change the buffer to be an array. Thus it simulates the same behaviour as mentioned in Section 6.1.

There are multiple scenarios in which the permissions of arrays should fail. We changed the original program slightly for multiple scenarios, such that the implementation should fail in each scenario.

1. The *Sink* will try to modify a position in the array
2. Having 2 identical Objects in the array should throw an error

6.2.1 Results

The succeeding case of the *Shared Buffer* program for arrays, does not return any errors when it is executed. Thus the program conforms to the specifications that were provided.

The first scenario of the failing cases of the *Shared Buffer* program for arrays does throw an error if the *Sink* thread tries to modify the array. The second scenario of the failing cases also throws an error if the permission is checked for 2 identical objects in the array. The program throws an error inside the quantifier, which checks if the permission location is not checked twice in a quantifier. The errors also point to lines in the program that are failing.

6.3 Quantifiers test case

The quantifiers test case is a simple program that represents different types of quantifiers that can be created in VerCors. Not all of the quantifiers are tested, for instance, quantifiers without bounds are not tested since it would take too long for the program. Additionally, quantifiers that make use of syntactic sugar are not supported by the prototype and thus cannot be tested. By changing the program slightly for each quantifier, we can test if the quantifier implementation works. We do this by changing the operators in the quantifier statements, for instance, changing the < operator to >= operator. When changing the operators to be the complete opposite of each other, the program should always return an error.

6.3.1 Results

Running the succeeding test case does not create any errors. The failing test cases all failed, so that means that the prototype works as intended. Each quantifier that was changed throws an error that the quantifier does not hold and points to the line in the program that fails.

6.4 Loop Invariant test case

The loop invariant test case is the same simple program that is used for testing quantifiers, except that there are no pre- and post-conditions for the method, but only loop invariants for the for-loop. This results that we can test the loop invariant standalone and thus is not dependent on the pre- and post-conditions test case for quantifiers. Similarly to the quantifiers test case, we change the loop invariant contract slightly by inverting the operators, such that it fails.

6.4.1 Results

The succeeding loop invariant test case does not create any errors when executing the program. The slightly altered versions of the program, such that the loop invariant is not true, do return errors about the loop invariant not being correct. Similarly to the other test cases, the errors show that the invariant is not correct and point to the loop invariant that fails.

6.5 Lock Invariant test case

The lock invariant test case is a simple program that has a simple lock invariant on a field. The test case has a synchronized block on the object and alters the value of the field in the synchronized block. By altering the program, such that the field is accessed outside the synchronized block should return an error when executing the program.

6.5.1 Results

When executing the succeeding test case, there are no errors returned. However, when executing the failing test case an error will be thrown that there is not enough permission for the field that is being accessed and points to the line that fails in the program.

6.6 Predicates test case

The predicate test case is an example program shown in the wiki of the GitHub page of VerCors. This is a good test since it incorporates all the important aspects of predicates, where folding, unfolding and checking if a predicate exists are present for different objects. It has a *Main* class that creates some counters and folds 2 predicates, thus removing the permissions for the *count* field. Then it will call the *foo* method to test if the predicate can be transferred from one method to another method. In the *increment* method, the predicate gets unfolded, so that the *count* value can be changed. To test the implementation of predicates in the prototype we can use the following scenarios:

1. Accessing a field before unfolding the predicate should return a not enough permission error

2. Removing the initial folding should return in a predicate not present error

6.6.1 Results

The succeeding predicate test case does not throw any errors while executing the program. The first failing scenario of the program returns as expected an error that the thread that is accessing the field does not have enough permission. The second failing scenario of the program throws an error that the predicate is not present and points to the pre-condition of the method that requires the predicate.

Chapter 7

Related work

In this chapter, we will discuss related works to VerCors and runtime verification on (concurrent) programs. First, we will discuss tools that can already verify concurrent software using static verification. Followed by related works on runtime verification for sequential programs. Finally, we will discuss other related works that combine runtime verification and concurrent software.

7.1 Runtime verification for sequential programs

In this section, we will discuss other runtime verification software to verify the behaviour of programs. The main difference between the other runtime verifiers and the prototype is that they are built for one specific language, for instance, OpenJML is made for Java and E-ACSL is made for C. These programs can simply use the AST of the program, in VerCors the AST is first converted to an internal AST named Col. Therefore transforming the AST to an executable file in the original language is significantly more difficult in the prototype because essential information may be removed from the AST since it is not necessary for static verification. VerCors does transform each AST to Col to support multiple languages with the same tool. However, in this particular use case of runtime checking, it would be counterintuitive to first change the language to Col and then convert it back to the original language.

OpenJML

OpenJML [19] is a tool that uses the Java Modeling Language (JML) to prove the behaviour of sequential Java programs, using pre- and postconditions, ghost code, and assertions. This can be done statically, but also using Runtime Assertion Checking (RAC). When using RAC they change the pre- and postconditions, ghost code, and assertions, to Java assertions. These assertions will then be executed while the program is running on variables and inputs that occur in the program, and thus not check all the possible inputs as in static verification.

OpenJML RAC works similarly to the created prototype for VerCors. Both the OpenJML RAC and prototype change the program such that the behaviour of the program can be verified during execution. However, the OpenJML RAC uses a library to provide functionality to the runtime checker. This has the advantage that the standard code does not have to be generated each time the verification tool is being executed. A disadvantage is that the library's functionality needs to be accessed by the verification tool, so some mechanism

should be built to automatically generate nodes so that the tool can use the library's functionalities. OpenJML RAC can also compile the Java class to an executable, the prototype does not have this functionality yet, but is a feature that should be implemented in future developments.

7.1.1 Frama-C

Frama-C [7] is a tool that can be used to do static verification of sequential C programs using the ANSI/ISO C (ACSL) language. The ACSL language is inspired by the OpenJML specifications and thus looks similar syntactically. It is possible to do runtime verification using the E-ACSL [22] plugin in Frama-C. E-ACSL transforms an annotated C program into a new C program that has an inline monitor generated using runtime assertion checks.

Even though E-ACLS and the prototype are built for different languages, they work similarly in functionality. Both systems first transform the program into an AST that can be altered such that it can verified during runtime. Similar to OpenJML, E-ACLS can generate an executable that can be run, this is unfortunately not possible yet in the prototype.

7.1.2 Whiley

Whiley [20] is a sequential programming language with verification built into it. In Whiley, it is possible to create pre- and postconditions inside the function declaration or variable declarations. These conditions will be statically verified during the compilation of the program. Additionally, it is also possible to do the verification during runtime. The main difference between VerCors and Whiley is that VerCors is used for verifying other languages and Whiley is a language by itself. The advantage of verifying the language itself is that new features in Whiley also require to be verified by the compiler. In comparison, VerCors needs to adapt to new features of the languages that they support.

7.2 Runtime verification for concurrent programs

7.2.1 Old prototype

Gijsen [9] introduced a limited prototype to perform runtime assertion checking for VerCors. Gijsen investigated what the best methodology would be to do permission checking in runtime assertion checking for fields and objects. The prototype uses symbolic permission checking to verify if a thread has enough permission to a field. They store all the threads in a list that wants to have access, if there is only one thread in the list it has write permission. Otherwise, all the threads in the list have read permission. The prototype does not work with the current version of VerCors and should be revived so new features can be added to the runtime verification.

The new prototype replaces the prototype made by Gijsen. It uses fractional permission checking, similar to the fractional permission checking that VerCors uses for static verification. Additionally, the new prototype supports more features than the old prototype making it more usable than the old prototype.

7.2.2 OpenJML

Kandziora [15] introduced a way to do runtime assertion checking for concurrent programs in OpenJML. Kandziora makes use of the e-STROBE framework to take snapshots of the memory during execution. Therefore the OpenJML assertion checks can be executed on the snapshots instead. This protects the runtime assertion checker against assertion interference, which is when an assertion is being interfered by another thread, and results in unexpected behaviour in the assertion check. However, this ensures thread-safety of the assertion checking but does not provide information about checking if the program is thread-safe or not.

7.2.3 Concurrent and Distributed systems

Din et al. [8] performed a comparison between runtime assertion checking and deductive verification on concurrent and distributed systems, making use of a history-based specification approach. For runtime verification they made use of Maude backend of ABS. They extended this language such that pre- and postconditions and invariants are supported. For static verification, they used Theorem proving using the KeY theorem prover. In their comparison, they mention that runtime-assertion checking can be used to detect the presence of bugs, while formal verification can prove that the code is correct.

7.3 Static verification tools concurrent programs

7.3.1 VerCors

VerCors [3] is a static verification tool that verifies if concurrent software conforms to specified formal specifications written in Permission-Based Separation Logic (PBSL). VerCors makes use of fractional permissions, which in short is a method that stores a fraction between zero and one for each field and verifies if the thread has enough of that fraction to access the field. Fractional permissions are in Section 3.2. VerCors is the tool that is used for the prototype, therefore it is discussed in detail in Chapter 4.

7.3.2 VeriFast

VeriFast [14] is a tool that verifies sequential and concurrent software in C or Java using pre- and postconditions for each method. VeriFast also supports the use of fractional permissions to verify if a thread has enough access to a field. Additionally, using a plugin in VeriFast also supports counting permissions, which is more applicable for verifying programs that use reference counting for resource management.

7.3.3 VCC

Similarly to VerCors and Verifast, VCC [6] verifies programs using annotations for function pre- and postconditions, assertions, type invariants, and ghost code. VCC aims at verifying low-level concurrent C code, where each method will be verified individually. Permissions are being checked making use of claims, which detect the opening and closing of an object and the reference count to that object. This is applicable for objects, but not primitives,

so they create a *LOCK* for primitive fields to keep track of the claim.

7.3.4 Chalice

Chalice [16] (not maintained anymore) uses pre- and postconditions to verify if the program behaves as intended. It is possible to provide read and write access to objects/fields using percentages, where 100% is write permission and nonzero % is read permission. This is similar to a fractional permission approach.

7.3.5 GPUVerify

GPUVerify [2], as the name implies, verifies GPU kernels written in OpenCL and Cuda. For this, they create the synchronous, delayed visibility (SDV) semantics to verify if the program works correctly. In SDV, group executions are *synchronous*, meaning that the divergence of the program can be checked precisely. Every thread in the kernel creates a shadow of the shared memory, then it alters the shadow and thus limits the *visibility* of the thread on the shared memory. When all threads reach a barrier (*delayed*), the *visibility* of the shared memory is re-established. Using all of the shared memories of the threads, GPUVerify can determine if the program is race- and divergence-free and thus proves if the program behaves correctly.

Chapter 8

Conclusion

In this research, we worked on a prototype that extends the functionality of the prototype that Gijsen made. The prototype of Gijsen had the functionality to perform permission checking for fields of objects. Unfortunately, the prototype was not functional anymore with the current version of VerCors, so the prototype had to be revived so that new features can be added to the runtime checker.

The new prototype we developed starts by having the same functionality as the prototype of Gijsen The prototype makes use of fractional permission checking instead of symbolic permission checking that Gijsen used. This ensures that the same syntax for static verification can be reused in runtime assertion checking. All the permissions are stored in a global map that can be accessed by any thread anywhere in the program.

Next to basic permissions checking the prototype successfully supports the functionality for 5 other features. The first feature is permission checking for arrays, which reuses the same global map as basic permission checking and thus requires not a significant amount of memory to be supported as well. The second feature is the support of quantifiers, which are translated into for-loops in the generated code. These for-loops can check conditions that are specified in any type of contract specification. The third feature is the support for loop invariants, which are similar to pre- and post-conditions of a method but are meant to check permissions for pre- and post-conditions of loops. The fourth feature is the support of lock invariants, which take a part of the permission of a class and release the permission when a thread enters a synchronized block that is synchronizing on that class. The final new feature is the support for predicates, where details of the predicates are stored if a thread holds the predicate. A predicate can temporarily remove the permissions and restore them when the predicate unfolds.

Each feature is evaluated using a test case. The test cases should not throw any error if they are executed and by altering the test cases slightly, the test case should throw an error. As of now, all test cases throw the error that was expected and therefore the prototype works as intended. However, more and larger tests are required to show that the prototype is completely correct. The test cases produce a method and line coverage of 89% and 92% respectively, which shows that most of the prototype is tested thoroughly. Finally, the test cases are significantly faster for runtime verification than static verification. Runtime verification took an average of 5.039 seconds while static verification took 14.923 seconds. The prototype now supports a variety of features but other features are still available to be added to the prototype, such as support for the *old* and *result* keywords. Additionally,

an automated testing environment can be created such that the generated code can be executed directly and new features of the prototype can be tested immediately. Finally, a scoping mechanism for storing permissions per method can be used to determine if the thread has enough permission in the method.

Chapter 9

Future work

This section describes what can be done to improve/optimize the functionality of the runtime assertion checker in VerCors.

9.1 New features

The proposed prototype has new features, but there are still some features that are not implemented yet, for instance, the *old* and *result* keywords. The *result* keywords determine what the result of the function is and can be used in a post-condition to verify if the result of the function is correct. The *old* keyword is used to show behaviour in the post-condition where the program uses the value of the object/field that was in the pre-condition. Therefore a duplicate of this object/field must be created so that it can be used in the post-condition, which is a difficult task.

9.2 Automated testing environment

As mentioned in Chapter 6, the prototype does not support the functionality to be tested automatically. The developer must adjust the output program manually as mentioned in Chapter 6. However, the most optimal solution would be that the transformed program is automatically loaded into a Java Runtime Environment and is executed in this environment. Resulting in that the developer does not see the generated code at all. Additionally, this allows automated testing of new features that are added to the prototype.

9.3 Scoping permissions per function

In the static version of VerCors it is possible to specify in the pre-condition that the thread has only read permission for a field, which implies that the field cannot be changed in the method. In the current prototype, it is possible to change the field even though the pre-condition specified that it only be read if the thread has write permission for the field. Future research can be done to create a scoping mechanism for permissions, where each time the thread enters the method the topmost permission is checked and the new permission is pushed at the top of the stack. When the thread leaves the method it will pop from the stack.

Bibliography

[1] Scalacenter , Lightbend , and Virtuslab . Scala Programming Language, 2002. URL: https://www.scala-lang.org/. 4.2

[2] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. GPUVerify. *ACM SIGPLAN Notices*, 47:113–132, 11 2012. doi:10.1145/2398857.2384625. 7.3.5

[3] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software, 2017. URL: https://research.utwente.nl/en/publications/the-vercors-tool-set-verification-of-parallel-and-concurrent-soft, doi:10.1007/978-3-319-66845-1_7. 1, 1.1, 7.3.1

[4] John Boyland. Checking interference with fractional permissions, 06 2003. URL: http://dl.acm.org/citation.cfm?id=1760267.1760273. 3.2

[5] Stephen Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375:227–270, 05 2007. doi:10.1016/j.tcs.2006.12.034. 3.1

[6] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. *Lecture Notes in Computer Science*, pages 23–42, 08 2009. doi:10.1007/978-3-642-03359-9_2. 7.3.3

[7] Pascal Cuoq, Florent Kirchner, Nikolaï Kosmatov, Virgile Prévosto, Julien Signoles, and Boris Yakobowski. Frama-C. *Lecture Notes in Computer Science*, pages 233–247, 01 2012. doi:10.1007/978-3-642-33826-7_16. 2.2, 7.1.1

[8] Crystal Din, Olaf Owe, and Richard Bubel. Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems Envisage: Engineering Virtualized Services View project Credo: Modeling and analysis of evolutionary structures for distributed services View project Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems. 2014. doi:10.5220/0004877804800487. 7.2.3

[9] Stijn Gijsen. Runtime Permission Checking in Concurrent Java Programs. Master's thesis, 2015. 1.2, 2.5, 5.1, 6.1, 6.1, 7.2.1

[10] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41:1–76, 02 2009. doi:10.1145/1459352.1459354. 2.2

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 10 1969. doi:10.1145/363235.363259. 2.2, 3.1

[12] Marieke Huisman and Wojciech Mostowski. A Symbolic Approach to Permission Accounting for Concurrent Reasoning. *2015 14th International Symposium on Parallel and Distributed Computing*, 06 2015. doi:10.1109/ispdc.2015.26. 3.3

[13] Reiner Hähnle and Marieke Huisman. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. *Lecture Notes in Computer Science*, 10000:345–373, 2019. doi:10.1007/978-3-319-91908-9_18. 2.3.1

[14] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. *Lecture Notes in Computer Science*, pages 41–55, 04 2011. doi:10.1007/978-3-642-20398-5_4. 7.3.2

[15] Jorne Kandiziora. Runtime Assertion Checking of Multithreaded Java Programs. Master's thesis, 2014. 7.2.2

[16] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of Concurrent Programs with Chalice. *Lecture Notes in Computer Science*, 5705:195–222, 01 2009. doi:10.1007/978-3-642-03829-7_7. 7.3.4

[17] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. *Lecture Notes in Computer Science*, 9583:41–62, 01 2016. doi:10.1007/978-3-662-49122-5_2. 4.2

[18] Peter W O'Hearn, John V Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. pages 1–19, 09 2001. doi:10.1007/3-540-44802-0_1. 3.1

[19] OpenJML.org. Home | OpenJML, 2015. URL: https://www.openjml.org/. 2.2, 7.1

[20] David J Pearce and Lindsay Groves. Whiley: A Platform for Research in Software Verification. *Lecture Notes in Computer Science*, pages 238–248, 01 2013. doi:10.1007/978-3-319-02654-1_13. 7.1.2

[21] Violet Ka I Pun, Martin Steffen, and Volker Stolz. Deadlock checking by data race detection. *Journal of Logical and Algebraic Methods in Programming*, 83:400–426, 09 2014. URL: https://www.sciencedirect.com/science/article/pii/S2352220814000492, doi:10.1016/j.jlamp.2014.07.003. 2.1.1, 2.1.1

[22] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper), 2017. 7.1.1

[23] Steven Thomson. Validation and Verification of Formal Specifications in ObjectOriented Software Engineering, 2000. URL: https://scholar.afit.edu/cgi/viewcontent.cgi?article=5872&context=etd. 2.2

Appendix A

Detailed Prototype Implementation

This appendix will describe each class that is used in the prototype individually and what the responsibilities are of these classes.

A.1 Details Helper Classes

This section will discuss all the helper classes individually on how they work in detail.

A.1.1 Util

The *Util* class provides functions that can be used to find certain nodes/properties of the AST. These functions are not related to any specific transformation stage and can thus be generalized in the *Util* class and be used by different transformation stages.

A.1.2 LedgerHelper

The *LedgerHelper* class is used to create functions for the Ledger class. Moreover, other transformation stages can make use of the LedgerHelper to create method invocations to set and get permissions, predicates, and join tokens. The *LedgerHelper* file consists of 4 parts.

Static methods

The first part is a helper function called *findNumberInstanceField* that can retrieve the associated number of any instance field anywhere in the program.

LedgerRewriter

The second part is the *LedgerRewriter*, which is responsible for rewriting the Ledger and the *DataObject* class. This rewriter will most likely be called before each rewrite stage so that the Ledger can be used in each stage.

LedgerMethodBuilderHelper

The third part is the *LedgerMethodBuilderHelper* case class, which is responsible for easily accessing functions and creating method invocations to the Ledger. This class will be

initialized in the *LedgerRewriter* so that it can be used in the transformation stages.

DataMethodBuilderHelper

The final part is the *DataMethodBuilderHelper* case class, which is used for accessing functions and creating method invocations to the *DataObject* class. This class will also, like the *LedgerMethodBuilderHelper*, be initialized in the *LedgerRewriter* so that it can be used in the transformation stages.

A.1.3 CreateConstructor

When VerCors transforms the program into their internal representation named Col, the constructor of each object is removed and is put into a *Procedure* node. In the final output, the constructor needs to be re-established, so the job of the *CreateConstructor* class is to recreate the old constructor to its original form.

A.1.4 PermissionData

The *PermissionData* class will be used by the *RewriteContractExpr* and *TransferPermissionRewriter* rewriters to create statement(s) for verifying and transferring permissions respectively. Table A.1 shows an overview of what can be stored in the *PermissionData* class.

Content	**Description**
cls	The class that currently is being rewritten
outer	The Rewriter class that is creating this PermissionData class
Factor	If not all of the permission should be transferred a factor is used to transfer permissions
threadId	On which thread object the permission check/transfer should be operated
offset	The offset is used to change the *this* keyword to another object. This is useful for transferring permissions and collecting the pre/post permissions of the *run* method, but on the correct object
ledger	Holds an instance of the *LedgerMethodBuilderHelper* so that the Rewriters can access the ledger
injectivityMap	Holds a variable to the closest injectivityMap possible.

Table A.1: Content of *PermissionData* class

A.1.5 FindBoundsQuantifier

The purpose of this class is to identify what the upper and lower bounds of a quantifier are. It will search through the quantifier expression to retrieve the upper and lower bounds. If it is not possible to find a lower and upper bound then it will use the minimal and maximal values respectively for an integer.

A.1.6 AbstractQuantifierRewriter

The *AbstractQuantifierRewriter* class is an abstract rewriter class for quantifiers. It has functions that can be used to transform a quantifier into a for-loop. The class makes

use of the *FindBoundsQuantifier* to collect the bounds for the new for-loop. The method that should be implemented by the class extending the *AbstractQuantifierRewriter* is the *dispatchLoopBody* method so that the *Rewriter* can determine what the body of the loop should be.

A.1.7 RewriteContractExpr

The *RewriteContractExpr* class is an extension of the *AbstractQuantifierRewriter* and focuses on transforming a contract expression into assert statements. It first unfolds the expression into smaller segments that can be checked using a single assertion check. After that, it will transform the segments into assert statements. When a quantifier occurs then it will use the functions of the *AbstractQuantifierRewriter* to transform the quantifier into a for-loop and uses the *dispatchLoopBody* to insert the assert statements into the for loop.

A.1.8 TransferPermissionRewriter

The *TransferPermissionRewriter* class is also an extension of the *AbstractQuantifierRewriter* and focuses on transforming a contract expression into adding/removing statements for transferring permissions.

Similar to the *RewriteContractExpr* it will also first unfold the contract expression into smaller segments. Then only the *Perm* and *Starall* nodes are transformed into permission transferring statements. The *Perm* node is normally used to check if a thread has enough permission for an object. The *Starall* node represents the $forall*$ notation, which can also hold permissions and thus should be able to transfer the permission. All other expressions cannot hold permissions and thus do not have to be rewritten into transferring statements. When the expression has an *Starall* node it will make use of the *dispatchLoopBody* method of the *AbstractQuantifierRewriter* to create the transferring of permissions inside the newly created for-loop.

A.2 Details Stage Classes

This section will discuss in detail how the stage classes work individually and how they contribute to creating the final implementation.

A.2.1 CreateLedger

The *CreateLedger* stage is responsible for creating the *Ledger* and the *DataObject* classes. These classes are very important for handling permissions, predicates, and join tokens. Both classes are being generated in this class making use of the *DataMethodBuilderHelper* and *LedgerMethodBuilderHelper*. Whenever a field or method is created the two helper classes are recreated, so that everything that has been generated already is available for all the other methods to find fields and call methods of the *Ledger* and *DataObject* classes.

The *DataObject* class and the *Ledger* class will be discussed separately to give a better understanding of how each of the classes works and what they contribute to the final output.

DataObject

This section will discuss everything about the *DataObject* class. We will discuss each field and method separately and what the purpose is of each field and method.

Listing A.1 shows the *data* field. This field holds the associated *Object[]* that needs to be compared to other *Object[]* with similar content.

```
Object[] data;
```

LISTING A.1: *DataObject*: data field

Listing A.2 shows the *setData* method. This method sets the *data* field of the *DataObject*. We can simply not use a constructor, because it is not a natively supported Node by the internal language Col. It was not possible to create a parameter in the constructor, which can then be assigned to the field. Since this was not a possibility we created a function to set the data.

```
void setData(Object[] data) {
        this.data = data;
}
```

LISTING A.2: *DataObject*: setData method

How can a *DataObject* be created if the class does not have a constructor? For this, the static method *create* is used. Since methods are supported by the Col language, this was a relatively easy workaround. Listing A.3 shows how the *create* method works.

```
static DataObject create(Object[] data) {
    DataObject object;
    object = new DataObject();
    object.setData(data);
    return object;
}
```

LISTING A.3: *DataObject*: create method

Listing A.4 shows the *equals* method of the *DataObject* class. This is also the *equals* method that will be used in the HashMaps/ArrayList to compare objects with each other. This makes the most important method of the *DataObject* class. First, it is checked if the object that is compared to it is the same as itself. Then it checks if the *obj* is not null and if it is of the same type as itself. Finally, it can cast the other *Object* and make use of the *Arrays.equals* method to compare both *data* fields of both *DataObject* objects.

```
boolean equals(Object obj) {
    DataObject otherObject;
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }
    otherObject = (DataObject) obj;
```

```
    return Arrays.equals(this.data, otherObject.data);
}
```

Listing A.4: *DataObject*: equals method

Finally, Listing A.5 shows the *hashCode* method. It overrides the *Object* hashCode method and uses the *Arrays.hashCode* method to create a hash for the *DataObject*.

```
int hashCode() {
    return Arrays.hashCode(this.data);
}
```

Listing A.5: *DataObject*: hashCode method

Ledger

This section will discuss everything about the *Ledger* class. We will discuss each field and method separately and what the purpose is of each field and method. For initializing all *HashMaps* fields we use the *Collections.synchronizedMap* to ensure that the *HashMap* is concurrently safe. It is allowed to use a locking mechanism for storing data for the runtime environment because this will not introduce locking for the program itself, but only for the storage of the runtime data. Moreover, we make use of *WeakHashMap* to still be able to do garbage collection when threads are done with their execution.

Listing A.6 shows the *__runtime__* field of the *Ledger* class. This field holds all the permissions for each *Object* that has been created. The type of the field is *Map<Long, Map<Object, Fraction»* where the *Long* represents the id of the thread. The *Object* is the object for which permission is stored and the *Fraction* is the permission of that object.

```
static Map<Long, Map<Object, Fraction>> __runtime__ = Collections.
    synchronizedMap(new WeakHashMap<>());
```

Listing A.6: *Ledger*: permission store field

The *__join_tokens__* field holds all the join tokens in the system, see Listing A.7. The *Object* key refers to the started thread and the *Fraction* value is the remainder of the join token of that thread.

```
static Map<Object, Fraction> __join_tokens__ = Collections.
    synchronizedMap(new WeakHashMap<>());
```

Listing A.7: *Ledger*: join token store field

The final field is the *__predicate_store__*, see Listing A.8, which holds all the predicates for each thread. The *Long* object refers to the id of the thread. The value is of type *ArrayList<DataObject»*, which holds all the predicates that are in the system for a specific thread. Therefore we cannot use a *Map* to store the predicates, because a thread can hold multiple predicates of the same predicate type. Thus in order to store multiple predicates of the same type they need to be stored in a list.

```
static Map<Long, ArrayList<DataObject>> __predicate_store__ =
    Collections.synchronizedMap(new WeakHashMap<>());
```

Listing A.8: *Ledger*: predicate store field

Listing A.9 shows the *createHashMap* method. This method is used to initialize the *__runtime__* and the *__predicate_store__* when a new thread wants to set/get permission or predicates from the predicate store.

```
static void createHashMap() {
    if (!(LedgerRuntime.__runtime__.containsKey(Thread.currentThread().
        getId()))) {
         LedgerRuntime.__runtime__.put(Thread.currentThread().getId(),
             Collections.synchronizedMap(new WeakHashMap<>()););
    }
    if (!(LedgerRuntime.__predicate_store__.containsKey(Thread.
        currentThread().getId()))) {
         LedgerRuntime.__predicate_store__.put(Thread.currentThread().
             getId(), new ArrayList<DataObject>());
    }
}
```

Listing A.9: *Ledger*: createHashMap method

The next method is the *getPermission* method shown in Listing A.10. This method first checks if the thread has its hashmaps initialized. Then it will retrieve the permission for a certain object. If the thread does not have permission stored for the object then it will return *Fraction.Zero*.

```
static Fraction getPermission(Object input) {
    LedgerRuntime.createHashMap();
    return LedgerRuntime.__runtime__.get(Thread.currentThread().getId()
        ).getOrDefault(input, Fraction.ZERO);
}
```

Listing A.10: *Ledger*: getPermission method

To set the permission for an object for a thread, we make use of the *setPermission* method, see Listing A.11. This method has a *Object* and *Fraction* as input. The fraction is the new value for the object for a thread. So first the fraction must be checked that it is valid (between 0 and 1). Then it needs to be checked that the thread has its *HashMap*s initialized. Finally, the value will be stored in the *__runtime__* permission storage.

```
static void setPermission(Object input, Fraction value) {
    assert (value.compareTo(Fraction.ZERO) != -1): "Permission␣cannot␣
        be␣below␣0";
    assert (value.compareTo(Fraction.ONE) != 1): "Permission␣cannot␣
        exceed␣1";
    LedgerRuntime.createHashMap();
    LedgerRuntime.__runtime__.get(Thread.currentThread().getId()).put(
        input, value);
}
```

Listing A.11: *Ledger*: setPermission method

When a new Object is created the *initiatePermission* (see Listing A.12) method will be used. This method receives an input *Object* and an optional integer called size. The size is used to show how many fields the *Object* has. Then the permission for the *Object* and the fields of the *Object* will be set to *Fraction.One*.

```java
static void initiatePermission(Object input, int size) {
    int i;
    LedgerRuntime.setPermission(input, Fraction.ONE);
    for (i = 0; i < size; i = i + 1) {
        LedgerRuntime.setPermission(DataObject.create(new Object[]{
            input, i}), Fraction.ONE);
    }
}
static void initiatePermission1(Object input) {
    LedgerRuntime.initiatePermission(input, 0);
}
```

Listing A.12: *Ledger*: initiatePermission method

The *getJoinToken*, shown in Listing A.13, returns the join token of the thread that is being joined.

```java
static Fraction getJoinToken(Object input) {
    return LedgerRuntime.__join_tokens__.get(input);
}
```

Listing A.13: *Ledger*: getJoinToken method

The *setJoinToken*, shown in Listing A.14, will set the value of the join token to a new *Fraction*. This fraction is first checked that it is not below 0. This method is called when a thread is joining another thread and takes a part of its permission.

```java
static void setJoinToken(Object input, Fraction value) {
    assert (value.compareTo(Fraction.ZERO) != -1): "Join token cannot
        be below 0";
    LedgerRuntime.__join_tokens__.put(input, value);
}
```

Listing A.14: *Ledger*: setJoinToken method

Listing A.15 shows the *hasPredicateCheck* method, that checks if the thread holds a predicate that is equals to the input *Object*.

```java
static void hasPredicateCheck(Object[] input) {
    assert (LedgerRuntime.__predicate_store__.get(Thread.currentThread
        ().getId()).contains(DataObject.create(input))): "Thread does
        not own the predicate";
}
```

Listing A.15: *Ledger*: hasPredicateCheck method

The *foldPredicate* method(see Listing A.16) first checks if the thread has its *HashMaps* initialized. Then it will store the newly created predicate into the *__predicate_store__*.

```java
static void foldPredicate(Object[] input) {
    LedgerRuntime.createHashMap();
    LedgerRuntime.__predicate_store__.get(Thread.currentThread().getId
        ()).add(DataObject.create(input));
}
```

LISTING A.16: *Ledger*: foldPredicate method

The *unfoldPredicate* method(see Listing A.17) also first checks if the thread has the *HashMaps* initialized. Then it will check if the thread owns such a predicate. Finally it will remove the predicate from the *__predicate_store__*

```
static void unfoldPredicate(Object[] input) {
    LedgerRuntime.createHashMap();
    LedgerRuntime.hasPredicateCheck(input);
    LedgerRuntime.__predicate_store__.get(Thread.currentThread().getId
        ()).remove(DataObject.create(input));
}
```

LISTING A.17: *Ledger*: unfoldPredicate method

Finally, in Listing A.18, is the *checkForInjectivity* method. This method receives a mapping of *Object* to *Fraction*. This method will check if the wanted permissions in the mapping are equal or less than the permission for that object in the *__runtime__* permission store. This resolves the permission-checking shown in Listing 5.8.

```
static void checkForInjectivity(WeakHashMap<Object, Fraction>
    injectivityMap) {
    for (Object key : injectivityMap.keySet()) {
        assert (injectivityMap.get(key).compareTo(LedgerRuntime.
            getPermission(key)) != 1): "Permission cannot exceed 1 due 
            to injectivity";
    }
}
```

LISTING A.18: *Ledger*: checkForInjectivity method

A.2.2 RemoveSelfLoops

In the Resolution stage of VerCors, see Chapter 4, other objects and definitions that are not defined in the loaded program will be loaded into VerCors so that they normally can be used to do static verification. In the runtime verification, this is not the case and this results that all created classes extending the *Object* class. This is true, but they have to be removed if a class wants to extend another class, because in Java it is only allowed to have 1 extending class. So this would make it impossible for other classes to extend the *Thread* class. The *RewriteSelfLoops* rewriter is used to remove all loops back to the *Object* class.

A.2.3 RefactorGeneratedCode

The *RefactorGeneratedCode* will recreate the constructor for classes. It first determines if the *Class* node is an interface or a normal class. If the node is a class it will recreate the constructor using the *CreateConstructor* rewriter, otherwise it will not create a constructor.

A.2.4 AddPermissionOnCreate

The *AddPermissionOnCreate* is responsible for:

- Initializing permissions for classes and objects

  - *Classes*: The *AddPermissionOnCreate* searches for the constructor of a class and the number of fields the class has. Using the *initiatePermission* method from the *Ledger* class it is possible to instantiate the permissions for each field in the class.
  - *Arrays*: When a new array is created, the *initiatePermission* method of the *Ledger* is called with the size of the array.

- Initialize join token in the constructor if the class is extending the *Thread* class to *Fraction.ONE*

A.2.5 CreatePredicateFoldUnfold

The main function of the *CreatePredicateFoldUnfold* is to transform the fold/unfold statements to statements that can be used for handling predicates during runtime.

When a fold statement is found the following operations will be done to create statements for handling predicate during runtime:

1. Collect the expression of the predicate
2. Check if the thread has all the permissions that are required by the predicate
3. Remove the permissions from the predicate, using the correct object as an offset
4. Bundle predicate arguments, the object, and the predicate name into an *Object[]*
5. Call the *foldPredicate* method of the *Ledger*

When a unfold statement is found in the AST the following operations need to be performed to translate it to new statements for handling the unfold method:

1. Collect the expression of the predicate
2. Bundle predicate arguments, the object, and the predicate name into an *Object[]*
3. Call the *unfoldPredicate* method of the *Ledger*
4. Add permissions from the predicate back to the thread

A.2.6 CheckPermissionsBlocksMethod

The *CheckPermissionsBlocksMethod* rewriter is a relatively complex class and implements the idea of checking permissions when the heap changes mentioned in Section 5.1.2.

Figure A.1 shows the general steps the *CheckPermissionsBlocksMethod* rewriter will have to take. It starts by setting some global variables. The first variable is the *isTarget* variable which is a boolean value representing that the dereferences that are occurring are from the target. The second variable is the *dereferences* variable, which is of type *ScopedStack[HashMap[Expr, Boolean]]*. A *ScopedStack* is a utility class in VerCors, where it is possible to create a new scope and thus create a new object, in this case, *HashMap*, on the stack. We make use of a *ScopedStack* because multiple blocks can be nested into each other, requiring us to transform each block independently and thus keep track of the dereferences of only the block that it is currently in. The *HashMap* uses an *Expr* as key, this will be the node that holds a dereference to an object. The value is a *Boolean*, which

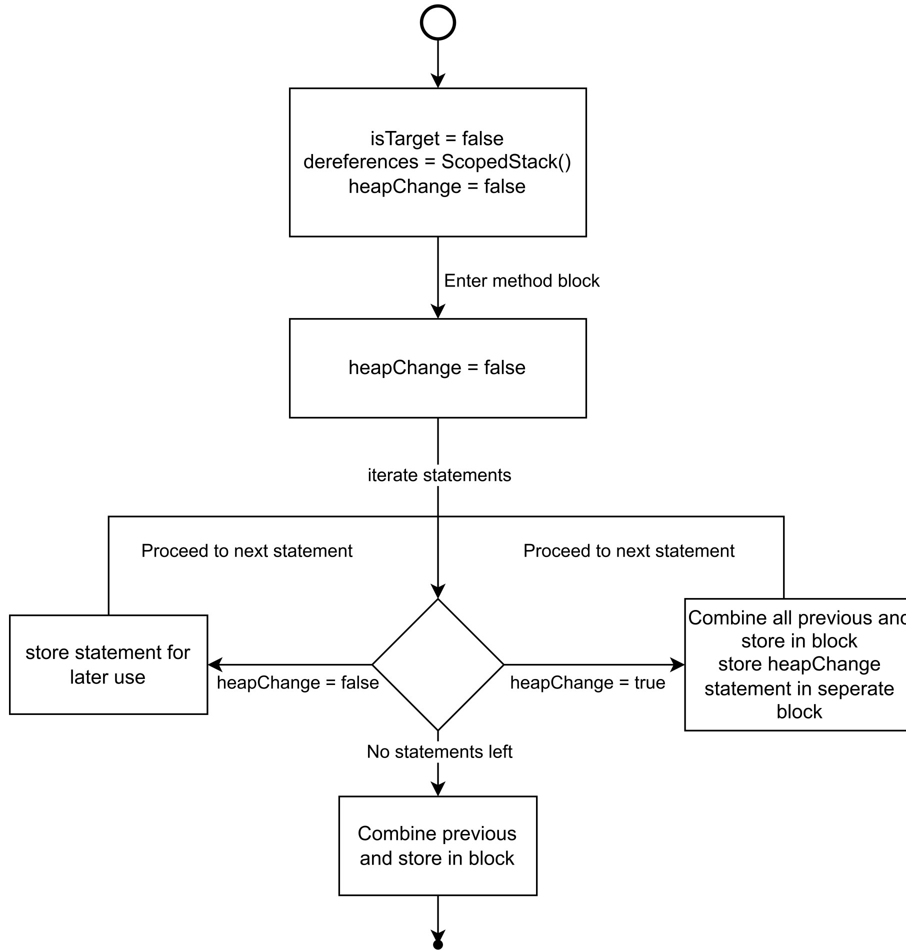

FIGURE A.1: *CheckPermissionsBlocksMethod* general flow for permission checking in blocks

represents true if the dereference needs write permission otherwise it will be set to false.

The second step in the figure is by entering a method block and setting the *heapChange* variable to false. Then all of the statements in the method will be evaluated. If the *heapChange* variable was set to false while evaluating the statement, then the statement will be stored in a buffer for later use. If the *heapChange* variable was set to true then it will collect all the statements from the buffer and transform it to a single block with the correct assertions. The statements that had a change in the heap will be put in their own block. The buffer and the dereferences will be cleared when proceeding to the new statement.

Finally, when there are no statements left of a block, all the remaining statements that are in the buffer will be put in a block with the correct assert checks for the dereferences.

Figure A.1 did not describe how the *isTarget* variable is being used. Figure A.2 shows the flow of how derefences are archived in the *dereferences* stack. The figure starts the same by initializing the *isTarget*, *dereferences*, and *heapChange* variables. Then it starts iterating over all the statements, which is explained in Figure A.1. If it finds a statement or expression that is an assignment, then before evaluating the left side for dereferences

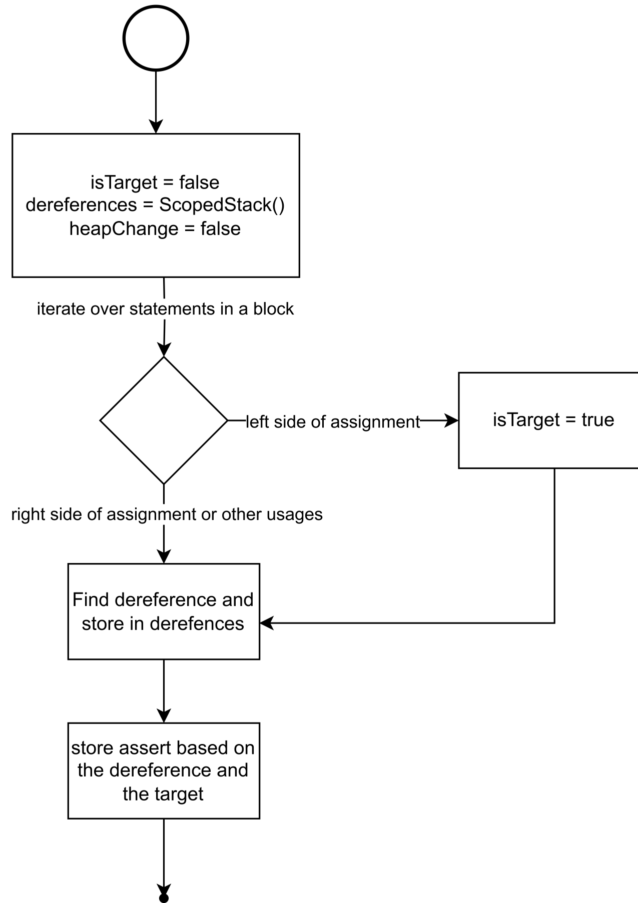

FIGURE A.2: *CheckPermissionsBlocksMethod* general flow for dereferences to asserts

the *isTarget* variable is set to true.

When a dereference is found in the AST the dereference is added to the *dereferences* map using the *isTarget* as its value. If the dereference already exists in the *dereferences* list it will update the value to *true* if the *isTarget* variable is *true*, otherwise it will not change the map. Due to this we know that when the boolean value of the dereference is set to true it always remains true and thus require write permission.

Finally, when a block is formed all the dereferences of that block will be transformed using the *dereference* map to assert statements that can be put at the beginning of the block.

A.2.7 CreatePrePostConditions

The *CreatePrePostConditions* rewriter is used to transform pre- and postconditions of a method into a method using assert statements. This also initializes the *injectivityMap* for each method. The preconditions are transformed using the *RewriteContractExpr* to assert statements which are checked at the beginning of the method. Normally the preconditions are checked at the call point in static verification, however, this would generate a significant amount of code. When inserting it at the beginning of the method, the preconditions are generated only once and keep the code as clear as possible. The postconditions are

transformed by the *RewriteContractExpr* as well but are inserted before each *Return* node or at the end of the function if there is not a *Return* node.

A.2.8 CreateLocking

The *CreateLocking* rewriter, converts a lock invariant to work in a runtime environment. It will do the following steps to ensure that locking works as intended:

1. When a class has a lock invariant, remove permissions of lock invariant in the constructor of that class

2. When entering a synchronized block, add permissions (using *TransferPermissionRewriter*) of the lock invariant to the thread that entered the synchronized block

3. Before leaving a synchronized block, check that the thread has enough permissions to reestablish the lock invariant (using *RewriteContractExpr*). If it has enough remove the permissions (using *TransferPermissionRewriter*) of the thread. If it does not have enough permissions then throw an error

A.2.9 CreateLoopInvariants

The *CreateLoopInvariants* rewriter converts the loop invariant using the *RewriteContractExpr* and inserts the statements at the beginning and the ending of the loop. Additionally, when a *continue*, *break*, or *return* keyword is found the statements are inserted before the keywords because these keywords will end the loop prematurely and thus the loop invariant needs to be checked there as well. False assert statements inside the loop refer back to the loop invariant when the error is thrown. So this rewriter is similar to the *CreatePrePostConditions*, but it inserts the post statements at more places than the *CreatePrePostConditions* rewriter does.

A.2.10 ForkJoinPermissionTransfer

The main responsibility of the *ForkJoinPermissionTransfer* stage is to transfer the permission of threads when a new thread is started or when the thread is joined. Different steps need to be taken to achieve this functionality:

1. When finding the *run* method of a Class that is extending the *Thread* class, transfer the permissions in the precondition of the *run* method to the thread. When the thread is leaving the *run* method and thus is done with its execution, check and remove the permissions that are in the postcondition of the *run* method from the thread.

2. When a *start* method invocation occurs, remove the permissions of the preconditions of the *run* method from the current thread, before starting the new thread. The *Ledger* will check if the current thread has enough permission to start the new thread, so there cannot be a thread without enough permission.

3. When a *join* method invocation occurs, 3 steps need to occur to retrieve the permission from another thread

    (a) Collect the join token that is specified above the *join* method invocation

    (b) Check using the *Ledger* that there is enough left of the join token and set the new remainder of the join token otherwise throw an error

(c) Add the permission to the joining thread using the join token as a factor for the permissions that should be transferred

A.2.11 GenerateJava

The final stage is the *GenerateJava* rewriter. This rewriter has the responsibility to add/remove nodes from the AST so that it is as close as possible to executing the program. This stage is not performed in the *Transformation* stage in VerCors, but in the *CodeGeneration* stage, which is an additional stage that is not normally used in VerCors. This stage works similarly to the *Transformation* stage but is less strict about rewriting the code and thus allows us to use the Java Nodes without references in VerCors.

Appendix B

Code of tests

This appendix shows all the tests that we will use to validate if the implementation of the features is correct.

B.1 Test case for reviving code Gijsen

```
import java.util.*;


class Source extends Thread {
    int i;

    /*@
        requires Perm(this.i, 1);
        ensures Perm(this.i, 1);
     */
    public void run() {
        i = 42;
    }

    public void join(){}
    public void start(){}
}

class Sink extends Thread {
    Source source;

    public void setSource(Source source) {
        this.source = source;
    }

    /*@
       requires Perm(source, 1);
       ensures Perm(source, 1);
       ensures Perm(source.i, 1/2);
    */
    public void run() {

        //@ source.postJoin(1\2);
        source.join();
    }
```

```
    public void join(){}
    public void start(){}
}

class Main{

    public void main() {
        Source source = new Source();
        Sink sink = new Sink();
        sink.setSource(source);

        source.start();
        sink.start();
        //@ source.postJoin(1\2);
        source.join();
        //@ sink.postJoin(1);
        sink.join();
        source.i = 1988;
    }
}

class Thread{

}
```

Listing B.1: Shared buffer

After changing the program to fail:

```
...
    public void run() {

        //@ source.postJoin(1\2);
        source.join();
        source.i = 1;
    }

...
```

Listing B.2: Change in program to fail Shared Buffer case

B.2 Test cases for new features

B.2.1 Arrays

```
class Source extends Thread {
    int[] i;

    public void createI() {
        this.i = new int[2];
    }

    /*@
        requires Perm(this.i, 1);
```

```
        requires (\forall* int j; 0 <= j && j < i.length; Perm(i[j],
            write));
        ensures Perm(this.i, 1);
        ensures (\forall* int j; 0 <= j && j < i.length; Perm(i[j],
            write));
     */
    public void run() {
        i[0] = 42;
        i[1] = 43;
    }


    public void join(){}
    public void start(){}
}

class Sink extends Thread {
    Source source;

    public void setSource(Source source) {
        this.source = source;
    }

    /*@
       requires Perm(source, 1);
       ensures Perm(source, 1);
       ensures Perm(source.i, 1/2);
       ensures (\forall* int j; 0 <= j && j < source.i.length ==> Perm(
           source.i[j], 1/2));
    */
    public void run() {
        //@ source.postJoin(1\2);
        source.join();
    }

    public void join(){}
    public void start(){}
}

class Main{

    public void main() {
        Source source = new Source();
        Sink sink = new Sink();
        sink.setSource(source);
        source.createI();

        source.start();
        sink.start();
        //@ source.postJoin(1/2);
        source.join();
        //@ sink.postJoin(1);
        sink.join();
        source.i[0] = 1988;
    }
}
```

```
class Thread{

}
```

Listing B.3: Shared buffer arrays

```
...
    public void run() {
        //@ source.postJoin(1\2);
        source.join();
        source.i[0] = 1;
    }
...
```

Listing B.4: Change in program to fail Shared Buffer array case Scenario 1

```
...
class DummyData {
    int val;
}

class Source {
    DummyData[] i;

    public void createI() {
        this.i = new DummyData[2];
        DummyData o = new DummyData();
        this.i[0] = o;
        this.i[1] = o;
    }

    /*@
        requires Perm(this.i, 1);
        requires (\forall* int j; 0 <= j && j < i.length; Perm(i[j].val
            , write));
        ensures Perm(this.i, 1);
        ensures (\forall* int j; 0 <= j && j < i.length; Perm(i[j].val,
            write));
     */
    public void test() {

    }

    public void main(){
        Source source = new Source();
        source.createI();
        source.test();
    }
}

...
```

Listing B.5: Change in program to fail Shared Buffer array case Scenario 2

B.2.2 Quantifiers

```
class Program {
    int[] a;
    int b;


    /*@
        requires (\forall* int i; 0 <= i && i < a.length; Perm(a[i],
            write));
     */
    public void setA(int[] a) {
        this.a = a;
    }

    /*@
        requires a.length > 0;
        requires (\forall* int x; 0 <= x && x < a.length; Perm(a[x],
            write));
        requires (\forall int x; 0 <= x && x < a.length; a[x] > 0);
        requires (\forall int x; 0 <= x && x < a.length; (\forall int j
            ; 0 <= j && j < x; a[j] <= a[x]));
        ensures (\forall* int x; 0 <= x && x < a.length; Perm(a[x],
            write));
        ensures (\forall int x; 0 <= x && x < a.length; a[x] > 0);
     */
    public int indexOf(int b) {
        for (int i = 0; i < a.length; i++) {
            if (a[i] == b) {
                return i;
            }
        }
        return -1;
    }

    public void main() {
        Program program = new Program();
        int[] tmp = new int[3];
        tmp[0] = 2;
        tmp[1] = 4;
        tmp[2] = 6;
        program.setA(tmp);
        program.indexOf(4);
    }
}
```

LISTING B.6: Quantifiers test case

```
...
    /*@
        requires a.length <= 0;
        requires (\forall* int x; 0 <= x && x < a.length; Perm(a[x],
            write));
        requires (\forall int x; 0 <= x && x < a.length; a[x] <= 0);
        requires (\forall int x; 0 <= x && x < a.length; (\forall int j
            ; 0 <= j && j < x; a[j] < a[x]));
```

```
        ensures (\forall* int x; 0 <= x && x < a.length; Perm(a[x],
            write));
        ensures (\exists int x; 0 <= x && x < a.length; a[x] <= 0);
    */
...
```

LISTING B.7: Change in program to fail quantifiers case

B.2.3 Loop Invariants

```
class Program {
    int[] a;

    public void setA(int[] a) {
        this.a = a;
    }

    public int indexOf(int b) {
        //@ loop_invariant i <= a.length;
        //@ loop_invariant (\forall* int j; 0 <= j && j < a.length;
            Perm(a[j], read));
        //@ loop_invariant (\forall int j; 0 <= j && j < a.length; a[j]
            % 2 == 0);
        for(int i = 0; i < a.length; i++) {
            if(a[i] == b) {
                return i;
            }
        }
        return -1;
    }

    public void main() {
        Program program = new Program();
        int[] tmp = new int[3];
        tmp[0] = 2;
        tmp[1] = 4;
        tmp[2] = 6;
        program.setA(tmp);
        program.indexOf(4);
    }
}
```

LISTING B.8: Loop Invariants test case

```
...
    //@ requires (\forall int x; 0 <= x && x < a.length; (\forall int j
        ; 0 <= j && j < x; a[j] >= a[x]));
...
    public int indexOf(int b) {
        for (int i = 0; i < a.length; i++) {
            if (a[i] == b) {
                return i;
            }
        }
        return -1;
```

```
    }
```

LISTING B.9: Change in program to fail loop invariant case

B.2.4 Lock Invariant

```
//@ lock_invariant Perm(this.i, write);
class Source {
    int i;

    public void test(){
        synchronized (this) {
            this.i = 0;
        }
    }

    public void main() {
        Source source = new Source();
        source.test();
    }
}
```

LISTING B.10: Lock Invariant test case

```
...
    public void test(){
        int x = this.i;
        synchronized (this) {
            this.i = 0;
        }
    }
...
```

LISTING B.11: Change in program to fail lock invariant case

B.2.5 Predicates

```
public class Main {

    //@ requires c != null ** c.state(0);
    //@ ensures c.state(2);
    void foo(Counter c) {
        c.increment(2);
    }

    public void execute() {
        Counter c = new Counter();
        //@ fold c.state(0);
        Counter c2 = new Counter();
        //@ fold c2.state(0);

        foo(c);
        foo(c2);
    }
```

```
    public void main() {
        Main m = new Main();
        m.execute();
    }
}


class Counter {
    int count;

    //@ resource state(int val) = Perm(count, write) ** count == val;

    //@ requires state(count);
    //@ ensures state(count);
    void increment(int n) {
        //@ unfold state(count);
        count += n;
        //@ fold state(count);
    };
}
```

Listing B.12: Predicate test case

```
...
    void increment(int n) {
        int z = count;
        //@ unfold state(count);
        count += n;
        //@ fold state(count);
    };

...
```

Listing B.13: Change in program to fail predicate case