NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

PEDRO MIGUEL SILVA RIBEIRO SALGADO
Bachelor in Computer Science

SAFE AND SOUND LOCK GENERATION WITH DATA-CENTRIC CONCURRENCY CONTROL

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
March, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

SAFE AND SOUND LOCK GENERATION WITH DATA-CENTRIC CONCURRENCY CONTROL

PEDRO MIGUEL SILVA RIBEIRO SALGADO

Bachelor in Computer Science

Adviser: António Ravara
Associate Professor, NOVA University Lisbon

Co-advisers: Hervé Paulino
Associate Professor, NOVA University Lisbon

Mário Pereira
Assistant Professor, NOVA University Lisbon

Examination Committee:

Chairs: Doutora Ana Gualdina Almeida Matos
Assistant Professor, Lisbon University

Doutor Luís Manuel Marques da Costa Caires
Full Professor, NOVA University Lisbon

Doutor António Maria Lobo César Alarcão Ravara
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
March, 2023

Safe and sound lock generation with data-centric concurrency control

Copyright © Pedro Miguel Silva Ribeiro Salgado, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

Acknowledgements

This work was partially supported by the Fundação para a Ciência e Tecnologia (FCT) in the context of the DeDuCe project (PTDC/CCI-COM/32166/2017)

Abstract

The rise of multi-threaded programming propels shared-memory concurrency control to an even more important role, as it hinders program performance with multiple errors such as deadlocks.

Most research on concurrency control tends to be control-centric rather than data-centric. This approach focuses on marking the beginning and end of critical regions that require synchronization. However, it decentralizes concurrency and increases the likelihood of errors and bugs by the programmer.

In this state-of-the-art analysis, I examine existing control-centric approaches and their specifications, as well as provide a more in-depth analysis of a data-centric approach. I explain how these approaches prevent deadlocks, ensure serializability, and outline the annotations required from the programmer.

The RC^3 model aims to improve upon current data-centric concurrency control implementations. I provide a thorough explanation of the model's strategy and pipeline, step-by-step, with real examples of each step's purpose.

In this thesis we take the RC^3 Java algorithm as a starting point and create a version of it in OCaml. This OCaml version simplifies some of the complex details from the original Java code while keeping the critical logic intact. The thesis walks through each step of this OCaml implementation, explaining it thoroughly and providing practical code examples. Additionally, we use Cameleer, along with its annotations, combined with the Why3 graphical interface, to check that the OCaml implementations are correct and to ensure both the functionality and the robustness of the algorithm.

In the end, I provide a detailed explanation of the results I achieved with the OCaml implementation and Cameleer verification. I'll clarify which parts of the algorithm I successfully verified and point out areas that require future work. Additionally, I'll describe the OUnit tests I performed to ensure the robustness of my OCaml implementation.

Keywords: Data-centric Concurrency, Control-centric Concurrency, Deadlock-freedom, Serialization, Formal Verification

Resumo

A ascensão da programação multithread impulsiona o controlo de concorrência em memória partilhada para um papel ainda mais importante, uma vez que esta pode prejudicar o desempenho do programa com vários erros, tal como os deadlocks.

A maioria das investigações sobre o controlo de concorrência tende a ser `control-centric`, em vez de `data-centric`. Esta abordagem foca-se na marcação do início e do fim de regiões críticas que requerem sincronização. No entanto, esta descentralização da concorrência aumenta a probabilidade de erros e falhas por parte do programador.

Nesta análise do estado da arte, examino algumas abordagens `control-centric` existentes e respetivas especificações, além de fornecer uma análise mais aprofundada de uma abordagem `data-centric`. Explico também como estas abordagens previnem deadlocks, garantem a serializabilidade e delineio as anotações necessárias por parte do programador nelas.

O modelo RC^3 visa melhorar as implementações `data-centric` atuais de controlo de concorrência. Forneço uma explicação detalhada da estratégia e pipeline do modelo, passo a passo, com exemplos reais do propósito de cada passo.

Nesta tese, partimos do algoritmo em Java do modelo do RC^3 como ponto de partida e criamos uma versão em OCaml. Esta versão em OCaml simplifica alguns dos detalhes complexos do código Java original, mantendo a lógica crítica intacta. A tese percorre cada passo desta implementação em OCaml, explicando-a detalhadamente e fornecendo exemplos práticos de código. Além disso, utilizamos o Cameleer, juntamente com as suas anotações e o Why3, para verificar que as implementações em OCaml estão corretas e para garantir a robustez do algoritmo.

No final, forneço uma explicação detalhada dos resultados alcançados e clarificarei quais partes do algoritmo foram verificadas com sucesso, destacando as áreas que requerem trabalho futuro. Além disso, descreverei os testes OUnit que realizei para garantir a robustez da minha implementação em OCaml.

Palavras-chave: Concorrência centrada nos dados, Concorrência centrada no fluxo de controle, Prevenção de Bloqueios, Seriabilidade, Verficação Formal

Contents

List of Figures ix

Acronyms xii

1 Introduction 1

1.1 Motivation . . . 1

1.2 *RC^3* . . . 2

1.3 The Problem . . . 3

1.4 Summary of the Contributions and Roadmap . . . 4

2 Background 6

2.1 Basic Concepts . . . 6

2.1.1 Types of Locks . . . 6

2.1.2 Deadlock Solving Methods . . . 6

2.1.3 Serialisability . . . 7

2.1.4 Approaches to program verification . . . 8

2.2 Tools for deductive verification . . . 9

2.2.1 Dafny . . . 9

2.2.2 F* . . . 9

2.2.3 VeryFast . . . 10

2.2.4 Why3 . . . 10

2.2.5 Discussion . . . 11

3 Related Work 12

3.1 Handling Deadlocks in Control-Centric Approaches . . . 12

3.1.1 Deadlock Prevention and Solving . . . 12

3.1.2 Serialisability . . . 17

3.1.3 Annotations . . . 18

3.1.4 Proved Properties . . . 19

3.1.5 Experimental Evaluation . . . 20

3.2 Handling Deadlocks in Data-Centric Approaches . . . 22
3.2.1 Atomic Sets (AJ) . . . 22
3.2.2 Deadlock detection and prevention . . . 24
3.2.3 Annotations . . . 25
3.2.4 Properties . . . 26
3.2.5 Experimental Evaluation . . . 27
3.2.6 Final Remarks . . . 27
3.3 Cameleer . . . 27
3.3.1 What is Cameleer . . . 27
3.3.2 Examples of Cameleer . . . 28
3.3.3 Conclusion . . . 29

4 The Lock Inference Algorithm 31
4.1 Concepts . . . 31
4.2 The Algorithm . . . 33

5 Implementation and Verification 39
5.1 Repository . . . 39
5.2 Concepts . . . 39
5.3 Initialization . . . 42
5.4 Algorithm . . . 47

6 Evaluation 58
6.1 Verification . . . 58
6.1.1 Cameleer maturity . . . 58
6.1.2 Steps to a successful verification . . . 58
6.1.3 Results . . . 59
6.2 OUnit tests . . . 61

7 Conclusions 64
7.1 Contributions . . . 64
7.2 Future Work . . . 65

Bibliography 67

List of Figures

2.1 Two phase locking protocol with the growing phase followed by the shrinking phase 8

6.1 Table of which functions have a verified specification (green) have a specification very close to be accepted or a suggested verification (yellow) don't have a specification (red). 60

6.2 Table of which functions have OUnit tests testing their behaviour. 63

List of Listings

1 Bank Account example annotated with RC3 atomic annotations 4
2 Tree Class in AJ. Annotations: atomicset(a)- Declaration of an atomic set in a class or interface; atomic(a)-Annotation on instance fields and classes. A fieldcan belong to at most one atomic set. Annotated fields can only be accessed from the this reference 23
3 Method copyRoot() execution by Thread T3 and T4 where both of them start by acquiring different trees and then try to execute the method on the tree the other thread has acquired the lock for 24
4 Tree with concurrent access to weight variable of different nodes 25
5 Node implementation with the annotation |this.a<a|-Annotation on variables and constructors. Specifies the order between atomic set a in the annotated variable or constructed object, and the atomic set a in the current object 26
6 First Cameleer example 29
7 WhyML translation OCaml code 29
8 Subsumption example 35
9 Two Phase Locking example 37
10 Resource Type module. Repository Link: resource.ml 40
11 ResourceAccess OCaml Type. Repository Link: resource.ml 41
12 Resource Operation OCaml Type. Repository link: roperation.ml 42
13 Resource Group OCaml Type. Repository Link: resourceGroup.ml . . . 42
14 Backus-Naur Form (BNF) grammar used for the parser 44
15 ComputeResource function of the Mapping Module. Resource Link: mapping.ml 45
16 computeResourceMethodsMap function of the Mapping Module. Resource Link: mapping.ml 46
17 dominatesCheck function of the Resource Subsumption Module. Resource Link: rSubsumption.ml 48

18 logic function of the RSubsumption Module. Resource Link: rSubsumption.ml 49
19 getPair function of the ResourceOperations Module. Resource Link: ResourceOperationsMapGeneration 50
20 labelsVerification function of the ResourceOperations Module. Resource Link: ResourceOperationsMapGeneration 51
21 insert function of the ResourceOperations Module. Resource Link: ResourceOperationsMapGeneration 51
22 addParameters function of the Parameter Grouping Module. Resource Link: parameterGrouping.ml 53
23 CheckConflict function of the Parameter Grouping Module. Resource Link: parameterGrouping.ml 54
24 Iterate function of the two phase lock Module. Resource Link: twoPL.ml 54
25 ropListTrasverse function of the two phase lock Module 55
26 bindingsTrasverse function of the guards Module. Resource Link: guards.ml 56
27 CreateResourceList function of the guards Module. Resource Link: guards.ml 57
28 CheckConflictTest function of the ParaGroupingTest Module 62

Acronyms

AST Abstract Syntax Tree 33

BNF Backus-Naur Form x, 43, 44

CFG Control Flow Graph 16, 17

MRI Method Resource Information 33

1

Introduction

1.1 Motivation

In a single-threaded program, whenever it is executed, the instructions are executed by a central processing unit one at a time. After the first instruction is executed, it jumps for the next one and proceeds in this manner until all the instructions are finished. With the constant evolving of technology, it was obvious that this type of single-threaded programs had a very impactful bottleneck, i.e, the instructions are being executed by one and only one processing unit. In order to improve computation speed and allow programs to have much higher response, parallel computing was invented. In order to solve a problem, it makes use of multiple processing units simultaneously instead of just one. By decomposing the problem into separate parts so that each processing unit executes a part of the algorithm simultaneously, it is able to achieve much better results than previously.

Implementing programs using multiple threads became very popular since it would bring a lot of advantages like improved throughput, better application responsiveness and many others. The use of multiple threads, instead of just one, is a challenge to program and ensure the same properties as in a single-threaded program. When programming with multiple-threads there is a much larger sample of program behaviours that can occur in an execution of a program. Its behaviour can vary from execution to execution making it extremely hard to understand what's the root cause of the problem. Even thought multi-threading has a lot of benefits, the problems that come from it are also a challenge to prevent and resolve.

From all the problems that occur in multi-thread programs deadlock is one of the most famous among them. With multi-thread programs rising in popularity due to their benefits, deadlock became an issue that was not only hard to detect when we're dealing with large systems, but can be also hard to fix.

Deadlock happens when a thread A has a lock $l1$ for a certain memory location while a thread B has lock $l2$ for another memory location and both of them try to acquire the locks the other thread is holding. This means that thread A is trying to acquired $l2$ while

thread B is trying to acquire $l1$ while both of these locks are already being hold by the other thread. Whenever this situation occurs we enter in a deadlock state due to both of them trying to acquire a lock that will never be released. This can happen between a multitude of threads and it's not limit to simply two. As long as they are waiting in a circular dependency, then this is a also a deadlock situation.

In the last decades multiple different approaches and methods were developed in order to prevent, avoid and detect deadlock. As we will see in the state of art, the approaches that I am giving more attention to, are the ones that make use of pessimistic approaches so that their programs can not get in a state of deadlock. For this to work they mostly rely on analysis and graphs to understand what are the locks needed for a critical section and then imposing a linear order to acquire those locks.

Even today, a lot of the approaches to solve deadlocks suffer from multiple issues that go from relying on the programmer to annotate the code correctly to affect the performance greatly due to bad locking policy in high contention programs.

1.2 RC^3

There are two distinct approaches to synchronisation, control-centric or code-centric and data-centric.

A control-centric approach focuses on which part of the code and how it wants threads to access it atomically. Whenever it's needed to constrain an access to a shared memory object, it is required to delimit explicitly which instructions need to operate atomically upon those object. This approach is the essence of control-centric concurrency control.

Control-centric approaches ensure most of the time safety and liveness properties but decentralise concurrency management. This type of approaches have a major drawback, they are vulnerable to the dispersal of bugs related to concurrency. These bugs make extremely hard to reason about the correctness of the code the more it scales. For example, a single error at annotating a lock or unlock, can completely modify an application's behaviour. I will showcase approaches that focus on addressing this issue [20].

In contrast, a data-centric approach focuses on what data does the programmer want to evaluate atomically. It protects groups of data instead of protecting a whole section of the code. These approaches are the alternative that promotes local reasoning instead of a distributed one. However it is a new approach that has some flaws like not guaranteeing progress at all times and it puts a heavy burden on the compiler since it has to infer a lot more information.

Data-centric concurrency control is a rather recent research area. In the existing models, the programmer does not have to reason about the synchronisation of execution flows, but instead about which memory locations share consistency properties. These models rely heavily on several annotations in order to create atomic sets, bind variables to it and other functionalities.

The intention is to provide a solution for data-centric concurrency control using atomic sets that simply relies on the programmer to specify where the atomic sets are and nothing more. He doesn't have to reason about the synchronisation execution flows at all that in the other approaches hinders reasoning and is error-prone with the amount of annotations that it's needed from the programmer.

RC^3 [20, 19] intends on only using one single word that the programmer needs to annotate in its code in order to automatically generate the needed locks for a section that guarantees deadlock-freedom. This annotation may be applied to all variable declarations like class fields, parameters and local variables and also to return type of methods.

The main goal is to ensure that concurrent accesses to atomic resources, that are data items attained from atomic variables in a method, happen in mutual exclusion. For this, in a multi-threaded Java program methods are "units of execution"that, if two of them access concurrently to a resource, the methods need to execute in mutual exclusion.

As an example of this model, in Listing 1 we have a typical Bank with a method `transfers(src, dst, amount)` to transfer money from one account to another. Naturally, these operations must be executed atomically, in order to ensure the consistency of the accounts' balances. By simply assigning accounts to atomic variables in lines 2, 4, 5, 10 (i.e. their declaration as atomic resources), it's guaranteed the atomicity property. Consequently, transfer operations that operate upon the same must accounts execute in mutual exclusion. An execution,disregarding the amount, like `transfer(acc1, acc2)` and `transfer(acc1, acc3)` will not happen concurrently because they share and modify `acc1`.

1.3 The Problem

The RC^3 model needs to be encoded into a low-level concurrency control primitives from its high-level abstraction in order to perform the necessary algorithms. It starts as a normal Java coded program and it's the type system the first to analyse it. The type system has a very important mission that is to verify if the mutable objects that have been assigned with the Atomic annotation are being assigned to other annotated variables. It is not possible to assign a primitive type variable to an Atomic annotated one and vice-versa. To accomplish this, an atomic type of each was primitive type was created.

With this new atomic type, strong atomicity and absence of data-races can be ensured. Whenever there's a missing annotation on what should have been an atomic variable, that will trigger a compilation error because the program will only compile when atomic variables are assigned to other atomic variables. Data-race freedom between two concurrent method execution can be achieved with semantics from the type system. After being analysed by the type system, this code is compiled into java byte code where the points to analysis will occur.

Related work will serve as a background of what points-to analysis were developed in order to infer the resources accessed inside the method and what is the right locking

Listing 1 Bank Account example annotated with RC3 atomic annotations

```
class Bank {
  Map<Integer, @Atomic Account> accounts;

  void transfer (int src, int dst, float amount) throws OverDraftException {
    Account destAccount = accounts.get(dst);
    Account srcAccount = accounts.get(dst);
    srcAccount.withdraw(amount);
    destAccount.deposit(amount);
  }
  float balance (int accountNumber){
    Account acc = accounts.get(accountNumber);
    return acc.getBalance();
  }

  class Account {
    float balance;
    void deposit (float amount) ( ... )
    void withdraw(float amount) throws OverdraftException( ... )
    float getBalance()( ... )
  }
}
```

policy given that information. RC^3 points to analysis will have several steps that goes from getting resource information and finding where are the critical sections of the code that needs concurrency control, to inferring the right locks in order to maintain the properties while having a good performance. This analysis is detailed in chapter 4.

This analysis needs to make sure it maintains properties such as deadlock-freedom, serialisability, starvation-freedom and atomicity. Making a program deadlock-free is very simple if we coarse-lock a big portion of the program. Although this type of solutions guarantee correctness, it will induce overhead in the program whenever there's a lot of contention, hindering its performance. The same can be said for all the other properties. There's a limit in how much time a user is willing to wait for a certain action to occur, this is why there's a need for a fast response time of programs.

It is not very difficult to create a program that respects this properties. However, is it possible to create a locking policy strategy that respects this properties and at the same time having a performance comparable to fine-tuned locked programs?

1.4 Summary of the Contributions and Roadmap

RC^3 model has already an implemented points-to analysis for lock inference and a strategy for deadlock prevention. Although this algorithm already demonstrated a good performance, there's still no insurance that it respects and maintains all the properties.

This thesis contributes to the process of verifying that the model respects and maintains the properties mentioned above since without a formal proof of its properties, there's no way to formally say that the algorithm guarantees all of them.

To sum up, the contributions of these thesis are as follows:

- A defined OCaml implementation of the start (receiving and processing the input) and the core RC^3 functions that generate the output for the creation of the graph.
- A Cameleer specification that captures the behaviors of the implemented modules either in their entirety or focuses on the most important functions within those parts, ensuring their logical correctness
- A series of OUnit tests for the crucial functions that could not be successfully specified using the Cameleer specification.
- Cameleer has made significant progress in terms of maturity as a verification language. During this thesis, Cameleer has become more reliable and ready to be used in larger and more complex programs that involve advanced data structures.

This thesis starts with an overview of basic concepts (Section 2.1) that are important to understand in order to make the study of the concurrency control approaches mentioned below easier. We also go through several tools for verification, in Section 2.2, that could have been used and discuss the reason of the choice that we we did.

In the state of art we start by doing an analysis of several control-centric approaches in Section 3.1 and how each of them differs from each other discussing how they guarantee or not important properties such as serialisability or deadlock-freedom. Afterwards, I go explain in-depth a data-centric control approach in Section 3.2 and what properties does it guarantee.

In Chapter 4, I introduce the RC^3 model starting by important concepts in Section 4.1 that are required for the better understanding of the algorithm behind it. After covering the basic concepts, I explain the algorithm step-by-step in 4.2 and provide several examples that help in understanding the goal and how each phase functions.

Next, I explain how the OCaml implementation of the algorithm was executed. This chapter follows the same order as Chapter 4 and explains how the concepts were implemented in OCaml in Section 5.2. The implementation and verification in Cameleer of the functions is then explained in detail with code examples. It is done a separation of the initialisation of the OCaml algorithm that is detailed in Section 5.3 and the algorithm related to the RC^3 model itself explained in Section 5.4.

Chapter 6 presents a critical analysis of the results obtained. I take a look at which parts of the algorithm were successfully verified and what tests were executed.

To conclude, Chapter 7 is dedicated to what contributions did this thesis achieve and what future work follows it.

2

Background

This section is dedicated to explain the key concepts for the development of our project, in what situations can they occur and why are they relevant for the project that we will be developing.

2.1 Basic Concepts

2.1.1 Types of Locks

A mutex (a shorthand for MUTual EXclusion object) is an object whose state can be either locked by a unique thread, a sequential processes that share memory, (the owner of the mutex) or unlocked (or free). A thread can become the owner of a mutex by acquiring it, i.e., using a blocking operation that waits until the mutex is free and returns after actually taking it. When a thread acquires a mutex, it can execute the desired operations over the memory locations protected by the mutex that it is now holding. A thread waiting for one or more mutexes is blocked and it is making progress if it is not blocked [7].

Reader/Writer Locks are a different type of locks that take in consideration the intention of the thread regarding the memory location that is accessing. These locks extend mutexes by allowing a lock to be acquired by n readers or by 1 writer. They can significantly increase available concurrency for programs accessing mostly read-only data structures. For instance, a hash table protected by a global read/write lock might allow concurrent look-ups [17].

2.1.2 Deadlock Solving Methods

Three groups of methods to solve the problem of deadlocks have been identified long ago [13]: deadlock prevention; deadlock avoidance; deadlock detection and recovery.

Deadlock prevention is a strategy to infer and detect critical sections of the code where deadlocks are very likely to happen. After its known what are the zones in the code that can induce the program in a deadlock state, it's then possible to use an algorithm to prevent that from happening instead of letting it happen and solving it afterwards. Deadlock prevention is usually the best approach when dealing with programs with very

high contention. This is the main approach that I will studying in the works presented below.

Deadlock avoidance is a dynamic form of deadlock prevention, where the run-time decision to enter a critical section, by locking a mutex, is deferred, based on the presumption that this could lead to a deadlock in the future. Deadlock prevention and avoidance techniques both reduce the parallelism in the program, by disallowing some paths that could otherwise be taken. Of course this reduction should be as minimal as possible. For deadlock avoidance, the prediction that a locking decision might lead to a deadlock should be as precise as possible and computing it as fast as possible.

Deadlock detection and recovery is another approach to solve deadlocks. Contrary to deadlock prevention, there's no previous analysis to find where deadlocks are possible. Instead, this approaches have protocols to scan the program at run-time and find out if there's threads in deadlock. If it's detected a deadlock situation then, it's executed a protocol that usually reverts this situation and lets one of the threads that were deadlocked progress at the cost of the other. There's multiple different implementations to solve deadlocks in an optimistic way.

2.1.3 Serialisability

Properties like serialisability are very important to maintain in multi-threaded programs. According to [8], an execution of read and write events performed by a collection of threads is serialisable if it is equivalent to a serial execution, in which each thread's transactions (or atomic sections) are executed in some serial order. A serialised schedule provides a criterion for correctness of critical section execution.

Group locking locking protocol where a thread acquires all the locks before entering the critical section and only starts executing it, when it has in its possession all the necessary locks. Only after finishing the execution of the code block does it release the locks. By using a group locking protocol we are guaranteeing the serialisability property inside the critical section.

Two-phase locking, Figure 2.1, is another locking policy that guarantees serialisability and is as follows. As a process executing a critical section proceeds from operation to operation, it acquires the shared resources it needs along the way. This is called the "growing phase". The set of resources held is constantly increasing since a process must not release any resources as long as there's more resources to acquire. Once any resource is released, no others can be acquired and the set of held resources is constantly decreasing. This is called the "shrinking phase". The conceptual "instant of time"ta at which the action occurs can be regarded as the time at which the first resource is released. This protocol of resource acquisition and release guarantees that actions execute in a serial order [15].

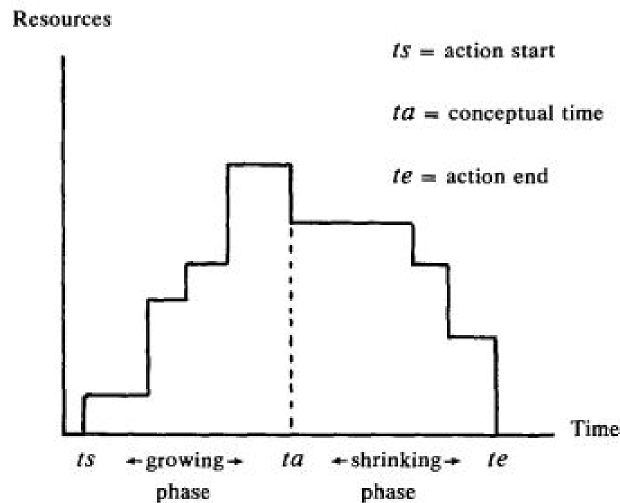

Figure 2.1: Two phase locking protocol with the growing phase followed by the shrinking phase

2.1.4 Approaches to program verification

In order to prove deadlock-freedom, serialisability and other properties, it is essential to make use of program verification. There are three important types of program verification: static, dynamic and deductive.

2.1.4.1 Static and Dynamic

Automatic verification of software can be done using a static or a dynamic approach [18].

Static verification can be defined as a group of algorithms and techniques applied to a representation of the program's code, that we want to analyse, without executing it. Static analysis is an automated process and, usually, it's used during the development phase in order to find errors in the program. Finding these errors is the main objective of static verification. Static analysis tools use a variety of methods that allow us to identify vulnerabilities in code. The most usual method is the use of type systems. However, static analysis is not perfect, since some of the properties it tries to check are undecidable.

Differently from static verification, dynamic verification is done while the code is executed. The focus of this verification is usually to make sure the behaviour of the program is correct, if the memory and cpu usage is the desired and if the performance is up to standards. Dynamic analysis works in two ways: integrating the code into a build-time application or providing a form of platform emulation to understand the internal behaviour of a run-time application. As the size and execution time of the program increases, the performance of dynamic analysis tools degrades, because the construction of the necessary models is a task that scales with the software dimension and run-time, reaching dimensions that do not fit in memory.

2.1.4.2 Deductive

Deductive verification comes from formal verification that is the act of proving or disproving if the algorithms of a program are correct with respect to a certain formal specification or property. This type of verification can be very useful whenever we're dealing with systems that have a mathematical model and that we want to make sure our implementation is respecting the rules of this model.

Deductive verification aims at checking behavioural properties of programs, where behaviours are formally specified by logical annotations: function pre- and postconditions, code assertions, loop invariants, global invariants, etc. On the bright side, deductive verification is very powerful: it is modular (each function can be verified independently), it is guaranteed to be sound and it allows describing complex behaviours of programs. On the dark side, annotations must be added manually by the programmer, which can be a very hard task [10] .

2.2 Tools for deductive verification

There are multiple tools in order to verify programs with deductive verification. In this section I go over some of them and why I decided to choose Why3.

2.2.1 Dafny

Dafny [14] is a program verification tool which includes a programming language and specification constructs. The Dafny user creates and verifies both specifications and implementations. The Dafny programming language is object-based, imperative, sequential, and supports generic classes and dynamic allocation. The specification constructs include standard pre- and postconditions, framing constructs, and termination metrics. Pre- and postconditions state what properties have to be true at method entry and exit, respectively, and are used to construct the specifications or contracts of methods.

The method callers must establish preconditions and can assume postconditions. The method implementers can assume preconditions and must establish postconditions. Dafny programs are statically verified for total correctness, i.e., that every terminating execution satisfies its contract and that every execution does indeed terminate. Dafny's program verifier works by translating a given Dafny program into the intermediate verification language Boogie in such a way that the correctness of the Boogie program implies the correctness of the Dafny program.

2.2.2 F*

F* [1] is a full-fledged programming language with a powerful dependent type system that supports user-defined effects for stateful code.The F*type-checker can prove that programs meet their specifications using a combination of SMT solving and interactive

proofs. F* programs can be extracted to efficient C, WASM, OCaml, F#, or ASM code after being verified. This allows verifying the functional correctness and security of realistic applications.

Its type system has dependent types, refinement types, and a weakest precondition calculus. These features are very important when it comes to express precise and compact specifications for programs. Using the SMT solving and interactive proofs its now possible for the F* type-checker to prove that programs meet their specifications.

2.2.3 VeryFast

VeriFast [11] is a tool for modular formal verification of correctness properties of C and Java programs annotated with preconditions and postconditions written in separation logic. In order to express complex specifications, the programmer can define predicates in separation logic, inductive datatypes, primitive recursive pure functions over these datatypes. In order to detect errors, VeriFast symbolically executes the body of each function of the program, starting from the symbolic state described by the function's precondition, checking that permissions are present in the symbolic state for each memory location accessed by a statement, updating the symbolic state to take into account each statement's effect, and checking, whenever the function returns, that the final symbolic state satisfies the function's postcondition. To verify some specifications, the programmer can also creates functions that serve only as proofs that their precondition implies their postcondition, they are called lemmas. The verifier also checks that lemma functions terminate and do not have side-effects [12].

2.2.4 Why3

Why3 [6] is a verification language that provides a rich language for specification and programming (WhyML) and makes use of external theorem provers to discharge verification conditions written by the programmer. Why3 comes with a standard library of logical theories and basic programming data structures like Boolean operations, sets and maps, arrays, queues and others. More complex structures will make use of this logical theories in order to complete their verification specifications. WhyML is used as an intermediate language for the verification of C, Java, or Ada programs. WhyML strives to be comfortable as a primary programming language and inherits numerous high-level features from ML.

The specification language of Why3 can serve as a common format for theorem proving problems, that can use multiple automated and interactive provers, such as Alt-Ergo, CVC3, Z3, Coq and others. When a proof obligation is dispatched to a prover that does not support some language features, Why3 applies a series of encoding transformations to, for example, eliminate pattern matching or polymorphic types .

Why3 also has an IDE that is very easy to use and interactive that allows the programmer to view all the code, what Why3 is trying to prove, what are the problems and in that

case, the tool provides counter-examples if it cannot prove it.

2.2.5 Discussion

When we compare them with each other, only Why3 relies on multiple theorem provers. This is an important difference from Why3 to other verification languages. With Why3 we can use a lot of different external theorem provers and make use of them as mean to compensate for each others' weaknesses. When other languages might struggle in proving certain aspects or structures, Why3 has the option of relying on provers specially designed for those instances which makes the verification process much simpler. Also, in Why3 IDE, the programmer has control over the verification steps and is allowed to divide those steps, call different provers and other functionalities that make the verification easier to accomplish. These are the main reasons for why I am using Why3 over the other verification tools.

3

Related Work

In this chapter I shall be presenting a review of related work in the area of, explaining the different approaches that each of them take and comparing them with each other so we can understand what are the benefits of each of them and what we're able to learn from the results obtained in each of the approaches. Inside each property, each subsection will correspond the different approaches that were studied. Also I will be doing a deeper analysis on one of the works that has very similar concepts to what I will be developing. The table in 3.1 summarises the approaches studied and their specifications. I will go into more detail in each of those specifications.

3.1 Handling Deadlocks in Control-Centric Approaches

In the control-centric concurrency approaches studied, the majority make use of atomic sections in order to protect a certain region in the code. Atomic sections allow programmers to give a high level specification of the concurrency semantics. Without any intervention from the programmer, the underlying system enforces that atomic sections, i.e. sections of code protected by an atomic keyword, appear to be executed atomically. Under strong atomicity semantics these sections appear to be atomic with respect to any other statement in the program. Whereas under weak atomicity semantics atomic sections appear to be atomic with respect to other atomic sections in the code. That is, for any program execution trace, there must exist an equivalent trace where all atomic sections are executed in some serial order. For programmers this is a very attractive alternative, since all the difficulties of manual pessimistic concurrency are abstracted away [3].

3.1.1 Deadlock Prevention and Solving

Even though there is a lot of diversity in how each work approaches deadlock prevention, there is also similarities in the ideas presented. In this section we will study how deadlock prevention is done and what are the possible drawbacks and benefits of the approaches presented.

Table 3.1: Summary of the specifications of the approaches studied

Algorithm	Control or Data-Centric	Type of locks	Deadlock strategy	Serialisation strategy	Annotations	Proved Properties	Tools used for verification	Experimental evaluation
Lock Allocation [5]	Control-Centric	Mutex	Prevention	Group Locking	X	X	X	Unnecessary coarse-locking
Lock Inference [9]	Control-Centric	Mutex	Prevention	Group Locking	X	X	X	Cases of unwanted overhead
Inferring Locks [3]	Control-Centric	Read/ Write	Prevention	Group Locking	Starvation-freedom	Their own language	X	Cases of unwanted overhead
Auto-locker [17]	Control-Centric	Mutex	Prevention	Two-Phase	Performance	Deadlock-freedom and starvation-freedom	LockC	Slower than manual versions
Jthread [7]	Control-Centric	Mutex	Prevention and Avoidance	Group Locking	Performance	Starvation-freedom	Their own language	Same as pthread
Discrete Control [23]	Control-Centric	X	Prevention and Avoidance	Control Logic	Ambiguity Clarification	X	X	Good performance compared to an atomic-section

A great majority of the works studied makes use of atomic sections as their control tool for a critical section. These type of approach asks the programmer to annotate the code to identify where are the regions where concurrency needs to be controlled. When these regions are annotated there's usually an algorithm that infers statically a set of locks for each atomic section so that each thread, before entering them, must acquired all the locks necessary to execute that critical region. Until the thread has in its possession all those locks, it is not possible for it to progress and it will wait for them to be released.

How is deadlock guaranteed? Some of these approaches developed algorithms to create a linear order for threads to acquire locks, i.e, if two threads want to acquire the same group of locks, there will never be a situation where thread A starts by acquiring a different lock than thread B. There are different ways to create this locking order. There's also approaches that use other algorithms and graphs in order to avoid deadlock. I will now describe these approaches in detail to showcase how do they deal with deadlock situations and how they solve it.

Lock Allocation [5]. In Lock Allocation they rely on a static flow insensitive points-to analysis and dependency relation to infer the locks needed for each atomic section. To obtain the locking order, they make use of the accessed before relation that specifies what are the data dependencies in the atomic sections. They say there is an accessed before relation between two values or variables if, in the control-flow graph, there is a path where one of the values is accessed, then modified and only after is the second value accessed. By traversing the graph they can obtain this approximation of the accessed before relation. If

the graph is acyclic then they can order the locks linearly by a topological sort of the access graph. If there is cycles in the graph, they need to order the strong connect component of the graph linearly in order to avoid deadlock between threads. This work also mentions how they deal with nested atomic sections. Atomic sections may transitively call methods containing other atomic sections. A lock obtained in a nested atomic section cannot always be safely released at the end of that section because the same lock may be re-obtained by another nested section under the same parent, potentially violating atomicity. To avoid this issue, they use a two-phase locking discipline. Locks are obtained when the associated value is first accessed, but are not released until the outermost atomic section finishes.

Lock Inference [9]. In order to prevent deadlock this approach makes use of an algorithm that efficiently infers a set of locks that should be acquired for each atomic section. The key part is determining what are the memory locations that the threads could share between them and use this information to generate the locks for the program. In this analysis they map each pointer to a certain abstract name that represents the memory pointed to at run-time. A lock is then created for each abstract name and its used to guard the access to that run-time memory location. To ensure they don't have two threads acquiring different locks and then deadlocking they also created a total order between each lock so that they are statically acquired according to this. The way they do this is by defining the meaning of "Dominates"and an algorithm called "Mutex Selection". The definition says that a memory location P dominates an access to a location P2 if every atomic section containing an access to P2 also contain an access to P. If we find a memory location P dominating a location P2 then we can protect both of them both with P's lock. Using this definition its computed the relationship between all the memory locations access and this is used to then compute a set of locks to guard shared locations using the Mutex Selection Algorithm. In each step of this algorithm, they slowly create the ordering in which threads will have to acquire locks to enter each specific atomic section. Then, they pick the total ordering in all the locks, insert code that makes sure thread acquires the locks in order, perform their actions and then releases all the locks.

Inferring Locks for atomic Sections [3]. This approach makes threads acquire all the locks at the entry of an atomic section, similar to some works we saw before. The compiler makes an analysis that estimates which memory locations are accessed inside a critical section and creates locks in order to protect those regions in the memory. They also do an analysis on which type of locks should be introduced based on the memory locations accessed. Sometimes it may be safe to introduce fine-grain locks but other times it may resort to a more coarse grain locking strategy. Their compiler always tries to do a locking policy as fine-grained as possible in order to make the program more efficient in case there is a lot of contention. With multi-grain locks there are pairs of locks that cannot be held concurrently. Hence, acquiring multigrain locks in any linear order does not guarantee

that locking is deadlock free. To support multi-grain locks, they implemented a library that makes use of three functions in order to prevent deadlock: to-acquire, acquire-all and release-all. They provide the library with the relevant portion of the locking structure and then it inserts these functions into the right spots in order to prevent deadlock situations.

This approach also makes distinctions between operations of read and write and applies different locks according to what is being done. To achieve this, it makes use of the intention mode that introduces tree new types of locks: intention to read, intention to write, and read-only with the intention to write some child nodes. With this, it is possible for threads to access memory locations concurrently depending on the mode they want to operate since some modes might not be compatible with each other.

Autolocker: Synchronization Inference for Atomic Sections [17]. Autolocker is an approach that relies on analysing the program and understanding the right locking policy that is free of deadlocks.

Their type system generates computation histories that creates a summary of the synchronisation-related behavior of a program. This history guides autolocker in how to place locks in atomic sections. It can also be used to check if a program might or not deadlock and if it accesses data without the proper lock. The developers followed a stric two-phase locking policy that ensures threads will only release the locks whenever they are out of that specific atomic section ensuring consistency in the data but also having a disadvantage whenever they deal with structures such as Trees. When they are traversing a tree in a top-down order, autolocker does not let a thread release the lock on the parent node whenever it acquires the child node. Since there is no way for a thread to deadlock in this scenario their implementation is obviously not ideal since if there's a lot of contention the program efficiency will suffer a lot.

In order to prevent deadlock Autolocker has an order in which threads need to acquire locks before entering an atomic section. By analysing the data dependencies between locks that form an order on the locks IDs, the program creates an order in which locks should be acquired. If it is unable to order locks in a way that guarantees freedom of deadlock it rejects the program. It's possible to solve this problem by replacing the offending locks with global ones due to them being easier to analyse. Then, the partial order in each atomic section is converted into a total order using a topological sort. After having this order, the algorithm only has to make sure the lock acquisitions are placed throughout the program in the order that was inferred.

Jthread [7]. This work created a library that ensures that when a program uses it, it will never run into a deadlock.To achieve this result, they first defined what a region is. Jthread mutexes are grouped into regions. Each mutex can only belong to one region and the programmer has the ability to choose which region does a mutex belong to. A new mutex by default has it's own new region if it's not created inside a already created region. Then, the developers dynamically create an ordering between all the regions

present which is used to make sure there are no circular mutex locking when a thread tries to lock mutexes from different regions. To create this ordering they resorted to an algorithm where all the threads have a stack of regions and all the threads are initialised in a region that it's greater than all called "Top". When a thread wants to enter a region, the algorithm starts by checking if the locks that it needs to acquire to enter a region really belong to it. If this is true and the region the thread is currently is greater than the region it wants to enter, then the thread pushes the region into its regions stack. When the thread finally enters the region, a global order is updated creating a relation between the region this thread was and the region it is currently, mapping the regions this way. This approach also creates a solution for potential starvation when threads try to acquire mutexes. To do this, it's created a global queue that places threads depending on an order relationship. This queue lets threads acquire its locks whenever there's no thread ahead of it in the queue that desires to acquire at least of the same locks. In this way we can make sure a thread has progress and is does not suffer starvation which is a property not all these works are guaranteeing.

The Theory of Deadlock Avoidance via Discrete Control [23]. Their strategy is to avoid deadlock through a combination of offline static analysis and run-time execution control: They start by extracting per-function Control Flow Graphs (Control Flow Graph (CFG)) from program source code. Then, they enhance the graph to facilitate deadlock analysis by including information about lock variable declarations and accesses. After they have the CFG enhanced, they translate it into a Petri net model. The model includes locking and synchronisation operations that are constructed in a way that deadlocks in the original program correspond to structural features in the Petri net. These features in the Petri net model are named siphons. Siphons occur when there is a state where the Petri net cannot find progress and it's paths can not progress. To solve this, they augmented the Petri net obtained, with features that will guarantee deadlock avoidance in the original program. These features are control places with incoming and outgoing arcs to transitions that delay resources acquisitions when there's more than one path to obtain it. The developers concluded their solution can eliminate deadlocks from the given program without introducing new deadlock situations or a bottleneck in the performance. If by any chance deadlock avoidance is impossible for a given program, then they issue a warning explaining the reason why they can't avoid the deadlock and terminate the program.

Discussion

After studying all these approaches, besides [23], there was a clear dominance of static analysis inferring a set of locks that a thread needs to obtain before entering an atomic section or region. After having the locks inferred, in order to prevent deadlock they created a total lock order so that threads by respecting it do not risk deadlocking with

other threads also trying to obtain them.

I also analysed Jthread [7] which implemented a mechanism to counter thread starvation when trying to obtain the locks to enter a specific atomic section by using a scheduling mechanism to infer which one is gonna have access to that specific lock. Most of the works studied did not mention mechanisms to counter potential starvation and only assumed this occurrences would be very rare or non existent.

In some approaches the developers had the care to try to understand what are the best locking granularity for a certain code block like in Inferring Locks for atomic Sections [3] while others leave that for the programmer to think about and decide himself what should be the more efficient granularity for an atomic section [17]. This two type of approaches vary a lot in the results depending on who is annotating the programs. If a programmer can annotate a program very efficiently in order to obtain a good locking strategy and improving the program performance, then the Autolocker solution can have a lot of benefits attached to it. In contrast, it can be very sub-optimal if the programmer ends up coarse-locking most of the program and inducing, under high contention, a not desirable performance.

It was also common to see the use of graphs, that even thought they differ from each other in some specifications,are used to map the program in order to identify possible instances of deadlocks in them by finding cycles and circular dependencies in the acquisition.

The Theory of deadlock avoidance [23] also had a very interesting approach making use of CFG and its features to understand where are the potential deadlock situations and preventing them by augmenting the graph with control places that delay resource acquisition. This approach was slightly different from the other approaches studied.

3.1.2 Serialisability

Regarding serialisation, I will analyse if all the approaches that use atomic sections, regions or their group lockings respect the serialisation property and how they achieve it. The two widely used protocols in order to respect serialisability were the two-phase protocol and the group locking protocol. While in the two-phase protocol we have threads acquiring locks at the same time they are executing, in the group locking protocol, a thread will not progress into the critical section until all the locks are acquired.

Two-Phase Locking. Some approaches [23, 17] made use of two-phase locking as their policy to ensure serialisabity.

In Theory of Deadlock Avoidance [23], in order to do a lock acquisition transition that needs to be controlled, the control logic must check a token availability. They replaced the native lock-acquisition function with a wrapper to implement the required test for those transitions. The wrapper internally uses two-phase locking with global ordering on the set of control places to obtain the necessary tokens.

Autolocker [17] also uses a policy of strict two-phase locking where all locks are released at the end of atomic sections and not before. Thus, Autolocker guarantees that data remains consistent at all times.

Lock Allocation [5] made use of two-phase locking specifically whenever inside an atomic section, they called methods containing other atomic section. A lock obtained in a nested atomic section cannot always be safely released at the end of that section. That same lock may be re-obtained by another nested section under the same parent, potentially violating atomicity. To avoid this issue, they use a two-phase locking discipline. With this locks are obtained when the associated value is first accessed, but are not released until the outermost atomic section finishes.

Group Locking. All the other approaches made use of group locking to protect code blocks by making sure that every thread before entering it has to acquire all the locks corresponding to the memory locations inside it. After it acquires the locks, only the thread with those locks acquired can enter the atomic section and until the locks are released, no other thread will execute the atomic section. This effectively serialises the whole critical region.

Discussion

Since most of the works studied used a lock inference algorithm followed by imposing an order in the lock acquisition before entering an atomic section, they had to use a Group Locking policy. There was only

3.1.3 Annotations

There were three types of annotations that have different impact on how these algorithms behave.

Performance. Autolocker [17], in contrast with other approaches, guarantees that a program is deadlock free and handles many of the tedious and error-prone details of parallelism but it lets the programmer have control over the efficiency and performance of the program with annotations. Even thought it's the compiler that handles the management of the set of locks, the programmer also needs to write annotations that connect locks to shared data but in a simpler way than manual locking the entire program. With this approach, even though the programmer has to write annotations and has more responsibility than in other approaches, Autolocker still makes sure that the likelihood for a programmer annotation to induce an error is very low.

Jthread [7] also has an unsafe mode that unchecks rules the programmer would have to follow, otherwise it would throw an error. Without these checks, the programmer can improve the performance of the program by annotating the regions in a way he finds to

benefit the program efficiency. The authors recommend that this mode should only be used after the program is completely tested and debugged.

Ambiguity. Annotations were also used to help clarify the ambiguity caused by dynamic constructs that may lead to wrong interpretations and false positives. Appropriate user annotation alleviates problems when doing the control synthesis in [23] algorithm in large programs where numerous false control flow paths could result in very conservative control logic and hindering the performance of the program. Not only this work allows the programmer to help the algorithm understand structures where it can not infer a good locking policy or even may end up thinking there's a deadlock situation where it's not true.

Discussion

There was clearly different uses regarding annotations and what the developers wanted the user to have control upon. One of the approaches would not only give the programmer the responsibility of indicating where the atomic sections are but also would ask him to define what would be the desired locking granularity for the corresponding critical section. Some works required programmers to help the compiler understand where there was no risk of deadlock avoiding then situations of false positives.

3.1.4 Proved Properties

Throughout these approaches, it was very rare to see formally proved properties and verification tools being used. However, a few of them proved some important properties that it's worth it to showcase and others simply guarantee these properties without formally proving them.

Deadlock-freedom. This property is the most important property to prove. All of the approaches studied guarantees that their algorithm for solving deadlocks is correct and guarantees deadlock-freedom. However, there's some works that go into more detail by proving their properties. From all the works studied [17] decided to formally prove this property which is an insurance of the correctness of their algorithm.

Starvation-free. This property is very important in the context of deadlock prevention. Even thought most of the approaches can prevent deadlocks, most of them do not tackle the issue of a thread being starved while trying to acquire the necessary locks in order to enter an atomic section. Only [7] and [17] formally proved that their approach is indeed starvation-freedom.

I will also showcase what were the tools and languages that the developers of which work use to prove the correctness and soundness of their implementations.

Inferring Locks for atomic Sections [7]. The developers decided in this work to create their own input and output language with semantics that allow them to define rules, definitions and other semantics that they need to prove the correctness of their algorithm. The language contains standard constructs such as heap allocation, assignments, control-flow constructs and atomic sections. Expressions include variables, dereferences, address-of variables, offsets, allocations, and null values. All values in this language are locations or null.

Autolocker: Synchronisation Inference for Atomic Sections[17]. It is used LockC as a language to formalise Autolocker and to define statements, expressions, definitions and rules of the type system.

Jthread [7]. Jthread also makes use of its own language to prove its properties. In their language, the only values are mutexes, sets of mutexes, and unit, and the only operations are structured locking and the launching of a thread. This language only makes sense as part of a larger language featuring general-purpose constructs such as functions.

Discussion

From all the works studied only a couple either created their own formal language to establish their rules and definitions or used simple logic to explain the correctness and soundness of the properties they intended. The amount of approaches doing formal proofs of properties is very low and it would be important to show that the properties they are implying their algorithm has, are formally proven and correct.

3.1.5 Experimental Evaluation

Depending on what test were done by the developers, in this section I will analyse what limitations in terms of performance do these programs have, what are the downsides when doing deadlock prevention and what can be accomplished by changing the granularity of the locking policy of a program.

Lock Allocation [5]. In one of the experimental test using an hash table example, the developers mentioned that their protocol had a rough time understanding the data structure and what's the best locking policy for it. Since it couldn't fully grasp when it could do finer locking without compromising the occurrence of deadlock this protocol end up locking the whole structure and making the program slower in case there's high contention for that atomic section when it was not necessary. They also tried their implementation in a source web-server. The results showed more that their implementation scaled well with the size of the program than that it was doing a good locking policy. This was due to the fact that the server had simple locking assignments that with a simple locking scheme it was easy for their implementation to ensure atomicity and a good efficiency.

Lock Inference [9]. Lock Inference didn't present any experimental evaluation since they were still in the process of implementing their algorithm. However, the developers discussed that there is cases of unwanted overhead in their algorithm just like in Lock Allocation [5], for example, whenever we consider a linked list with atomic sections per node of the list, this protocol was not be capable of locking each node but instead would create a global lock on the list to guard all accesses to the nodes. This would have a significant impact in the efficiency of the algorithm whenever different threads would try to access data inside the list that would never cause any deadlock problem but now one of them will have to wait for the other to finish whatever operation it is doing in the list and release the lock on it.

Jthread [7]. In the tests performed, this library was compared to pthread in order to understand how would its performance would fair against it. The tests showed that jthread has significant slower performance than pthread when we're just locking and unlocking. In an Hop server both of these showed no clear difference in their response times which is very positive since there is a lot of benefits by using jthread as explained in the previous section. There was also some micro-benchmarks that jthread showed a slightly better performance due to its efficient deadlock avoidance algorithm that leads to deadlock when implemented with pthread.

Inferring Locks for atomic Sections [3]. In some of the benchmarks tested, their solution had a negative impact in the performance since the analysis for lock inference ended up creating too many fine grained locks that cause overhead when locking and unlocking too much, hindering the performance of the program. The opposite effect was observed in some micro-benchmarcs. In one of the benchmarks it was clear that there was a benefit in tracking read and write effects and differentiating the locks. Also, their adjustment of the locking granularity had a good impact in the performance.

Autolocker: Synchronization Inference for Atomic Sections [17]. In this work Autolocker was compared with manual locking and two recent software transactional memory implementations in a highly concurrent hash table and a FIFO queue with less potential for currency. In the Hash table test, autolocker had a slightly worse performance than manual locking due to the overhead of Autolocker run-time that needs to keep track of the set of locks each thread has. In the FIFO queue autolocker was also slower than manual locking but was 1.6x to 2x faster than the transaction manager software.

The Theory of Deadlock Avoidance via Discrete Control [23] In a web server test done by the developers they concluded that their deadlock avoidance implementation has a negligible impact on light-load response times. This approach is directly compared with an atomic-section based work and they achieve way better performance than the counter

part. They conclude that sometimes locks permit better exploitation of available physical resources than atomic sections.

Discussion

It's not easy to do a perfect program that solves deadlocks with no impact in the performance. A lot of these approaches have a difficult time analysing some structures and getting the best locking policy for them. Sometimes they have to resort to coarse lock when it's not needed like in the case of a linked list. Jthread [7] showed that they could create an algorithm that solves deadlocks and decides the locking policy that has roughly the same or even slightly better performance than the pthread library. It's also important to highlight that in Inferring Locks for atomic Sections [3], their implementation of having different locks depending if it is a read or write operation contributed to a better performance which was something that no other work mentioned.

3.2 Handling Deadlocks in Data-Centric Approaches

In this section I will analyse with more detail an approach [16] that uses AJ annotations to support data-centric concurrency control and understand how they solve deadlock problems, what annotations do they require from the programmer and other system specifications.

3.2.1 Atomic Sets (AJ)

AJ [4] extends the syntax of the Java programming language with annotations needed to support a data-centric programming model. An AJ class can have zero or more atomic set declarations. Each atomic set has a symbolic name and intuitively corresponds to a logical lock protecting a set of memory locations. Associated with each atomic set is a set of units of work, code fragments that, when executed without interruption, preserve the consistency of their associated atomic sets. AJ assumes that all non-private methods of a class are units of work for the atomic sets it declares [22].

Since it is very easy to refactor Java programs to AJ without losing any performance, in [16] the developers decided to use AJ in order to present an algorithm for detecting possible deadlock in AJ programs by ordering the locks associated with atomic sets.

The authors presented a tree example to illustrate how this approach works (Listing 2). One issue that this tree can have, if it's not ensured a good concurrency control, is a data-race in the calls of the method incWeight() and compute() where the compute method might return an obsolete value . To counter this, an atomic set is declared and both the root and the size are included in this atomic set. Due to this, at run time, each Tree object will have an atomic set instance t containing the corresponding fields. The AJ compiler will now insert locks to ensure that different units of work execute atomically. In order to protect all the elements of a Tree including all the Nodes inside it, there is the aliasing

Listing 2 Tree Class in AJ. Annotations: atomicset(a)- Declaration of an atomic set in a class or interface; atomic(a)-Annotation on instance fields and classes. A fieldcan belong to at most one atomic set. Annotated fields can only be accessed from the this reference

```
class Tree {
    atomicset(t);
    private atomic(t) Node root|n=this.t|;
    private atomic(t) int size = 1;
    Tree (int v)            { root = new Node|n=this.t|(v);;}
    int size()              {return size;}
    INode find(int v)       {return root.find(v)}
    void insert (int v)     {root.insert(v); size++;}
    void insert (int v)     {rerturn root.compute()}
    void copyRoot(Tree tree) {tree.insert(root.getValue());}
}

interface INode {void incWeight(int n);}

class Node implements INode{
    atomicset(n);
    private atomic(n) Node left|n = this.n|;
    private atomic(n) Node right|n=this.n|;
    private atomic(n) int value, weight = 1;

    Node(int v) { value = v; }
    int getValue() {return value;}
    void insert(int v){
        if (value==v) weight++;
        else if ( v < value) {
            if (left ==null) left = new Node|n=this.n| (v);
            else             left.insert(v);
        } else {
            if (right ==null) right = new Node|n=this.n|(v);
            else              right.insert(v);
        }
    }

    public void incWeight (int n) { weight += n; }
    INode find(int v) {
        if (value==v) return this;
        else if (v < value) return left == null? null : left.find(v);
        else                return right ==null? null : right.find(v);
    }

    int compute() {
        int result = value + weight;
        result += (left == null) = 0 : left.compute();
        return result + (right==null)? 0 : right.compute();
    }
}

```

Listing 3 Method copyRoot() execution by Thread T3 and T4 where both of them start by acquiring different trees and then try to execute the method on the tree the other thread has acquired the lock for

```
1  class U extends Thread {
2      U(Tree t1, Tree t2) {tree1 = t1; tree2 = t2}
3      public void run() {tree1.copyRoot(tree2);}
4      Tree tree1, tree2;
5  }
6
7  public static void main(String[] args) throws... {
8      Tree tree1 = new Tree(1), tree2 = new Tree(2);
9      Thread T3 = new U(tree1, tree2);
10     Thread T4 = new U(tree2, tree1);
11     T3.start(); T4.start(); T3.join() T4.join();
12 }
```

annotation "Node left|n = this.n|"that specifies that the atomic set instance n of the object referenced by left is unified with that of the current object.

3.2.2 Deadlock detection and prevention

For any object o created in runtime there will be an atomic set o.t that will protect its fields. Deadlock may arise if two threads concurrently try to acquired resources out of order.

Looking at the example 3, there are two references for tree1 and tree2 passed for both threads but in a different order for each. This will cause the methods to call copyRoot() on each other's tree. Since the tree arguments were passed in a different order each thread will have the lock for one of the trees and try to get the other one, causing a deadlock situation.

How to avoid this deadlock?

It can be prevented by totally ordering all possible locks and making sure that every unit acquires locks in that specific order. To do this, their algorithm tries to find a partial order on atomic sets where $a < b$ means that threads always acquire the a lock before attending to get the b lock. If this order is not found then deadlock is deemed possible. For each path in the call graph from a method m that is a unit of work for an atomic set a to a method n that is a unit of work for atomic set b, they create an ordering constraint $a < b$.

Another deadlock prevention strategy goes through a refactoring of the lock acquisition order. In this strategy we can prevent deadlock using the unitfor annotation that declares a method to be an additional unit of work for the specified atomic set in the argument object. Looking at the previous Example 3, if we annotate copyRoot() with the unitfor annotation then this method will be declared as a unit of work for atomic set instance tree.t, as well as this.t. When a method is a unit of work for multiple atomic set

Listing 4 Tree with concurrent access to weight variable of different nodes

```
class Tree {
    atomicset(t);
    private atomic(t) Node root;
    Tree(int v) {root = new Node(v);}
    ...
}
class Node implements INode{
    atomicset(n);
    private atomic(n) Node left;
    private atomic(n) Node right;
    ...
    void insert(int v){
    ... left = new Node(v) ...
    ... right = new Node(v) ...
    }
}
```

instances then it's guaranteed by AJ that the resources are acquired atomically and thus preventing deadlock.

Right now we are locking the entire Tree in order to avoid a deadlock. Listing 4 presents a Tree annotated to allow concurrent access the weight variable to different nodes in the tree since there's no atomic annotation on the weight field. This means that it's possible to call incWeight() on two nodes simultaneously but if they desire to do a Tree.insert() this would involve acquiring the lock associated with the tree's atomic set t and this ensuring the desired mutual exclusion.

Now, with this implementation, since there is no aliasing annotations, these nodes now have a distinct atomic set instances and the algorithm concludes that deadlock is possible. This occurs since it cannot predict that two threads will not try to access different Nodes in different orders creating a deadlock. However we can verify that this program is deadlock free because once we call insert we transverse the list in a top down order locking then the root and moving trough the tree locking the children of the previous Node.

With the annotations in Listing 5, programmers can now specify an ordering between instances of the same atomic set not allowing the compiler to generate constrains regarding a false positive deadlock when this annotations are present. The type checker ensures that the specified order is acyclic while the analysis verifies that the annotations are consistent with the lock acquisition order.

3.2.3 Annotations

As we analysed previously in the Tree example, without the ordering annotations given by the user, some structures might induce the compiler to a false positive deadlock error or

Listing 5 Node implementation with the annotation |this.a<a|-Annotation on variables and constructors. Specifies the order between atomic set a in the annotated variable or constructed object, and the atomic set a in the current object

```
1  class Node implements INode{
2      atomicset(n);
3      private atomic(n) Node left|this.n<n|;
4      private atomic(n) Node right|this,n<n|;
5      ...
6      void insert(int v){
7      ... left = new Node(v)|this.n<n| ...
8      ... right = new Node(v)|this.n<n| ...
9      }
10 }
```

warning. In cases where we have threads executing units of work associated with multiple instances of the same atomic set the tracking ordering constraints at the atomic-set level are insufficient.

The authors solution for this is having the programmer itself specify ordering annotations that indicate a finer-grained partial order between instances of the same atomic set. Their type system allows one atomic set to be ordered relative to exactly one other atomic set. It also makes sure there's no issues when they are being constructed since they are connected to each other.

This solution is similar to previous works that were studied where the responsibility on the programmer is not simply annotating where the atomic sections are.

3.2.4 Properties

Even though there was no formal proof of these properties, the authors guaranteed the following properties.

Deadlock-freedom. If the analysis has found a valid partial order on all atomic set instances in the program that is consistent with the order in which threads acquire them, then it's guaranteed that program to be deadlock-free.

Serialisability. Whenever threads reach an atomic section their algorithm makes sure that only the thread that gathers all the locks necessary to enter the atomic section executes its code. So, given data-centric synchronisation annotations, the AJ compiler inserts concurrency control operations that are sufficient to guarantee that any execution is atomic-set serialisable [8], i.e., equivalent to one in which, for each atomic set, its units of work occur in some serial order. A unit of work is equivalent to an atomic section that is only atomic with respect to a particular set of memory locations. Accesses to locations not in the set are visible to other threads.

3.2.5 Experimental Evaluation

In the absence of ordering annotations, their analysis guarantees the absence of deadlock in 7 out of the 10 programs tested. Two programs required minor refactoring's before the absence of deadlock could be demonstrated. One of the programs was not able to immediately guarantee the absence of deadlocks. To demonstrate and help the compiler understand the absence of deadlock in that program it was required 4 ordering annotations by the programmer. The running time of the analysis is at most 75 seconds in all cases.

One downside of this evaluation was the small programs where it was tested. Since AJ is not a used language it had to be the developers to transform the Java code into AJ to test them. However, due to the different types of benchmarks used that include a widely-used benchmark and several programs used in research on concurrency error, the developers are optimistic that their implementation will scale well into bigger problems.

3.2.6 Final Remarks

In this work, I made a more in-depth analysis of how several problems of deadlock prevention are tackled in data-centric approach. I started by analysing an example of a syntax of the Java programming language that supports a data-centric programming model called AJ. In the first example the authors proposed a too restricting locking strategy that by making it more fine grained, introduced a deadlock problem. After that I learned how to deal with this type of problems by either using the unitfor annotation or the partial order annotation of AJ. Even thought in most scenarios this algorithm didn't need any help from the programmer, in one of the programs tested, the algorithm was unable to guarantee deadlock and had to ask for help from the programmer.

Even though this approach has a lot of benefits, it is very demanding for the programmer to annotate correctly the partial orders needed in order for the program to work correctly.

3.3 Cameleer

The Cameleer project focuses on creating a tool for checking the correctness of OCaml programs.

3.3.1 What is Cameleer

The Cameleer project [21] introduces a groundbreaking tool for verifying the correctness of OCaml programs. OCaml, known for its compatibility with formal verification, serves as the foundation for this innovation. Cameleer employs GOSPEL [2], a specification language, to precisely define the expected behavior of OCaml programs, enhancing clarity and precision.

This tool streamlines the verification process by accepting OCaml programs directly as input, eliminating the need for extensive code rewriting for verification purposes. Users simply provide their OCaml code along with the GOSPEL specification. Cameleer then takes care of the rest, including the conversion to WhyML, a language tailored for software verification. WhyML generates verification conditions, which are subsequently examined by standard solvers to establish program correctness.

Cameleer's success is evidenced by its ability to verify the correctness of various OCaml programs, including those utilizing queues and heaps from OCaml libraries. This innovative approach simplifies the formal verification of OCaml programs while ensuring precision and reliability throughout the process.

3.3.2 Examples of Cameleer

Let's start with one of the most basic examples. In Listing 6 we can observe OCaml code that consists of two recursive functions, even and odd, which are used to determine whether an integer is even or odd, respectively. The even function returns true if the input integer x is even (meaning it has no remainder when divided by 2) and false otherwise. It employs a recursive approach, subtracting 1 from x with each recursive call until it reaches 0. The odd function, conversely, returns false if the input integer y is 0, indicating that it's even. If y is not 0, it calls the even function with y-1 as an argument, effectively checking if y-1 is even. The Cameleer annotations provide specifications for these functions. For instance, they specify preconditions that require both x and y to be non-negative integers. The annotations also include postconditions, which describe the expected behavior of these functions, helping to clarify their intended use and assisting in formal verification and reasoning about their correctness. Specifically, for even, the postcondition asserts that the result b is equivalent to the expression x mod 2 = 0, indicating an even number. Similarly, for odd, it states that the result b is equivalent to y mod 2 = 1, signifying an odd number. These annotations play a crucial role in documenting and ensuring the correctness of these recursive functions.

The significant step forward occurs when the Cameleer infrastructure translates this OCaml code, along with its annotations, into the WhyML OCaml code presented in Listing 7. This transition is essential because it transforms the code into a form compatible with the Why3 platform.

WhyML is both a programming language and a specification language specifically designed for formal verification. It allows developers to embed their specifications directly into the code structure, creating a seamless integration of specification and implementation. However, the real power of WhyML comes to the fore when combined with the Why3 platform.

Why3, as explained before, is a comprehensive platform for formal verification that provides tools and a graphical proof environment for software verification. By producing WhyML code from the original OCaml code with Cameleer annotations, we facilitate

Listing 6 First Cameleer example

```
let rec even x =
    if x = 0 then true
    else odd (x-1)
(* b = even x
    requires x >= 0
    variant x
    ensures
        b <=> x mod 2 = 0*)
and odd y =
    if y = 0 then false
    else even (y-1)
(* b = odd y
    requires y >= 0
    variant y
    ensures
        b <=> y mod 2 = 1*)
```

Listing 7 WhyML translation OCaml code

```
let rec even x
    requires { x >= 0 }
    variant { x }
    returns { b ->
        b <-> x mod 2 = 0 }
    = if x = 0 then True
    else odd (x-1)
with odd y
    requires { y => 0}
    variant { y }
    returns { b ->
    b <-> y mod 2 = 1}
    = if y = 0 then false
    else even (y-1)
```

the use of Why3 in the formal verification process. Why3's tools, including automated provers and theorem proving capabilities, can then be employed to rigorously verify the correctness of the code against its specifications. This ensures that the software behaves as intended, making it a valuable tool, especially in critical applications and formal methods.

3.3.3 Conclusion

Throughout the course of this thesis, Cameleer played a pivotal role in ensuring the correctness of the implemented functions. Although there were instances when Cameleer faced challenges beyond its initial capabilities, extensive troubleshooting sessions with Professor Mario led to its evolution and maturation.

Now, Cameleer stands poised to verify even the most complex functions, including those involving intricate data structures like Maps and Sets. As we reached the culmination of this research, Cameleer has become an indispensable tool, enabling whoever comes next to successfully conclude the work undertaken during this thesis.

4

The Lock Inference Algorithm

In this chapter we will be overviewing what is the current algorithm used for lock inference in the RC^3 model and what is its strategy in order to prevent deadlock situations.

In the Concepts section 4.1 we take a look at simple but very important concepts that are necessary to comprehend and are regularly used in the complex parts of the algorithm.

The Algorithm section 4.2 details in-depth what is the strategy and the goal for each step of the algorithm.

4.1 Concepts

It is important to detail some basic notions about the algorithm in order to further explain how the algorithm works.

Labels

The algorithm assumes that the position of every instruction needs to be uniquely identified. In order to do this, it was introduced the concept of label.

A label, denoted l in *Label*, is a unique identifier that identifies the position of a language construct in the source program. For instance, if the value of the variable x is read in the fifth line of a method, a label will be created to store information about this read operation upon the x variable. It is necessary to have this information since the algorithm needs to know what data is being accessed in what order. To have a order relation between labels, it's defined a successor function that states which label comes after the other and a *Label* set where labels are totally ordered according to their position in the method scope.

Resources

A resource directly represents a memory object, this is, a value in the heap memory.

To represent a memory object, the algorithm stores information about several aspects related to it. For example if the resource is atomic, what is its type, if it is public or private and many others that will be detailed in the Implementation section 5.2.

It is important to note that whenever there is a method call different arguments may result in different accesses to resources. It's in this phase of the algorithm that is processed which resources can be accessed in the caller method.

Resource Access

A resource access is a concept that makes it possible, given a resource, to identify where, during a method execution, are operations relative to the resource executed and which ones. The resource access retains then information if the resource is being accessed for a read or a write operation and in which specific labels of the source program is this happening.

A resource access, denoted $ra \in ResourceAccess = (Label \times Label \times Label \times Label \times Label)$ is represented by a five-element tuple $(l_a, l_{fr}, l_{lr}, l_{fw}, l_{lr})$ where l_a represents the first operation that does the first read of the resource, l_{fr} represents the first time the resource is read, l_{fl} represents the last time the resource is read, l_{fw} represents the first time the resource is written, l_{lw} represents the last time the resource is written.

Now, by using this five-element tuple, information on read and write operations performed on a resource during a method execution can be stored.

Resource Access Map

In a program, resources may be accessed in multiple methods, with each of such accesses being denoted by a resource access. To facilitate this mapping, an important data structure known as the resource access map is used to associate, for each method, its respective resource access relationship.

The resource access map tracks which resources were accessed and which operations were done within each method, making it a crucial tool for a lock inference algorithm. It will be essential to have this map, since multiple data-structures and important phases of the algorithm will rely in the information captured by it.

Resource Operation and Resource Group

A resource operation is a code that details which operation is a resource access doing in a specific part of the method.

There are seven different types of operations that can be executed. This concept takes the resource being accessed and the specific operation and creates a pair (op, r) where r is the resource of the memory object and $op \in LockOperation = \{A, S, E, B, D, U, R\}$ S denotes **Shared** (read) lock operation, E denotes **Exclusive** (write) lock operation, B

denotes a Shared **Before** Exclusive lock operation, D denotes **Downgrade** lock operation, U denotes **Upgrade** lock operation, R denotes **Release** lock operation.

A resource group, as the name suggests, stores a list of pairs (op, r) with the goal of creating a total lock order between the resource accesses that were deemed to belong to the same resource group.

4.2 The Algorithm

The RC^3 model has an important pipeline that is worth showcasing. It starts by taking Jave byte code, parsing it and transforming it into an Abstract Syntax Tree (AST) where the points-to analysis occurs. This analysis extracts Method Resource Information (MRI) that is needed in order to perform the lock inference algorithm to determine the necessary locks for each resource. After this analysis is complete and the lock inference algorithm has terminated the AST is generated into Java byte code with calls to the RC^3 lock runtime.

MRI contains information about which memory locations are accessed during this method execution and which methods are called inside this method's body. It is important to have in consideration what are the memory locations accessed by those methods as well.

The lock inference algorithm is showcased in Algorithm 1. The first part of the algorithm is performed for each class and method in the program. In this phase (line 4-12), several steps will take place in order to ensure more efficiency and simplicity in the graph that will be created for deadlock detection and prevention. The second phase starts when the graph is created. There is a step where each class's graph is merged into a general program graph and then changes are executed in order to prevent deadlock situations (line 15-24).

Method Resource Info and Resource Map

Before the start of the algorithm, it is done a points-to-analysis to the program that is being analyzed where information about what resource accesses are being executed and a Map that stores such information is also created.

Algorithm 1 starts exactly by extracting not only the method resource info that contains information about the resources accessed and the method calls inside the method (line 4), but also the resource Map that was created in the analysis (line 5). In the map, it is detailed where the resources are first and last accessed by a read or write operation.

This mapping is crucial for the lock inference algorithm, since the next phases will need the resource access map in order to complete their tasks.

The resource access map isn't the only one needed to perform the algorithm. It is also needed to store information about which methods access which resources in order for the algorithm to be able to subsume some of the resources.

Algorithm 1 RC^3 lock inference and deadlock prevention strategy algorithm

```
 1: g ← graph
 2: for c ∈ Classes do
 3:     for m ∈ methods(c) do
 4:         MRI ← getMethodResourceInfo(m)
 5:         RA ← getResourceMap(MRI)
 6:         Cs ← getMethodCalls(MRI)
 7:         RA ← RA ∪ resolveMethodCalls(Cs)
 8:         RA ← resourceSubsumption(RA)
 9:         RO ← RA2RO(RA)
10:         RO ← parameterGrouping(RO)
11:         RO ← 2pl(RO)
12:         addOperationsToGraph(RO, g)
13:     end for
14: end for
15: while (sccs ← getSCCs(g)) ≠ ∅ do
16:     for scc ∈ sccs do
17:         (b, scc) ← canBreakCycle()
18:         if b then
19:             updateGraph(g, scc)
20:         else
21:             colapseSCC(g, scc)
22:         end if
23:     end for
24: end while
```

Resolve Method Calls

During this phase, an analysis is conducted to identify the resources that are accessed inside the body of methods called. The algorithm will take all the resource accesses inside the the method calls and display them as if they were called inline in the calling method execution.

This procedure done in the Resource Method Calls (line 6) may give the impression that there are no method calls taking place but instead, all the resource accesses are done within that specific method.

Resource Subsumption

The resource subsumption phase is a simplification phase where, given the resource access maps for each method created previously, after an analysis, the maps are simplified if there is an opportunity to do so.

This simplification is based on the dominance property. There are some resources that dominate others (as explained in the *Lock Inference* paragraph of Section 3.1.1), where it is safe to stop having these resource accesses protected by a different locks and combine them under the same lock, reducing the amount of total locks in the algorithm. This

Listing 8 Subsumption example

```java

public class SubsumptionExample {
    private int resource1;
    private int resource2;

    // Method 1 dominates resource2 with resource1
    public void method1() {
        resource1++; // FirstWrite and FirstRead of resource1
        resource2++; // FirstWrite and FirstRead of resource2
        resource2--; // LastWrite and LastRead of resource2
        resource1--; // LastWrite LastRead of resource1
    }

    // Method 2 also dominates resource2 with resource1
    public int method2() {
        int value11 = resource1; // FirstRead of resource1
        int value21 = resource2; // FirstRead of resource2
        int value22 = resource2; // LastRead of resource2
        int value12 = resource1; // LastRead of resource1
        return (value11 * value21) - (value22 * value12);
    }
}
```

operation does not hinder performance but rather improves it, since constantly locking and unlocking can have a bad impact in the efficiency of the program.

In order to do this, the resource access map is required for computing which resources dominate the others. The way this is done is by analyzing, for each resource in the program, if for every method it is accessed in, there is another resource that dominates it every time. In case there are resources that dominate others in every method, it is safe to make it so that these resources are be protected by the same lock and it isn't needed in the next phases of the algorithm, to reason about the dominated resource.

In Listing 8 we can observe an example of how a resource will dominate the other. In this example there are two resources and two methods executing operations on both. In `method1`, both writing operations of `resource2` are contained between `resource1`'s operations. It also happens in `method2` but for reading operations. Given `method1`'s operations, the resource access information for resources `resource1` and `resource2` are, respectively, the following:

$$resourceAccess1 = \{first = FIELD, firstRead = 1, LastRead = 4, firstWrite = 1, LastWrite = 4\}$$

$$resourceAccess2 = \{first = FIELD, firstRead = 2, LastRead = 3, firstWrite = 2, LastWrite = 3\}$$

Assuming these are the only methods `resource2` is present on, it is easy to conclude that resource1 dominates `resource2` since every `resource2` label is contained within resource1 labels.

Resource Operations Map

The next step is to compute the resource operations map *RO*. This is a map that projects labels into a set of resource groups. Reminding that each label is a unique identifier that identifies the position of a language construct in the source program, this map is associating each of such positions to a set of resource groups.

Not all lock libraries allow Upgrade operations. This Upgrade operations can be related to java Read/Write locks. So, whenever that is the case, there is an intermediate step where all the Shared Before Exclusive type of locks are transformed into Exclusive (write) lock operations and the Upgrade operations disappear.

The RO map co-relates each line of the program to a list of resource groups that contain which operations are done in that specific line and to which resources. With the resource operations map, it is possible to have a different view of the program from the resource access map, since it has information related to the resource operations and not only where are the read and write operations to each resource.

Parameter Grouping

If there are parameters that are of the same type or have a sub-type relationship between each other, then they are placed inside the same resource group. By grouping resources of the same type, instead of locking each resource individually and in some circumstances, causing a potential deadlock situation, they are now locked in a total defined order of execution making it not possible for a deadlock situation to arise.

This is achieved by an analysis of the resource operations map generated previously and making the necessary changes in order to group the resources of the same type. For this to happen, there will have to be a selection of two resources at a time to test if, in this pair, there is indeed a type conflict. It is tested if both resources are parameters, if their operations conflict with each other, that is, if both of them are executing a Release or Upgrade operation and lastly, if both of these resource types are the some or sub-types of each other. If the conflict is verified, then one of the resources is placed inside the other resource group.

Given the bank account example 1, in the transfer method, both accounts are placed in the same resource group due to them being of the same Account type. With this grouping, the locks for both accounts will be acquired together.

Two Phase Locking (2pl)

Function `2pl` executes the two-phase locking protocol explained in detailed in Listing 2.1.3 where there are two separate phases, one for resource lock acquisition and one for resource lock release.

This is achieved by pushing the Release and Downgrade operations to after the Shared, Exclusive and Upgrade operations are all executed. In this way, it never starts unlocking

Listing 9 Two Phase Locking example

```
public class SubsumptionExample {
   private int resource1;
   private int resource2;
   private final Lock lock1 = new  ReentrantLock();
   private final Lock lock2 = new  ReentrantLock();

   //Method that does not respect 2pl protocol
   public void method1() {

            lock1.lock(); // Acquire lock1 exclusively
            resource1++; // Access resource1 exclusively

            lock1.unlock(); // Release lock1

            lock2.lock(); // Acquire lock2 exclusively
            resource2++; // Access resource2 exclusively

            lock2.unlock(); // Release lock2

   }

   //Method that respects 2pl protocol
    public void method1() {

            lock1.lock(); // Acquire lock1 exclusively
            lock2.lock(); // Acquire lock2 exclusively

            resource1++; // Access resource1 exclusively
            resource2++; // Access resource2 exclusively

            lock2.unlock(); // Release lock2
            lock1.unlock(); // Release lock1
        }
}
```

resources until all the locks have been acquired ensuring serializability and not compromising atomicity.

The resource operations map is essential in order to perform this phase of the algorithm. Looking at the labels and the operations that are associated with them it is possible to understand if the two-phase locking protocol is being respected, and if it's not, a change on how the operations are being executed is needed to ensure the protocol is respected.

In Listing 9 ,we have an example of what what this phase of the algorithm does. Let's say we have two resources, `resource1` and `resource2`, that in `method1` execute operations. We can observe that in `method1` first iteration (line 10), `lock1` does its Release operation (line 15) before resource two acquires its lock (line 17). According to two phase locking

protocol, there can't be any Release operations until all locks are acquired. Due to this, this phase moves locking operations to a correct order that respects the protocol. In the second iteration of method1 (line 25), the locking strategy now abides to the protocol. Both lock acquires (line 27-28) come before both releases (line 33-34).

Add operations to graph

In this phase, the algorithm adds to the graph the information until now inferred. For each resource it is added two nodes to denote shared and exclusive locking.

This graph is simultaneously a directed and an undirected graph since it has edges that detail the lock order between resource groups but there is also edges with no direction that just establish a connection between the resources. Each node inside a group is connected with the other resources accessed with an edge with weight 0 while the edge connecting different groups has a weight of 1.

The order in which different groups are connected is defined in the construction of the graph while the order inside a group can be random at first but then it's defined in execution time. For the sake of creating this order the resource operation map is iterated from the lowest to the highest label creating edges between the different groups that establish the total locking order.

Strong connect components

The last part of the algorithm is what allows it to prevent deadlocks. Now that it has the graph created with all the resource groups ordered between themselves, according to the features extracted from the source code and annotations, it is necessary to understand if the order created has a potential deadlock situation or not.

In order to do this it's necessary to identify where are the Strong connected components, or a collection of group resources that form a circular path connecting them. If by any chance multiple threads obtain concurrently locks in that cycle dependence then a deadlock situation might happen.

There are two ways of dealing with Strong Connect Components. The first one is to break the cycle by inverting or changing some edges direction that break the cycle. This is the best case scenario. If this is absolutely not possible, then this strong connect component has to be collapsed into a single lock. Only one thread at a time is going to have access to all this resources which hinders the performance.

5

Implementation and Verification

In this section it is detailed how was the OCaml implementation of the algorithm executed and how was the Cameleer verification for Why3 done in some of the functions.

We take a look at each part of the algorithm once again with more detail and analyzing it step by step with code examples.

The code for the algorithm can be viewed in this repository:

https://github.com/PedroSalgado28/lockinferenceFinal

5.1 Repository

In order to be easier to navigate through the source code, I will give a brief explanation of the important folders and files.

The lib folder (Link to repository Lib folder) is where the code of the OCaml version of the RC^3 program is. In here we can find all the modules and their implementations, with the corresponding Cameleer annotations for their verification.

In the OUnit folder (Link to repository OUnit folder), we can find all the test suits for the modules that were not fully verified. This tests use the OUnit2 OCaml library.

In the sections to come, I will make references for this repository and for some particular functions in it.

5.2 Concepts

Even though these concepts were already explained in the previous section 4.1, it is important to understand how they were implemented in OCaml.

We will go through the implementation and type choice for several concepts such as the Label , the Resource and Resource Access and lastly the Resource Operation and Resource Group implementations.

Listing 10 Resource Type module. Repository Link: resource.ml

```
module ResourceClass =
struct

type resource = {
  position: int;
  ty: int;
  vis: int;
  nature: bool;
  whatIs: int;
  guardedBy : int;
  isParameter : bool
}
```

Label

A label is a unique identifier that identifies a position of a language construct in the source program. All the labels in the implementation will be represented by integers.

For example, in the Resource Access implementation in Listing 11 that consists of a five label tuple, it is represented as a five integer tuple.

Resource

A resource directly represents a memory object. There are different parameters that are important to have knowledge about in a memory object. For this, a `ResourceClass` module was implemented with a `resource` OCaml type, as showed in Listing 10, with the purpose of representing the resources. The type has the following variables:

- Position: An integer that is the unique identifier of the resource in the program.
- Type: An integer that represents the type of the resource in the source code.
- Visibility: An integer that represents if the resource is public, private or protected.
- Nature: A boolean that represents if a resource is or not atomic.
- WhatIs: An integer that represents if the resource is the result of a method call, a field, an object or the result of a cast.
- GuardedBy: An integer that represents the Id of a resource that is responsible for guarding the resource, this means, what lock is responsible for an atomic access to the resource. It has it's own Id if it's not guarded by another resource, meaning, the resource has its own lock protecting it.
- isParameter: A boolean that represents if the resource is a parameter or not.

The resource module comprises simple functions that have the purpose of making changes in the properties of a resource's state the can be changed during the algorithm.

Listing 11 ResourceAccess OCaml Type. Repository Link: resource.ml

```
module Access =
struct

type resourceAccess = {
                mutable first : int;
                mutable firstRead : int;
                mutable lastRead :int;
                mutable firstWrite : int;
                mutable lastWrite :int
}

```

ResourceAccess

A resource access consists of five labels that together have information about which operations does a certain resource do in that specific method of the program.

For this, a module was created together with a resourceAccess OCaml type that can be observed in the Listing 11. This type consists of the following variables:

- First: Also called accessiblefrom, this label, represented as an integer, contains information from where the resource is first referenced in a method and has a -1 value if the resource is a parameter and -2 if it is a field.
- FirstRead: A label, represented as an integer, that represents the line of the method code where the resource does its first read operation. If there is no read operation this variable has value -1 .
- LastRead: A label, represented as an integer, that represents the line of the method code where the resource does its last read operation. If there is no read operation this variable has value -1.
- FirstWrite: A label, represented as an integer, that represents the line of the method code where the resource does its first write operation. If there is no write operation this variable has value -1.
- LastWrite: A label, represented as an integer, that represents the line of the method code where the resource does its last write operation. If there is no write operation this variable has value -1.

Resource Operation

A resource operation is a concept that specifies what type of operation is being done in a specific line of the source code. There are 7 different types of operations: Available, Shared, Shared Before Exclusive, Exclusive, Upgrade, Downgrade and Release. To capture

Listing 12 Resource Operation OCaml Type. Repository link: roperation.ml

```
type rOp = { op : int;
             r  : int;
}
```

Listing 13 Resource Group OCaml Type. Repository Link: resourceGroup.ml

```
type resourceGroup = {
  id : int;
  ropList : rOp list
}
```

all these operations it was created a module with a resource operation type that we can observe in Listing 12.

The resource operation type attributes one of the seven resource operations *op* to an access that is done, during the method execution, to a single resource. This resource *r* is identified by its resourceID.

Resource Group

As its name implies, a resource group was implemented as an OCaml type that aggregates several resources and resource operations under one group.

For the implementation of the resource group type in Listing 13, it was necessary the creation of two variables:

- Id: The id variable represents the resource group unique identifier as an integer.
- RopList: A list of resource operations.

5.3 Initialization

In this section we present the first steps of the algorithm that include what input is received, what data-structures are initialized and the implementation required for that initialization.

Parser and input

The algorithm starts by receiving input from the RC^3 model. This input consists of data related to the resources accessed analyzed in the program. It details in which classes and methods were they accessed and what are their labels values.

We developed the displayed BNF grammar in the Listing 14 for the parser to process the input.

It is in this grammar that several restrictions in the input are created. For example, the accessible-from label that we explained in 5.2 can only contain values between -2 and a natural (line 31).

It is by using several restrictions in this parser that there is no need to check if the inputs are being received correctly or not. We assume for example, that the `labels` construct always consist of a five element tuple since in line 30 it is detailed that that is always the case.

Main

The information extraction of the file provided by the RC^3 model initial analysis is divided in two phases. A resource oriented phase where the file starts by giving information about which resources are present in the program and what are its variables values. The second phase focuses on what classes and methods access which resources and what are the labels for each resource access.

In the first phase it's given unique identifiers to each one of the resources. Then, the file proceeds to give the values of all the resource's OCaml type variables mentioned above. That is, its type, visibility, nature and so on. The `guardedBy` variable is initialized with the resource's own unique identifier and the `isParameter` variable is initialized by default with a false value.

To store this information it was used a Map functor of the OCaml language. The keys of the map are the unique resource identifiers while the values are the resource type of that identifier. With this, it is possible to get the resource type from its unique ID.

In the second phase, it is received information about the class and the method name. Subsequently, it is received data relative to which resources are present in that class and method pair and which are its labels. With these means, it is possible to build a skeleton of the program.

It was used again the Map functor of OCaml to store this data. In order to do it more efficiently and in an simplified way, the class and the method were merged in the same string and used as the key of the map. This map was called `classMethodInfoMap`. Since each class and method have multiple resources associated to them it is necessary that the value of the map is another Map functor. This second map contains as its keys the unique identifiers of the resources and, as the value, a resource access that contains the five labels associated with that resource. This second map is regarded as the `resourceAccessMap`.

The second phase is finished when the file ends.

Mapping

The mapping module serves as a crucial component for organizing incoming data into the appropriate data structures. This module operates in two distinct phases, with the

Listing 14 BNF grammar used for the parser

```

  <line-end> ::= <opt-whitespace> <EOL> | <line-end> <line-end>
<opt-whitespace> ::= " " <opt-whitespace> | ""
<text1> ::= "" | <character1> <text1>
<character1> ::= <character> | "'"
<character> ::= <letter> | <digit> | <symbol>
<letter> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<digit-nonzero> ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<negative> ::= "-"
<symbol> ::=  "|" | " " | "!" | "#" | "$" | "%" | "&" | "(" | ")" | "*" | "+" | "," | "-" | "." | "/" | ":" | ";" | ">" | "=" | "<" | "?" | "@" | "[" | "\" | "]" | "^" | "_" | "`" | "{" | "}" | "~"
<nat> :== <digit> | <digit> <nat>
<nat-nonzero> :== <digit-nonzero>
<integer> :== <nat> | <negative> <nat-nonzero> | <negative> <nat-nonzero> <nat>

<program> :== <resources> <classMethods>

<resources> :== "resources" <line-end> <resource> | "resources" <line-end> <resource> <resources>
<resource> :== <resourceID> <line-end> <type> <line-end> <visibility> <line-end> <nature> <line-end> <resourceType> <line-end>
<resourceID> :== <integer>
<type> :== <nat>
<visibility> :== <nat>
<nature> :== <nat>
<resourceType> :== "0" | "1" | "2" | "3"

<classMethods> :== "class" <line-end> <className> <line-end> <method> | "class" <line-end> <className> <line-end> <method> <classMethods>
<className> :== <text1>
<method> :== "method" <line-end> <methodName> <line-end> <resourceAccess> | "method" <line-end> <methodName> <line-end> <resourceAccess> <method>
<methodName> :== <text1>
<resourceAccess> :== "resource" <line-end> <resourceID> <line-end> <labels> <line-end> | "resource" <line-end> <resourceID> <line-end> <labels> <line-end> <resources>
<labels> :== <accessible-from> <label> <label> <label> <label>
<accessible-from> :== "-2" | "-1" | <nat>
<label> :== "-1" | <nat>

```

Listing 15 ComputeResource function of the Mapping Module. Resource Link: mapping.ml

```
1
2
3  let computeResource resId typ visib nature whatIs resourceMap resourceMethodsMap=
4    let (r:ResourceClass.resource) =
5    {ResourceClass.position = resId; ResourceClass.ty = typ; ResourceClass.vis = visib;
6    ResourceClass.nature = nature; ResourceClass.whatIs =  whatIs;
7    ResourceClass.guardedBy = resId; ResourceClass.isParameter = false} in
8    let newResourceMap = IntMap.add resId r resourceMap in
9    let newResourceMethodsMap = IntMap.add resId [] resourceMethodsMap in
10   newResourceMap, newResourceMethodsMap
11   (*@newResourceMap, newResourceMethodsMap =
12   computeResource resId typ visib nature whatIs resourceMap resourceMethodsMap
13     requires resId >= 0
14     ensures  let (r:ResourceClass.resource) =
15     {ResourceClass.position = resId; ResourceClass.ty = typ; ResourceClass.vis = visib;
16     ResourceClass.nature = nature; ResourceClass.whatIs =  whatIs;
17     ResourceClass.guardedBy = resId; ResourceClass.isParameter = false} in
18     (IntMap.mem resId newResourceMap && newResourceMap.IntMap.view resId = r) &&
19     (IntMap.mem resId newResourceMethodsMap && newResourceMethodsMap.IntMap.view resId
       ↪  = [] ) *)
20
21
```

first phase primarily dealing with resource insertion, as demonstrated in Listing 15.

The mapping module is an important module for the arrangement of the input received into the correct data-structures. In this module the input received is analyzed and several OCaml functions are going to make insertions in the maps accordingly.

During the initial phase, the module receives resource variables and creates a corresponding resource type. This newly created resource type is then placed into both the resource map and the resource methods map, which are the two return values of the `computeResource` function.

In Listing 15, the `computeResource` function (line 6) retrieves the resource variable values such as type, visibility, and nature, and then generates a new entry in the ResourceMap. This entry consists of a key-value pair where the key is the unique identifier of the resource, and the value is an OCaml type called Resource, defined with the received parameters. Additionally, it's essential to initialize the map that keeps track of which methods access this resource.

This map, referred to as the resource methods map, is introduced in line 7, where an entry is added with the resource identifier as the key and an empty list as the value. This list remains empty during this phase because it is still uncertain which methods are associated with this resource. Subsequent phases will populate this list.

The second phase of this module is responsible for processing inputs related to the

Listing 16 computeResourceMethodsMap function of the Mapping Module. Resource Link: mapping.ml

```
1
2  let computeResourceMethodsMap res classAndMethod resourceMethodsMap=
3    try let methodArray = IntMap.find res resourceMethodsMap in
4     let newMethodArray = append_item methodArray classAndMethod in
5     let newResourceMethodsMap = IntMap.add res newMethodArray resourceMethodsMap in
6     newResourceMethodsMap
7     with Not_found -> (
8       resourceMethodsMap
9     )
10      (*@ newResourceMethodsMap = computeResourceMethodsMap res classAndMethod
      ↪  resourceMethodsMap
11      ensures if IntMap.mem res resourceMethodsMap then
12          exists methodArray.
13          methodArray = newResourceMethodsMap.IntMap.view res && List.mem classAndMethod
          ↪  methodArray
14        else
15          IntMap.mem res newResourceMethodsMap = false
16  *)
17
```

structure of the source program and constructing new maps or enhancing existing ones with this information.

This phase begins with a function that takes as arguments the class and method, the unique identifier of the resource, the list of resource labels, and all three data structures created. These data structures are:

- `classMethodInfoMap`: A map that stores the program's skeleton.
- `resourceMap`: A map that maps resource IDs to their actual resource types.
- `resourceMethodsMap`: A map with resource identifiers as keys and lists of classes and methods they belong to as values. This map is crucial for subsequent parts of the algorithm.

The primary function in this module distinguishes whether the classMethodInfoMap already contains the class and method pair. If it is the first resource access for a specific method, the map needs initialization with all necessary keys and values.

In the `classNotFound` function (line 2), a simple addition is made to the `resourceAccessMap` using the received values, followed by adding it to the `classMethodInfoMap` with the class and method strings serving as its key.

Line 4 demonstrates the invocation of the `computeResourceMethodsMap` function (Listing 16) to compute the resource methods map.

Recalling that in the earlier phase, the resource methods map consisted of key-value pairs where the key was a resource identifier, and the value was an empty list since the associated methods were unknown at that time. In the current phase, whenever a resource is received, we access the corresponding key in the map and append the class and method to the list value (lines 3-4).

In summary, this module receives inputs from the main program and, in two distinct phases, stores resources in a resource map, and records all classes, methods, and resource accesses in their respective maps.

5.4 Algorithm

In this section it is done a deep dive into the algorithm and the verification of important functions.

Pipeline

Before starting the algorithm it is important to understand the pipeline that will follow suit. The OCaml implementation will follow the same pipeline used in the Java implementation.

It starts by doing resource subsumptions whenever it finds any resource that is being dominated by another and making the necessary changes in the resource access map of each method.

The updated resource access map serves as a base for the creation of the resource operations map. The newly created resource operations map is now used to find resources accessed of the same type and grouping them up, correct operations that are not respecting the two phase protocol and updating the resource map that has information about which locks are protecting each resource.

Resource Subsumption

In this section, we'll delve into the implementation of the dominance property within the context of resource labels captured in the resource access type. The dominance property stipulates that for one resource (A) to dominate another resource (B), we must consider the operations performed on resource B during method execution. Specifically, we examine whether resource B is only read, only written, or both.

For resource B to be dominated during a read operation, the following conditions must hold:

Both the firstRead and lastRead labels of resource B (and both write labels if there is also a write access) must be within the labels of resource A. This implies that firstRead and firstWrite of resource A must be greater than or equal to the corresponding values of resource B, and lastRead and lastWrite of resource A must be less than or equal to the corresponding values of resource B.

Listing 17 dominatesCheck function of the Resource Subsumption Module. Resource Link: rSubsumption.ml

```
1   let dominatesCheck (dominantAccess:Access.resourceAccess)
    ↪ (r2Access:Access.resourceAccess)=
2    let res = ref true in
3    if r2Access.Access.firstRead <> -1 then
4      begin
5        if (dominantAccess.Access.firstRead > r2Access.Access.firstRead
6          && (dominantAccess.Access.firstRead > r2Access.Access.firstWrite ||
           ↪ r2Access.Access.firstWrite = -1))
7          || (dominantAccess.Access.lastRead < r2Access.Access.lastRead
8          && (dominantAccess.Access.lastRead < r2Access.Access.lastWrite ||
           ↪ r2Access.Access.lastWrite = -1))
9           then
10           res := false
11         else ()
12       end
13     else ();
14
15     if  r2Access.Access.firstWrite <> -1 then
16       begin
17       if dominantAccess.Access.firstWrite > r2Access.Access.firstWrite
18         || dominantAccess.Access.lastWrite < r2Access.Access.lastWrite then
19         res := false
20       else ()
21     end
22   else ();
23   !res
24   (*@res = dominatesCheck dA rA
25      ensures res = false <-> (rA.Access.firstRead <> -1  && (dA.Access.firstRead >
      ↪ rA.Access.firstRead
26       && (dA.Access.firstRead > rA.Access.firstWrite || rA.Access.firstWrite = -1 )
27       || dA.Access.lastRead < rA.Access.lastRead
28       && (dA.Access.lastRead < rA.Access.lastWrite || rA.Access.lastWrite = -1))
29       || (rA.Access.firstWrite <> -1 && ( dA.Access.firstWrite > rA.Access.firstWrite
30       || dA.Access.lastWrite < rA.Access.lastWrite)))
31        *)
32
```

Here is the dominatesCheck function (see Listing 17) that implements this logic.

The verification process (lines 26-32) ensures that the result of the function is false if and only if the conditions described above are met.

Now, let's go back to the beginning of the module to understand how this label comparison is carried out. This phase of the algorithm iterates through all the classes and methods in the program, comparing each resource within them to others present in the same method. The goal is to determine whether the first selected resource dominates the second one.

Listing 18 logic function of the RSubsumption Module. Resource Link: rSubsumption.ml

```
let logic resource1 resource2 res classMethodInfoMap resourceMethodsMap resourceMap=
    try let r1methods = IntMap.find resource1.ResourceClass.position resourceMethodsMap
    ↪  in
        let r2methods = IntMap.find resource2.ResourceClass.position resourceMethodsMap
        ↪  in
        if sub r1methods r2methods then
           res := methodCycle r2methods true resource1 resource2 classMethodInfoMap
        else ();
    resourceMapUpdate !res resource1 resource2 resourceMap
  with Not_found ->
    resourceMapUpdate !res resource1 resource2 resourceMap

```

The logic function (see Listing 18) is responsible for this comparison. It begins by retrieving the lists of methods to which both resource1 and resource2 belong from the resourceMethodsMap. It then checks if all methods of resource2 are also in resource1, indicating that resource1 is present in every method of resource2.

If this condition holds, the algorithm proceeds to verify if the dominance property is true for every method of resource2. This involves iterating through the map containing information about each resource access in the class and method and comparing their labels using the dominatesCheck function. If any instance returns false, it is concluded that there is no dominance, and the algorithm moves on to the next resource access. If all instances in every method return true, it indicates that one resource dominates the other.

To establish dominance, the guardedBy variable within the resource class must be modified, and the resource map needs updating. The resourceMapUpdate function is responsible for this. It sets the guardedBy attribute of resource2 to resource1, signifying that resource1 dominates resource2. Additionally, it updates the resource map by replacing the old resource2 with the modified resSubsumed.

In summary, this subsection has provided insights into the implementation of the dominance property and its verification process. It has explained how resource labels are compared, how resources are checked within methods and across the entire program, and how resource maps are updated to establish resource subsumption.

Resource Operations Map

In this phase, the primary objective is to create the resource operations map, which plays a pivotal role in storing labels and resource groups. Specifically, for each label representing a line in the method's source code, there exists a list of associated resource groups. These resource groups, in turn, store information about resource operations. The resource operations map is of utmost importance for the subsequent steps of the algorithm.

As previously explained, a resource group is a collection of pairs (op, r) where op

Listing 19 getPair function of the ResourceOperations Module. Resource Link: Resource-OperationsMapGeneration

```
let rec getPair accessMap bindings rOperationsMap counter=
 match bindings with
  | [] -> rOperationsMap
  | x :: xs -> (
    try let access: Access.resourceAccess = IntMap.find x accessMap in
    let rOperationsMap, counter = computeROperations x access rOperationsMap counter in
    getPair accessMap xs rOperationsMap counter
    with Not_found -> (
      getPair accessMap xs rOperationsMap counter
    ))
```

denotes the operation code being executed, and r represents the resource associated with that operation.

To create this map, the algorithm requires the resource access map as input and iterates through it to build the resource operations map. The process begins by analyzing each resource from the resource access map individually.

In the getPair function (see Listing 19), this iteration is accomplished using pattern matching (match...with) on the bindings (integer keys of the resource access map corresponding to resource IDs). For each key, the algorithm computes the resource operations for that resource and updates the resource operations map accordingly (line 6). Once all bindings have been processed, the new resource operation map is returned.

Subsequently, the resource operation for each access to a resource is determined based on label comparisons. A series of conditions are executed to establish which resource operation corresponds to each resource access.

For instance, the `labelsVerification` function (see Listing 20) exemplifies a function responsible for part of the label comparison conditions. It showcases two conditions, one in line 2 and another in line 7. In the first condition, it checks if the resource performs read operations and whether the `lastRead` label occurs before the `lastWrite` label in the source program. If this condition holds, the `insertRO` function is called to insert a Release operation.

The resource operations map is built by executing a chain of functions that carry out multiple comparisons.

The createNewRGroup function (see Listing 21) is responsible for inserting resource operations into the resource operations map. It takes a label (the key of the map), a resource operations map, a counter (for generating resource group IDs), the resource operation to be inserted, and a list of resource groups (the values of the map) as input.

The post-condition of this function clarifies its behavior. It ensures that a new list (newList) exists in the returned map with the input label as its key (line 10). This new list should be longer than the previous list in the resource operations map since an insertion occurs in line 5. Additionally, the post-condition states that a new resource group must

Listing 20 labelsVerification function of the ResourceOperations Module. Resource Link: ResourceOperationsMapGeneration

```
let labelsVerification firstRead lastRead lastWrite res rOperationsMap counter=
      if firstRead == -1 || lastRead < lastWrite  then
        let newROperationsMap = insertRO 7 lastWrite res rOperationsMap counter in
        let newCounter = counter +1 in
        newROperationsMap, newCounter
      else
        if lastWrite < lastRead then
          let newROperationsMap, newCounter =
          firstReadORlastWrite firstRead lastWrite res rOperationsMap counter in
          let newROperationsMap =insertRO 7 lastRead res newROperationsMap newCounter in
          let newCounter = newCounter +1 in
          newROperationsMap, newCounter
        else rOperationsMap, counter
```

Listing 21 insert function of the ResourceOperations Module. Resource Link: ResourceOperationsMapGeneration

```
let createNewRGroup label rOperationsMap counter rop list=
  let rGroup = ResourceGroup.rGroupVar() in
  let newRGroup = { id = counter; ropList = rGroup.ropList} in
  let newRGroup = ResourceGroup.add rop newRGroup in
  let newlist = list @ [newRGroup] in
  let newROperationsMap = IntMap.add label newlist rOperationsMap in
  newROperationsMap
(*@newROperationsMap = createNewRGroup label rOperationsMap counter rop list
  ensures exists newList.
    (newList = newROperationsMap.IntMap.view label && IntMap.mem label newROperationsMap) ->
    List.length list < List.length newList &&
    exists newRGroup.
    let rGroup = ResourceGroup.rGroupVar() in
    newRGroup = { id = counter; ropList = rGroup.ropList} &&
    List.mem newRGroup newList
*)
```

be created, containing the received resource operation (added to its resource operation list in line 4). Finally, this new resource group should be present in the newly created list (newList) and inserted into the returned map (line 15).

By iterating through the resource access map, comparing labels, determining the resource operations to be inserted, creating resource groups, and populating the resource operations map, the algorithm successfully generates the required map for use in subsequent steps.

The presented text retains references to specific code segments and maintains clarity in explaining the process of resource operations map generation in the Resource Operations module.

Parameters Grouping

In the Parameter Grouping phase of the algorithm, the primary objective is to optimize resource grouping within a method. This optimization process aims to identify and resolve resource conflicts that arise when two resources of the same type are accessed concurrently. By resolving these conflicts and adjusting resource groups accordingly, the algorithm can enhance the overall efficiency and safety of concurrent programs.

The phase begins by utilizing the resource operations map generated in the previous section. This map contains vital information about the resource operations and their corresponding resource groups. Each resource operation is associated with a specific line of code within the method. The goal is to analyze these resource operations, identify conflicts, and then take appropriate actions to resolve them

Identifying Parameters

First, the algorithm must distinguish which resources within the method are parameters. Parameters are resources that are passed into the method and can potentially conflict with other resources accessed within the same method. To achieve this, the algorithm leverages a boolean flag known as isParameter associated with each resource. This flag indicates whether a resource is a parameter.

The algorithm begins by creating a set called "Parameters"and then iterates through the resource operations map. During this iteration, it checks each resource operation's associated resource to determine whether it is flagged as a parameter. If a resource is identified as a parameter, it is added to the "Parameters"set. This set serves as a filter, ensuring that only parameter resources are considered for conflict resolution.

The addParameters function in Listing 22, depicted in the code, captures this process. It iterates through the resource operations list, checks the isParameter flag for each resource, and adds parameter resources to the "Parameters"set.

Conflict Checking and Resolution

Once the parameters are identified, the algorithm proceeds to compare resources for potential conflicts. For each pair of resources, it evaluates whether there is a conflict in their resource operations and whether there is a type conflict based on their resource types. Additionally, the algorithm verifies that the second resource (resource2) is in the "Parameters"set and that its associated label is present in the resource operations map. These checks ensure that the algorithm focuses on resources that could potentially conflict.

The checkConflict function in Listing 23, as outlined in the code, embodies this conflict-checking process. It receives various parameters, including flags (hasR2), resource operations (rOp1, rOperation2), resource types (r1, r2), and relevant data structures (resource2, parameters, label2, roMap, handled, keys). This function is responsible for detecting conflicts and taking appropriate actions.

When a conflict is detected, the algorithm checks whether resource2 is already in the "Handled"set to avoid redundant processing. If not, it proceeds to resolve the conflict.

Listing 22 addParameters function of the Parameter Grouping Module. Resource Link: parameterGrouping.ml

```

let rec addParameters ropList parameters resourceMap=
match ropList with
| [] -> parameters
| x :: xs ->
    try let (resource: ResourceClass.resource) = IntMap.find x.r resourceMap in
    if resource.ResourceClass.isParameter then
     let newParameters = Parameters.add x.r parameters in
     addParameters xs newParameters resourceMap
    else
     addParameters xs parameters resourceMap
   with Not_found ->(
     addParameters xs parameters resourceMap
   )
(*newParameters = addParameters ropList parameters resourceMap
   variant ropList
   ensures forall rop. List.mem rop ropList ->
     let resource = resourceMap.IntMap.view rop.r && IntMap.mem x.r resourceMap &&
     resource.ResourceClass.isParameter ->
     exists p.
     Parameters.mem p newParameters && rop.r = p
*)
```

If resource2 is performing an Upgrade operation (op = 5), the algorithm searches for a Shared Before Exclusive operation in subsequent resource groups. If found, it replaces the Upgrade operation with an Exclusive operation. The removeR function is employed to remove resource2 and its associated resource group from the resource operations map.

Handling All Parameters

The conflict resolution process is applied iteratively to all parameters identified earlier. The algorithm processes each parameter, checks for conflicts, and makes necessary adjustments to the resource operations map. This comprehensive approach ensures that conflicts are addressed systematically for all relevant resources within the method.

In the end, the Parameter Grouping phase returns a modified resource operations map that reflects the optimized resource grouping, contributing to enhanced concurrency control and program efficiency.

Two-Phase Locking

The Two-Phase Locking Protocol Verification phase plays a crucial role in ensuring that each method adheres to the two-phase locking protocol, a fundamental concept in concurrency control. If a method deviates from this protocol, it may lead to synchronization issues and decreased program reliability. This phase takes the Resource Operations Map as input and focuses on identifying and correcting potential protocol violations.

Listing 23 CheckConflict function of the Parameter Grouping Module. Resource Link: parameterGrouping.ml

```
1
2
3 let checkConflict hasR2 rOp1 rOperation2 r1 r2 resource2 parameters label2 roMap handled
  ↪  keys=
4   let newROperation2 = ref rOperation2 in
5   let newHandled = ref handled in
6   let newRoMap = ref roMap in
7   if hasR2 == false && conflict rOp1 rOperation2.op && typeConflict r1 r2 &&
8   Parameters.mem resource2 parameters && IntMap.mem label2 roMap then
9     begin
10      newHandled := Handled.add resource2 handled;
11      if rOperation2.op = 5 then
12        let a,b  = outerLoop keys label2 roMap resource2 rOperation2 in
13        newROperation2 := a;
14        newRoMap := b;
15      else ()
16  end
17  else ();
18  !newROperation2, !newRoMap, !newHandled
19
```

Listing 24 Iterate function of the two phase lock Module. Resource Link: twoPL.ml

```
1
2 let rec iterate keys roMap (lastlock: int)=
3   match keys with
4   | [] -> lastlock
5   | x :: xs ->
6     let set = IntMap.find x roMap in
7
8     let rec iterateSet set (lastlock:int) key=
9       match set with
10      | [] -> lastlock
11      | (x:ResourceGroup.resourceGroup) :: xs ->
12      let newLastlock = getLastLock x.ropList lastlock key in
13      iterateSet xs newLastlock key in
14
15    let newLastlock = iterateSet set lastlock x in
16    iterate xs roMap newLastlock
17
```

Listing 25 ropListTrasverse function of the two phase lock Module

```

let rec ropListTrasverse ropList roMap key (rGroup:ResourceGroup.resourceGroup) lastlock
↪  set=
 match ropList with
 | [] -> roMap
 | (x:Roperation.rOp) :: xs ->
    let opCode = x.op in
    if opCode = 7 then (* opCode = release*)
      let newRGroupSet =
      List.filter ( fun x -> x.ResourceGroup.id <> rGroup.ResourceGroup.id ) set in
      let newRoMap = checkNewRGroupSetEmpty newRGroupSet key roMap in
      let newRoMap = ROperationsMapGeneration.insertR (lastlock+1) rGroup newRoMap in
      ropListTrasverse xs newRoMap key rGroup lastlock set
    else ropListTrasverse xs roMap key rGroup lastlock set


```

Determining the Latest Lock Operation

The first task of this phase is to identify the latest lock operation within the method. The algorithm iterates through the keys of the Resource Operations Map, examining the resource groups associated with each key. For each resource group, it searches for the latest lock operation.

The iterate function in Listing 24, as shown in the code snippet, encapsulates this iterative process. It starts by iterating through the keys, then, for each key, it iterates through the resource groups associated with that key, calling the `iterateSet` function. The `iterateSet` function examines each resource group's operations to find the last lock operation. The variable `lastlock` is updated whenever a lock operation is encountered.

Once the algorithm completes this process, it knows the value of `lastlock`, which represents the position of the latest lock operation within the method.

Adjusting Release Operations

After determining the `lastlock` value, the algorithm proceeds to ensure that release operations are correctly positioned in accordance with the two-phase locking protocol. Release operations should occur after the locking phase, i.e., after all locks have been acquired.

The `ropListTraverse` function in Listing 25, highlighted in the code snippet, plays a pivotal role in this adjustment process. It traverses the resource operations list and, when it encounters a release operation (opCode = 7), it removes the corresponding `resourceGroup` from its current key and adds it to the key immediately following the lastlock value. This repositioning ensures that release operations are delayed until after the locking phase.

By iteratively applying this process to all release operations within the method, the algorithm ensures that the two-phase locking protocol is upheld. All release operations

Listing 26 bindingsTrasverse function of the guards Module. Resource Link: guards.ml

```

let rec bindingsTrasverse bindings resourceMap=
  match bindings with
   | [] -> resourceMap
   | x :: xs ->
     let resource = IntMap.find x resourceMap in
     let guard = resource.ResourceClass.guardedBy in
     if guard <> x then
       let resourceList = [x] in
       let resourceList, finalGuard = createResourceList resourceMap guard resourceList in
       let resourceMap = changeGuard resourceMap resourceList finalGuard in
       bindingsTrasverse xs resourceMap
     else
       bindingsTrasverse xs resourceMap


```

are moved to their correct positions, and the Resource Operations Map is updated accordingly.

In summary, the Two-Phase Locking Protocol Verification phase is responsible for enforcing the two-phase locking protocol within methods, guaranteeing correct lock acquisition and release sequences. The provided code snippets and explanations illustrate how the algorithm identifies the latest lock operation and adjusts release operations to adhere to the protocol, thereby promoting safe and effective concurrency control.

Guards Generation

In the Guards Generation module, the focus is on updating and maintaining resource guards, ensuring that each resource correctly reflects its associated locks. Recall that during the resource subsumption phase, when one resource dominated another, the `guardedBy` variable of the dominated resource was changed to the ID of the dominating resource. However, in certain cases where multiple resources are indirectly dominated by the same resource, only one of them is updated with the correct `guardedBy` value.

To rectify this, the algorithm iterates through the resource map and corrects any discrepancies in the `guardedBy` values of resources. It traces back the resource hierarchy to ensure that all dominated resources have their `guardedBy` values updated correctly.

Iterating Through Resources

The `bindingsTraverse` function in Listing 26, as depicted in the code snippet, is responsible for iterating through the resources in the resource map. For each resource, it checks whether the `guardedBy` value matches its own `resourceID`. If these values are different, it means the resource is dominated, and the algorithm proceeds to trace the resources that guard it.

Tracing Resource Hierarchies

Listing 27 CreateResourceList function of the guards Module. Resource Link: guards.ml

```
1
2
3  let rec createResourceList resourceMap guard resourceList =
4    try let (resource:ResourceClass.resource) = IntMap.find guard resourceMap in
5    if resource.ResourceClass.position <> resource.guardedBy then
6      let resourceList = resourceList @ [resource.ResourceClass.position] in
7      createResourceList resourceMap resource.guardedBy resourceList
8    else resourceList, resource.ResourceClass.position
9  with Not_found -> (
10   resourceList, guard
11 )
12
13
14
```

The `createResourceList` function, shown in the code snippet, plays a crucial role in tracing back the resource hierarchy. It takes the `guardedBy` value of the currently analyzed resource and a list containing its `resourceID`.

The algorithm first retrieves the resource associated with the `guardedBy` value from the resource map. It checks whether this resource is also dominated. If it is, it adds it to the `resourceList` and recursively calls itself with the updated `guardedBy` value. This process continues until it reaches a resource that is not dominated.

Updating Resource Guards

Finally, the algorithm takes the resources in the resourceList and updates their guardedBy values to match the resourceID of the resource that dominates all of them. It then updates the resourceMap with these corrected entries. As a result, the resourceMap accurately represents which locks are associated with each resource, facilitating efficient consultation of lock-resource relationships.

In summary, the Guards Generation module ensures the consistency of resource guards and maintains accurate associations between resources and their respective locks. The provided code snippets and explanations illustrate how the algorithm iterates through resources, traces back hierarchies, and updates resource guards to achieve this goal.

Final Remarks

After this process of receiving input, creating the needed maps and updating them in order to simplify the amount of locks actually needed, the implementation that was achieved comes to an end.

With all these modifications to the resource operations map and now, with the resource map also containing correct information about which locks are guarding each resource, the algorithm is ready to start the creation of the graph that is responsible for deadlock detection and its prevention.

6 Evaluation

In this chapter we take a look at which parts of the algorithm have been verified and what tests were implemented in the algorithm.

6.1 Verification

6.1.1 Cameleer maturity

After completing the implementation of the OCaml version of the RC3 algorithm, the next crucial step involved verifying its correctness using the powerful combination of Cameleer and Why3.

During this verification phase, I encountered several challenges with Cameleer. One significant issue was its inability to handle modules that made use of data structures like OCaml Maps or Sets. Given that the algorithm relied heavily on these data structures, I initially focused on verifying the simpler modules, such as Resource Access and Resource Group.

To successfully verify the code, it was often necessary to refactor and adjust certain functions, making them more amenable to Cameleer's analysis.

Over the course of several weeks, I engaged in extensive troubleshooting to identify the specific areas where Cameleer encountered difficulties. Thanks to the invaluable guidance and collaboration with Professor Mário Pereira, we managed to pinpoint these issues, adapt the functions where possible, and even enhance Cameleer's capabilities to handle more complex data structures.

As a result of this dedicated effort, Cameleer's maturity significantly improved, enabling it to effectively verify larger programs involving intricate data structures.

6.1.2 Steps to a successful verification

In the Why3 environment, when I encountered verifications where Cameleer annotations couldn't be instantly verified, it usually indicated complex hypotheses in the goal that the tool struggled to handle independently. To overcome these challenges, I employed three strategies to make progress and achieve successful verifications.

My first approach was to break down the code into smaller, incremental steps. By gradually introducing new operations that required verification, I created numerous small functions in the OCaml implementation. This approach proved to be intuitive and significantly simplified the verification process.

To enhance the verification process, I focused on providing richer and more detailed specifications for the desired functions. Since the algorithm had an incremental nature, with interrelated functions, some verifications became more complex. As a result, I had to ensure that the specifications were comprehensive enough to accurately capture the intended behaviors, even if it made some verifications quite lengthy.

In cases where further fragmentation of the code or specification enrichment wasn't feasible, I directly assisted Why3. This usually happened when dealing with exceedingly complex hypotheses that made it challenging for Why3 to determine if the goal could be proven or not.

To aid the tool in reaching the verification goals, I utilized several operations. The most crucial one involved using the "destruct"command to eliminate complex hypotheses. This command allowed me to split the hypotheses into distinct parts. In more intricate cases, I relied on the "destruct rec"command to break down hypotheses into smaller portions.

Additionally, the "eliminate let"command was highly useful in removing instances of OCaml "let...in"structures from the specification. This helped me avoid dependencies on variables constructed within these structures, making it easier to split hypotheses.

Furthermore, I found the "instantiate"command valuable when dealing with iterations. By instantiating an element within loops, I could facilitate the proofs since Why3 struggled with highly complex loops. However, I must admit that this command presented a challenge I couldn't successfully resolve. Nevertheless, the combination of these approaches significantly contributed to the success of the verification process.

6.1.3 Results

In Table 6.1, I present the results of the algorithm verification, highlighting the different levels of verification achieved for each part.

The green color indicates functions that have undergone full verification with Cameleer annotations. These functions have verified pre- and/or post-conditions, along with variants for handling iterations within the function. Notably, the Resource Class, Access, Resource Group, Mapping, and Resource Subsumption modules have successfully passed all verification criteria, ensuring the correctness of their behaviors.

Moving on to the Resource Operations and Parameter Grouping module, I managed to verify their initial functions and some essential logical behaviors using Cameleer. However, completing the verification for these modules proved challenging. Some functions encountered obstacles, such as unverified pre- or post-conditions, which hindered reaching full verification.

Module	Function	Module	Function	Module	Function
ResourceClass	setIsParameter	RSubsumption	dominatesCheck	ParameterGrouping	ropListEmpty
ResourceClass	isField	RSubsumption	dominates	ParameterGrouping	removeR
ResourceClass	isCast	RSubsumption	methodCycle	ParameterGrouping	checkSBE
ResourceClass	isMethodcall	RSubsumption	resourceMapUpdate	ParameterGrouping	outerLoop
ResourceClass	isObject	RSubsumption	logic	ParameterGrouping	checkConflict
Access	resourceAccessCons	RSubsumption	findSndResource	ParameterGrouping	newRoMapCreation
Access	hasReadAccess	RSubsumption	sndResource	ParameterGrouping	ropListCycle
Access	hasWriteAccess	RSubsumption	fstResource	ParameterGrouping	checkGrouping
Access	replaceFirstWrite	RSubsumption	subsumption	ParameterGrouping	condition2ndResource
Access	replaceLastWrite	ROperations	createNewRGroup	ParameterGrouping	resource1Cycle
Access	getFirst	ROperations	insert	ParameterGrouping	addParameters
Access	getLast	ROperations	insertRO	ParameterGrouping	findRGroup
ResourceGroup	add	ROperations	noReadLocks	ParameterGrouping	addToParameters
ResourceGroup	getResult	ROperations	firstReadORlastWrite	ParameterGrouping	parameterGrouping
ResourceGroup	remove	ROperations	labelsVerification	Guards	changeGuard
Mapping	firstElement	ROperations	firstWriteBefore	Guards	createResourceList
Mapping	computeResourceMa	ROperations	firstReadBefore	Guards	bindingsTrasverse
Mapping	tryRes	ROperations	computeROperations	Guards	locks
Mapping	classNotFound	ROperations	getPair	twoPL	checkNewRGroupSetEmpty
Mapping	computeLine	ROperations	computeBindings	twoPL	ropListTrasverse
Mapping	computeResource	ParameterGrouping	conflict	twoPL	findGroups
Mapping	isResourceParameter	ParameterGrouping	removeFromGroupSet	twoPL	getLastLock
RSubsumption	sub	ParameterGrouping	addNewRGroup	twoPL	iterate
				twoPL	twoPhaseLocking

Figure 6.1: Table of which functions have a verified specification (green) have a specification very close to be accepted or a suggested verification (yellow) don't have a specification (red).

The complexity of the algorithm, especially in modules with multiple iterations through maps, contributed to the difficulties faced during verification. Despite employing helpful tools like the `destruct` command and other hypothesis simplification commands in Why3, some goals remained hard for Why3 and Cameleer to verify adequately.

When functions have a yellow marking in the table, it means that they possess suggested verifications that are nearly complete with Why3. However, a few branches of the verification process encountered some issues, preventing full completion.

Unfortunately, in the Guards and TwoPL modules, I encountered challenges in verifying any functions. Although the initial functions were promisingly close to verification, delving into the deeper functions within these modules proved problematic.

Thus, the red color marks functions where I could not conclude any verification. These functions heavily depend on specifications at a more bottom level, but the lack of suggested specifications for their behavior made it difficult to complete the verification process.

In summary, most parts of the implemented program have associated verified Cameleer comments or very close suggestions for verification. However, certain functions' complexity and issues in specific modules posed challenges in achieving full verification. Despite

these obstacles, the verified sections contribute to ensuring the overall correctness of the OCaml version of the RC3 algorithm.

6.2 OUnit tests

Due to the intricate nature of certain modules and functions, achieving approved and correct verifications for the entire program proved to be a challenging task. Consequently, it became imperative to adopt a rigorous testing approach to ensure the reliability and correctness of the OCaml implementation.

For this purpose, the OUnit2 testing framework was selected, drawing inspiration from the renowned JUnit framework used in Java. OUnit2 provides a comprehensive set of functions that enable the creation and execution of test cases, organization into test suites, and generation of detailed reports. Test cases are essentially defined as functions that return a boolean value, indicating whether the test has successfully passed or encountered a failure. On the other hand, test suites are designed as modules containing test cases or other test suites, allowing for a hierarchical structure to manage tests effectively.

The adoption of OUnit2 opened up a wealth of testing possibilities for the OCaml program. It allowed for targeted and systematic testing of functions with significant behaviors, covering a wide range of scenarios and edge cases. By designing test cases that accurately mimic real-world scenarios, it was possible to gain confidence in the correctness of the algorithm.

A concrete example of a test case, the `checkConflictTest` 28, exemplifies the approach used for functions involving OCaml maps. In this process, the first step is to strategically insert the necessary elements into the maps, creating a test state that is meaningful for evaluation. Subsequently, the algorithm's specific function is invoked, and the resulting output is meticulously examined to verify if the intended changes have taken place as expected. By conducting such thorough tests, we can gain insights into the algorithm's behavior and detect potential issues or discrepancies.

Organizing the test cases efficiently is crucial for effective testing. To achieve this, the test cases are grouped together in test suites corresponding to their respective modules. These test suites not only execute the functions but also store the desired results, facilitating easy comparison and analysis.

The OUnit2 framework provides comprehensive reporting capabilities, allowing us to review the test results in a clear and concise manner. By running the executable generated through "dune,"we obtain a detailed report indicating the number of successful tests and any instances where the tests did not yield the expected results.

Table 6.2 provides an overview of the functions and their corresponding modules that underwent OUnit testing, with particular emphasis on functions that lacked successful verification. These functions are of utmost importance, as they play a critical role in the algorithm and must be thoroughly validated to ensure accuracy and reliability.

Listing 28 CheckConflictTest function of the ParaGroupingTest Module

```
let checkConflictTest =
    let hasR2 = false in
    let rOp1 = {op = 4; r = 2} in
    let rOp2 = {op = 5; r = 2} in
    let r1 = Resource.ResourceClass.make_resource 3 30 0 false 3 3 true in
    let r2 = Resource.ResourceClass.make_resource 6 30 0 false 3 6 true in
    let parameters = ParameterGrouping.Parameters.empty in
    let label2 = 4 in
    let roMap = IntMap.empty in
    let handled = ParameterGrouping.Handled.empty in
    let keys = [3;4;5] in
    let parameters = ParameterGrouping.Parameters.add r1.position parameters in
    let parameters = ParameterGrouping.Parameters.add r2.position parameters in
    let rGroup1 ={id = 1; ropList = [rOp1]} in
    let rGroup2 = { id = 2; ropList = [rOp2]} in
    let roMap = IntMap.add label2 [rGroup2] roMap in
    let roMap = IntMap.add 2 [rGroup1] roMap in
    let _, _ , newHandled = ParameterGrouping.checkConflict hasR2 rOp1.op rOp2 r1 r2
    ↪  parameters label2 roMap handled keys in
    if ParameterGrouping.Handled.cardinal newHandled >
    ParameterGrouping.Handled.cardinal handled then
      true
    else false

let parameterGroupingTests = "test suite for subsumeResource" >::: [
  "CheckConflict Test where there is a type conflict" >:: (fun _ -> assert_equal
  true
  (checkConflictTest) ~printer:string_of_bool);

]

let _ = run_test_tt_main parameterGroupingTests



```

Module	Function
Parameter Grouping	RemoveR
Parameter Grouping	CheckSBE
Parameter Grouping	CheckConflict
Parameter Grouping	AddToParameters
Guards	ChangeGuard
Guards	CreateResourceList
Guards	bindingsTrasverse
twoPL	checkNewRGroupSetEmpty
twoPL	ropListTrasverse
twoPL	getLastLock
twoPL	iterate

Figure 6.2: Table of which functions have OUnit tests testing their behaviour.

By combining the OUnit tests for functions without completed verifications and the verified Cameleer annotations for the rest, we can confidently assert that the OCaml version of the RC3 algorithm behaves exactly as intended. This meticulous and comprehensive approach to testing and verification instills a high level of confidence in the correctness of the program's implementation, contributing to its overall robustness and reliability.

7 Conclusions

In this chapter we take a look at what this thesis achieved and what future work is required in order to finish the implementation and property verification of the RC^3 model.

7.1 Contributions

The main research question of my thesis was whether it is possible to develop a locking policy strategy that ensures deadlock-freedom, starvation-freedom, and also performs comparably to finely-tuned locked programs. To explore this question, I made several important contributions that brought me closer to finding an answer.

The first crucial step was to understand the RC^3 model and then structure the algorithm effectively in the OCaml language. While I could have used classes with the class module for an object-oriented approach, I found that Cameleer, the verification tool, worked better with a functional structured approach. Additionally, I conducted a study to identify better-suited data structures for the algorithm's development.

With a clear structure in mind, the first major contribution came through implementing the RC^3 algorithm in OCaml. I successfully implemented modules ranging from the simpler ones like Resource, Resource Access, and Resource Group, to more complex ones like Parameter Grouping and Resource Subsumption. This implementation formed a significant portion of the algorithm, involving graph creation and analysis. Without this implementation, it would have been impossible to proceed with property verification.

During the algorithm implementation, I used Cameleer to verify the parts that were possible to verify. However, Cameleer was not yet equipped to handle modules containing Map functors. So, I focused on verifying concept modules like Resource and Resource Access first. After adding the necessary OCaml modules, including Map functors, to Cameleer, I tackled issues related to these constructs in the implementation code.

A notable contribution arose from enhancing Cameleer's capabilities to handle complex algorithms involving intricate data structures. This involved weeks of troubleshooting and collaboration with Professor Mário Pereira to resolve incompatibilities between Cameleer and certain OCaml constructs. As a result, Cameleer is now well-prepared to

handle complex data structures and programs in the future.

Once the Cameleer troubleshooting was completed, I began the verification process. Following the algorithm steps, I verified modules like Mapping and Resource Subsumption, eventually reaching the Two Phase Locking and Guards module.

Another major contribution is the completion of verification for many functions in the implementation, as described in the previous chapter. For the functions I couldn't verify, I conducted extensive testing using the OUnit2 OCaml library. Moreover, some of these functions have near-complete Cameleer verifications.

In summary, my thesis achieved a successful implementation of a highly complex algorithm in a language quite different from Java. I also significantly improved the verification tool Cameleer, which was initially ill-equipped for handling such complex algorithms. Additionally, a large portion of the implementation underwent successful verification, with supplementary tests for the remaining parts that couldn't be fully verified.

7.2 Future Work

In the future work of this thesis, there are three main areas that require attention and further development.

As mentioned earlier, the implementation of the OCaml version of the RC^3 algorithm is almost complete. However, the verification of some functions is still pending. Some of these functions are very close to being verified, and they might only need a few adjustments in either the code of the function itself or the specification. On the other hand, some functions do not have any verification done at all, and in such cases, it will be necessary to create a comprehensive verification for their behavior.

Once the verification of the OCaml version of RC^3 is completed, the next step would be to examine what remains to be implemented. Looking at the algorithm in 2, the next step after my implementation involves taking the output and creating a graph in the `addOperationsToGraph` function (line 12). This graph will represent the program's resource dependencies.

After the graph is created, the subsequent step would be to implement a cycle that analyzes the graph and breaks any circular dependencies that might cause a deadlock.

When the OCaml version is finally finished, the last task will be to complete the proof of the algorithm. This involves verifying that all functions have the correct behavior and proving that essential properties such as deadlock-freedom, starvation-freedom, and serializability are indeed present in the RC^3 model using Cameleer verification.

In addition to these specific areas, there are broader aspects of future work that can be explored. One possible avenue is to revise some of the verification for simpler proofs. Some functions verification ended up being too extensive and can most likely be improved in order to capture the function behavior in a simpler and more concise way.

Another important direction for future work is to test the RC^3 OCaml algorithm in real-world complex programs. Testing the algorithm in practical settings and comparing

Algorithm 2 RC^3 lock inference and deadlock prevention strategy algorithm

```
1: g ← graph
2: for c ∈ Classes do
3:     for m ∈ methods(c) do
4:         MRI ← getMethodResourceInfo(m)
5:         RA ← getResourceMap(MRI)
6:         Cs ← getMethodCalls(MRI)
7:         RA ← RA ∪ resolveMethodCalls(Cs)
8:         RA ← resourceSubsumption(RA)
9:         RO ← RA2RO(RA)
10:        RO ← parameterGrouping(RO)
11:        RO ← 2pl(RO)
12:        addOperationsToGraph(RO, g)
13:    end for
14: end for
15: while (sccs ← getSCCs(g)) ≠ ∅ do
16:    for scc ∈ sccs do
17:        (b, scc) ← canBreakCycle()
18:        if b then
19:            updateGraph(g, scc)
20:        else
21:            colapseSCC(g, scc)
22:        end if
23:    end for
24: end while
```

its output to the Java counter part could provide valuable insights and help identify potential areas for improvement.

To sum it up, the future work of this thesis entails completing the verification of the OCaml version of the RC^3 algorithm, implementing the graph creation and deadlock-breaking steps, and then conducting a comprehensive proof of the entire algorithm. Beyond these specific tasks, there are opportunities to improve the verification and explore real-world applications to test the OCaml version.

In conclusion, there is still some future work to do. However, thanks to this thesis achievements, a future researcher will find it much easier to make progress and reach the goals of this work. My efforts have laid a strong foundation for someone to build upon and showed that it is possible to reach a conclusion for the properties of the RC^3 algorithm.

Bibliography

[1] K. Bhargavan et al. "A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021, pp. 523–542 (cit. on p. 9).

[2] A. Charguéraud et al. "GOSPEL—Providing OCaml with a Formal Specification Language". In: *Formal Methods – The Next 30 Years*. Ed. by M. H. ter Beek, A. McIver, and J. N. Oliveira. Cham: Springer International Publishing, 2019, pp. 484–501. ISBN: 978-3-030-30942-8 (cit. on p. 27).

[3] S. Cherem, T. Chilimbi, and S. Gulwani. "Inferring Locks for Atomic Sections". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 304–315. ISBN: 9781595938602. DOI: 10.1145/1375581.1375619. URL: https://doi.org/10.1145/1375581.1375619 (cit. on pp. 12–14, 17, 21, 22).

[4] J. Dolby et al. "A data-centric approach to synchronization". In: *ACM Trans. Program. Lang. Syst.* 34.1 (2012), 4:1–4:48. DOI: 10.1145/2160910.2160913. URL: https://doi.org/10.1145/2160910.2160913 (cit. on p. 22).

[5] M. Emmi et al. "Lock Allocation". In: New York, NY, USA: Association for Computing Machinery, 2007. ISBN: 1595935754. DOI: 10.1145/1190216.1190260. URL: https://doi.org/10.1145/1190216.1190260 (cit. on pp. 13, 18, 20, 21).

[6] J.-C. Filliâtre and A. Paskevich. "Why3—where programs meet provers". In: *European symposium on programming*. Springer. 2013, pp. 125–128 (cit. on p. 10).

[7] J. Grande, G. Boudol, and M. Serrano. "Jthread, a deadlock-free mutex library". In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*. Ed. by M. Falaschi and E. Albert. ACM, 2015, pp. 149–160. ISBN: 978-1-4503-3516-4. DOI: 10.1145/2790449.2790523. URL: https://doi.org/10.1145/2790449.2790523 (cit. on pp. 6, 13, 15, 17–22).

[8] C. Hammer et al. “Dynamic Detection of Atomic-Set-Serializability Violations”. In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE ’08. Leipzig, Germany: Association for Computing Machinery, 2008, pp. 231–240. ISBN: 9781605580791. DOI: 10.1145/1368088.1368120. URL: https://doi.org/10.1145/1368088.1368120 (cit. on pp. 7, 26).

[9] M. Hicks, J. S. Foster, and P. Pratikakis. “Lock inference for atomic sections”. In: *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. 2006 (cit. on pp. 13, 14, 21).

[10] T. Hubert and C. Marché. “Separation analysis for deductive verification”. In: *Heap Analysis and Verification (HAV’07)* (2007), pp. 81–93 (cit. on p. 9).

[11] B. Jacobs and F. Piessens. *The VeriFast program verifier*. Tech. rep. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2008 (cit. on p. 10).

[12] B. Jacobs, J. Smans, and F. Piessens. “The VeriFast program verifier: A tutorial”. In: *Tech. Rep.* (2014) (cit. on p. 10).

[13] E. G. C. Jr., M. J. Elphick, and A. Shoshani. “System Deadlocks”. In: *ACM Comput. Surv.* 3.2 (1971), pp. 67–78. DOI: 10.1145/356586.356588. URL: https://doi.org/10.1145/356586.356588 (cit. on p. 6).

[14] K. R. M. Leino. “Dafny: An Automatic Program Verifier for Functional Correctness”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by E. M. Clarke and A. Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4 (cit. on p. 9).

[15] D. B. Lomet. “Process Structuring, Synchronization, and Recovery Using Atomic Actions”. In: *Proceedings of an ACM Conference on Language Design for Reliable Software*. Raleigh, North Carolina: Association for Computing Machinery, 1977, pp. 128–137. ISBN: 9781450373807. DOI: 10.1145/800022.808319. URL: https://doi.org/10.1145/800022.808319 (cit. on p. 7).

[16] D. Marino et al. “Detecting deadlock in programs with data-centric synchronization”. In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 322–331. DOI: 10.1109/ICSE.2013.6606578 (cit. on p. 22).

[17] B. McCloskey et al. “Autolocker: Synchronization Inference for Atomic Sections”. In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’06. Charleston, South Carolina, USA: Association for Computing Machinery, 2006, pp. 346–358. ISBN: 1595930272. DOI: 10.1145/1111037.1111068. URL: https://doi.org/10.1145/1111037.1111068 (cit. on pp. 6, 13, 15, 17–21).

[18] P. Monteiro, J. Lourenço, and A. Ravara. "Uma análise comparativa de ferramentas de análise estática para deteção de erros de memória". In: *CoRR* abs/1807.08015 (2018). arXiv: 1807.08015. URL: http://arxiv.org/abs/1807.08015 (cit. on p. 8).

[19] D. Neves and H. Paulino. "Condition-based synchronization in data-centric concurrency control". In: *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*. Ed. by J. Hong et al. ACM, 2022, pp. 1268–1275. DOI: 10.1145/3477314.3507120. URL: https://doi.org/10.1145/3477314.3507120 (cit. on p. 3).

[20] H. Paulino et al. "From Atomic Variables to Data-Centric Concurrency Control". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: Association for Computing Machinery, 2016, pp. 1806–1811. ISBN: 9781450337397. DOI: 10.1145/2851613.2851734. URL: https://doi.org/10.1145/2851613.2851734 (cit. on pp. 2, 3).

[21] M. Pereira and A. Ravara. "Cameleer: A Deductive Verification Tool for OCaml". In: *Computer Aided Verification*. Ed. by A. Silva and K. R. M. Leino. Cham: Springer International Publishing, 2021, pp. 677–689. ISBN: 978-3-030-81688-9 (cit. on p. 27).

[22] M. Vaziri et al. "A Type System for Data-Centric Synchronization". In: *Proceedings of the 24th European Conference on Object-Oriented Programming*. ECOOP'10. Maribor, Slovenia: Springer-Verlag, 2010, pp. 304–328. ISBN: 3642141064 (cit. on p. 22).

[23] Y. Wang et al. "The Theory of Deadlock Avoidance via Discrete Control". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '09. Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 252–263. ISBN: 9781605583792. DOI: 10.1145/1480881.1480913. URL: https://doi.org/10.1145/1480881.1480913 (cit. on pp. 13, 16, 17, 19, 21).