Filipe Silva Meirim

MSc Student

Static Verification of Cloud Applications with Why3

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Carla Ferreira, Associate Professor,
NOVA University of Lisbon

Co-adviser: Mário Pereira, Post-doctoral researcher,
NOVA University of Lisbon

Examination Committee

Chairperson: António Ravara
Raporteur: Jorge Sousa Pinto
Member: Carla Ferreira

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2019

Static Verification of Cloud Applications with Why3

Copyright © Filipe Silva Meirim, Faculty of Sciences and Technology, NOVA University Lisbon.
The Faculty of Sciences 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.

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Abstract

Nowadays large-scale distributed applications rely on replication in order to improve their services. Having data replicated in multiple datacenters increases availability, but it might lead to concurrent updates that violate data integrity. A possible approach to solve this issue is to use strong consistency in the application because this way there is a total order of operations in every replica. However, that would make the application abdicate of its availability. An alternative would be to use weak consistency to make the application more available, but that could break data integrity. To resolve this issue many of these applications use a combination of weak and strong consistency models, such that synchronization is only introduced in the execution of operations that can break data integrity.

To build applications that use multiple consistency models, developers have the difficult task of finding the right balance between two conflicting goals: minimizing synchronization while preserving data integrity. To achieve this balance developers have to reason about the concurrent effects of each operation, which is a non-trivial task when it comes to large and complex applications.

In this document we propose an approach consisting of a static analysis tool that helps developers find a balance between strong and weak consistency in applications that operate over weakly consistent databases. The verification process is based on a recently defined proof rule that was proven to be sound. The proposed tool uses Why3 as an intermediate framework that communicates with external provers, to analyse the correctness of the application specification.

Our contributions also include a predicate transformer and a library of verified data types that can be used to resolve commutativity issues in applications. The predicate transformer can be used to lighten the specification effort.

Keywords: Replication, Data Integrity, Static Analysis, Consistency, Synchronization, Why3

Resumo

Atualmente aplicações de larga escala dependem de replicação de modo a poderem melhorar os seus serviços. Ter dados replicados em múltiplos *datacenters* melhora a disponibilidade do sistema, mas ao mesmo tempo podem haver atualizações concorrentes que violam a integridade dos dados. Uma possível abordagem para resolver esta situação é utilizar consistência forte sobre a aplicação, porque deste modo existe sempre uma ordem total das operações em todas as réplicas. No entanto, isso faz com que a aplicação abdique da sua disponibilidade. Uma alternativa é usar consistência fraca para tornar a aplicação mais disponível, mas, isso pode quebrar a integridade dos dados. Para resolver esta questão, diversas aplicações usam uma combinação de modelos de consistência fraca e forte. Neste caso, sincronização apenas é introduzida no sistema quando são executadas operações que podem colocar a integridade dos dados da aplicação em risco.

Para se construir aplicações que usam vários modelos de consistência, os *developers* têm a difícil tarefa de encontrar o equilíbrio entre dois objetivos contraditórios: minimizar sincronização enquanto se preserva a integridade dos dados. Para atingir este equilíbrio, os *developers* têm de pensar nos efeitos concorrentes de cada operação, sendo uma tarefa não-trivial quando se tratam de aplicações complexas e de larga escala.

Neste trabalho apresentamos uma abordagem que consiste numa ferramenta de análise estática para ajudar *developers* a encontrar o equilíbrio entre consistência fraca e forte em aplicações que operam sobre bases de dados com consistência fraca. O processo de verificação é baseado numa regra de prova que foi previamente provada correta. A ferramenta proposta usa a plataforma Why3 para comunicar com *provers* externos para analisar a correção da especificação duma aplicação.

As nossas contribuições também incluem um *predicate transformer* e uma biblioteca de tipos de dados verificada, que podem ser usados para resolver problemas de comutatividade. O *predicate transformer* pode ser usado para simplificar o esforço de especificação.

Palavras-chave: Replicação, Integridade dos dados, Análise estática, Consistência, Sincronização, Why3

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Context 1
1.2 Motivation 2
1.3 Contributions 3
1.4 Document Structure 4

2 Background 5

2.1 Analysis of Programs 5
2.2 Why3 Framework 6
2.3 Consistency Models 8
2.3.1 Operation Consistency Models 8
2.3.2 Transaction Consistency Models 9
2.4 CRDTs 10
2.5 Hoare's Logic 12
2.6 Design by Contract 13
2.7 Predicate Transformers 14
2.7.1 Weakest Precondition Calculus 15
2.7.2 Strongest Postcondition Calculus 16
2.8 CISE proof rule 17

3 CISE3 Architecture 19

3.1 CISE3 Overview 19
3.2 Proof Obligations Component 20
3.3 Token System Component 24
3.4 CRDTs library Component 26
3.5 Strongest Postcondition Component 26

4 Experimental Evaluation 29

4.1 Banking Application 29

4.2 Auction Application 35
4.3 Courseware Application 40

5 Related Work 51
5.1 Quelea 51
5.2 Q9 52
5.3 Repliss 53
5.4 Hamsaz 54
5.5 CISE tool 55
5.6 CEC tool 56
5.7 Tool Comparison 56

6 Conclusion 59
6.1 Discussion 59
6.2 Future Work 60
6.3 Final Remarks 60

Bibliography 61

A CRDTs library 67
A.1 Disable Once Flag 67
A.2 Enable Once Flag 67
A.3 Remove-wins Set 68
A.4 Add-wins Set 69

List of Figures

1.1 Concurrent execution of the withdraw operation. 3

2.1 Why3 as an intermediate language. 7
2.2 Remove-wins Set CRDT in Why3. 12
2.3 BNF of the ESC/Java language. 16

3.1 CISE3 architecture. 21
3.2 Generic program in Why3. 21
3.3 Generated commutativity and stability analysis function for operations `f` and `g`. 22
3.4 Generated stability analysis function for operations `f` and `g`. 24
3.5 BNF that represents our provided token system specification language. . . . 25

4.1 Specification and implementation of the banking application. 30
4.2 Statistics regarding the proof of the banking application. 30
4.3 Generated commutativity analysis function for the bank application. 32
4.4 Generated stability analysis functions for the bank application. 33
4.5 Statistics of the proof effort for the generated functions for the bank application. 34
4.6 Assume expression of `withdraw_stability` given the token system. 35
4.7 Specification and implementation of the auction application. 36
4.8 Statistics regarding the proof of the auction application. 37
4.9 Generated commutativity analysis function for the auction application. . . . 38
4.10 Generated stability analysis functions for the auction application. 39
4.11 Statistics of the proof effort of the generated functions for the auction application. 40
4.12 Specification of the courseware application. 41
4.13 Statistics regarding the proof of the auction application. 42
4.14 Generated commutativity analysis function for `enroll` and `remCourse`. 44
4.15 Courseware application with a CRDT. 45
4.16 Generated stability analysis functions for the courseware application. 46
4.17 Statistics of the proof effort of the generated functions for the courseware application. 48
4.18 Modified assume expression from `enroll_remCourse_commutativity`. 49

A.1 Disable Once flag in Why3. 68
A.2 Enable Once flag in Why3. 69
A.3 Remove-wins Set CRDT in Why3. 70
A.4 Add-wins Set CRDT in Why3. 70

List of Tables

2.1 Weakest Precondition Calculus Rules. 16
2.2 Strongest Precondition Calculus Rules. 17

5.1 Comparison between the studied tools. 58

Chapter 1

Introduction

This chapter serves to contextualize and motivate the work from this thesis. Additionally this chapter also presents our contributions and the structure of this document.

1.1 Context

Nowadays global-scale distributed applications, like social networks or online games, rely on geo-replicated storage systems in order to improve user experience. These geo-replicated storage systems consist of various replicas, scattered around the world, storing copies of logic and data from the application. This enables operations to have low latency since the requests are routed to the closest data centre, making the system more available to the user. However, when updates occur simultaneously over different replicas, data integrity can be compromised. A possible solution for this issue would be to introduce synchronization in the system, by using a strong consistency model. On the other hand the system can employ weak consistency. When a system uses weak consistency, if it receives a request it does not wait for the synchronization between replicas. This way the system becomes more available and its operations have low latency because after a replica processes a request it sends an answer to the client immediately. However, there is a possible loss of the correctness of the application due to undesirable concurrency updates that can lead to the violation of data integrity invariants. On the other hand, if strong consistency is used, the system behaves as a single centralised replica, thus guaranteeing data integrity. However, due to this behaviour the availability of the system will decrease [2, 37]. That being said, it is clear that each consistency model has its benefits and downfalls. So, the goal is to build an application that has a high availability combined with the guarantee of data integrity preservation.

In order to build applications that enjoy the advantages of weak and strong consistency, it was proposed that geo-replicated systems could use a combination of these models [2, 3]. Specifically, the approach is to use strong consistency whenever the correctness of the application is at risk and leverage the benefits of weak consistency when the concurrent execution is safe. However, finding a balance between weak and strong consistency is a non-trivial task [27]. This happens because the programmer needs to reason about the concurrent effects of every operation, and decide which operations require synchronization in order to assure the correctness of the application [2, 28].

Recently some static analysis tools have been proposed that assist the programmers in determining which operations of an application need synchronization in a distributed setting. These tools receive a specification of an application and then determine the adequate consistency model, such that its availability is maximized and data integrity is guaranteed.

1.2 Motivation

As mentioned before, large-scale applications rely on geo-replication to improve their availability and user experience. However, with replication comes the issue of concurrent operations breaking data integrity. As a motivation let us consider an example of a simple bank application where a client can only withdraw and deposit money from an account, and where each account must always have a non-negative balance. Let us also consider we have two replicas where the balance of an account is 100€. If there are two concurrent withdraw operations that are trying to remove 100€ from that account then both operations are allowed because they are not trying to remove more money than the existent amount in the account. So when the effects of the operations are propagated, we reach a state where in both replicas the balance of that account is -100€. This breaks the property that every account must always have a non-negative balance. To illustrate this example we present Figure 1.1.

Applications like Amazon's Dynamo [15] use weaker consistency semantics, thus providing high availability but at the possible cost of losing data integrity. Other applications like Yahoo's PNUTS [12] avoid data inconsistencies by requiring all operations that update the system to be funnelled through a primary site, but this approach makes them less available. So, we can observe that both concepts have their advantages and disadvantages. However, recent approaches [2, 37, 50] have showed that it is possible to achieve both data integrity and high availability by using a combination of strong and weak consistency models, with the objective of leveraging the advantages of both. However, finding the balance between these models is not trivial. If strong consistency is used in too many places then the availability of the system will decrease but if used in too few places then data integrity can be at fault. So, the main goal in these approaches is to find the places in the system where data integrity is at risk and use strong consistency. In the remaining

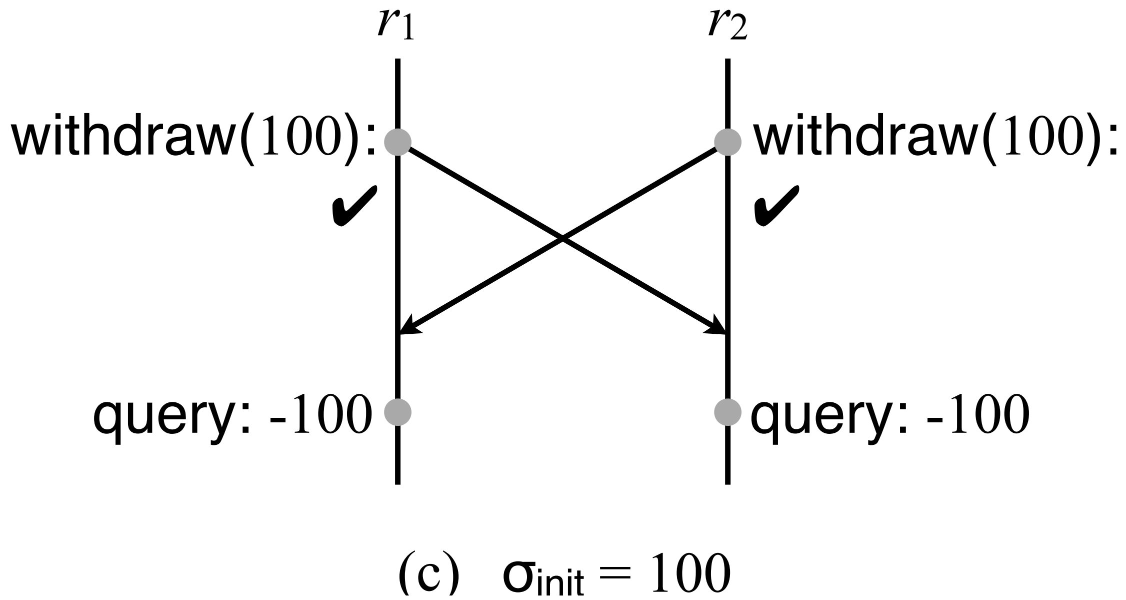

Figure 1.1: Concurrent execution of the withdraw operation.

components of the system the programmer employs weak consistency because the data integrity is not at risk.

1.3 Contributions

The work of this thesis focuses on building a static analysis tool that can automatically examine applications operating over weakly-consistent replicated databases. The goal of our approach is to analyse an application and understand where to use synchronization, in order to build an application that maximizes availability while guaranteeing data integrity.

In our approach we propose that the programmer provides a sequential specification of the application, for our tool to analyse. In the provided input the programmer needs to specify the state of the application and its integrity invariants as well as the operations of the application. Our approach is based on the proof obligations from the proof rule proposed by CISE [28]. The result from these proof obligations are the places in the system where conflicts can occur, and lead to the possible loss of data integrity. This way the programmer knows where to introduce synchronization in the system.

In order to make the verification process more robust, we implemented our tool over the deductive verification framework Why3 [19]. By implementing our tool over this framework, the programmer does not have to worry about said process, only needing to worry about writing the specification. And to aid the programmer with writing the specification we present a strongest postcondition predicate transformer. This predicate transformer automatically generates strongest postconditions, thus making the specification process less cumbersome for the programmer.

Another contribution is a library of Conflict-free Replicated Data Types (CRDTs) [47] verified and implemented using the Why3 framework. CRDTs are data types that guarantee data convergence in a self-established way, by means of a deterministic conflict resolution policy. This library of CRDTs can be used by the programmer to erase commutativity issues in applications, in a way that reduces the amount of introduced synchronisation.

Finally we present as a contribution a set of case studies that were implemented and verified using our tool. This set of case studies helped us validate our approach.

The work developed in the context of this thesis also produced two articles. The first one "CISE3: Verificação de aplicações com consistência fraca em Why3" presented in INForum 2019 in Guimarães.

1.4 Document Structure

The remaining of this thesis is organized as follows. In Chapter 2 we provide some preliminary concepts related to the work of this thesis. In Chapter 3 we describe the main concepts of our prototype. In Chapter 4 we show three complete case studies used to validate our approach. In Chapter 5 we present some static analysis tools that are related to ours, also showing some comparisons between them. Finally in Chapter 6 we discuss and summarize our approach, also presenting some future work.

Chapter 2

Background

This chapter focuses on presenting the key concepts involved in this thesis. The main concepts resolve around analysis of programs, the Why3 framework, consistency models, CRDTs, Hoare's logic, design by contract and predicate transformers.

2.1 Analysis of Programs

An important aspect regarding software construction is guaranteeing its correctness. By correctness we mean that the program satisfies its safety and liveness properties. In order to know what is the expected behaviour of a program there should be a specification defined by the developers. Specifications are used because they are simple, making it easier to analyse. However, specifications are usually incomplete and do not show the full set of correct states a program can have, but all the states it covers must be correct.

To analyse a program's behaviour, there are three kinds of analyses that can be performed: dynamic analysis, static analysis or a combination of both. Dynamic analysis techniques consist in verifying a program during its execution. Approaches to this technique can be some form of testing like unit/coverage testing, or runtime monitors to constantly check that the program does not violate correctness. Theses violations however, are only visible after an unexpected behaviour occurs. Another kind of dynamic analysis is runtime verification, which consists in checking formal properties during runtime [48]. Analysing a program with dynamic analysis has some disadvantages, like the introduction of an overhead to the system and the fact that it might not detect all the errors. Additionally, it does not guarantee the absence of errors because, the program may pass one test suite but there might be another test that could break the program. As an alternative to dynamic analysis, we have static verification. Static verification describes the verification process done at compile time, by verifying the source code of the

program instead of executing it. This technique can ensure the absence of errors of a well defined kind, for example no null dereferences. It can also verify some correctness properties like functionality and security. Another advantage of static analysis is that it does not introduce any runtime overhead. However, this approach has some limitations like the amount of time it consumes and the possibility of detecting false positives. A false positive occurs when an error is said to exist when it does not. These false positives occur because the static analysis verifies a program given a set of rules provided by the programmer, so if they are not well defined then false positives can be detected.

The purpose of these two techniques, is to build high quality software, and can be combined to get the best of both approaches. Normally when combining these techniques the process starts with static analysis, proceeded by a dynamic analysis process done having in mind the information obtained with the static analysis. The dynamic analysis is more effective because with the information obtained by the static analysis it is possible to restrict the sources of potential anomalies, thus having a decrease in the overhead introduced in the system. By restricting the possible sources of errors with static analysis the dynamic analysis will not be so broad and the test suites can be more precise [18]. Also, it is widely acknowledged that statically verifying every invariant is a difficult and time consuming task. Dynamic analysis can improve on such situation, since it delegates invariant checking for runtime execution. One example of a tool that combines both these analysis techniques is Spark2014 [29].

2.2 Why3 Framework

Why3 is a framework that uses deductive program verification which is the "process of turning the correctness of a program into a mathematical statement and then proving it"[19]. Why3's architecture is divided in two parts: a purely logical back-end and a programs front-end [19]. The front-end part receives files that contain a list of modules, from which verification conditions will be extracted, and subsequently sent to external theorem demonstrators.

This tool provides a programming language called WhyML, which has two purposes: writing programs and the formal specifications of their behaviours. Programs written in WhyML have at their disposal a first order language with some features commonly found in functional languages, like pattern-matching, algebraic types and polymorphism. Also it offers some imperative features like records with mutable fields and exceptions. The logic used to write formal specifications is an extension of the first-order logic with polymorphic types, algebraic types, inductive predicates, and recursive definitions, as well as a limited form of higher-order logic [22]. Another useful feature that WhyML provides is *ghost code* that has the main purpose of facilitating specifications. This type of code is similar to normal code in the sense that it is parsed, type checked and turned into verification conditions in the same way [19]. A particularity of ghost code is that it can be removed from a program without changing its final result. This happens because

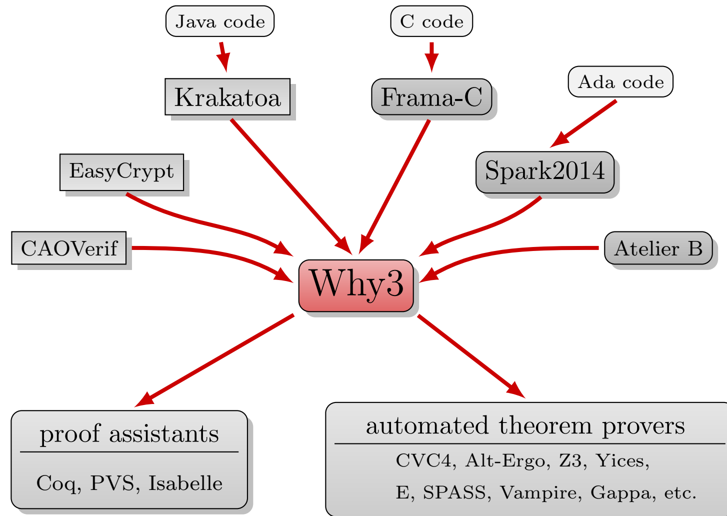

Figure 2.1: Why3 as an intermediate language.

ghost code cannot be used in a regular code computation, and it cannot modify a regular code mutable value [23]. Also this type of code cannot modify non-ghost data, just access it, but regular code cannot modify or access ghost data, only ghost code can modify ghost data. This language can also be used as an intermediate language for the verification of programs written in C, Ada or Java in a similar fashion to the Boogie language [21]. The programs are written in some language, then they are translated into a WhyML program and finally the verification conditions are extracted and sent to the provers. Frama-C [34], Spark2014 [29] and Krakatoa [20] are frameworks that use WhyML as an intermediate language for the verification of programs written in C, Ada and Java respectively as it is shown in Figure 2.1.

The framework focuses on the concept of task, a list of declarations followed by a goal. These tasks can be extracted from the files given to the front end component or can be obtained from a file given directly to the back end of the framework. These tasks then suffer some transformations in order for the specified theorem prover to comprehend them and for the tasks to fit the prover's logic. For example if we want to use Z3, then an issue arises since its logic is different from Why3's logic, hence the need for the transformations to be applied [8]. In order to ease the procedure of applying the transformations, there is a text file called *driver* that is associated to every prover supported by Why3 [8]. This *driver* file contains the transformations to be applied, the output format, the list of built-in symbols and corresponding axioms to ignore [19].

This framework has already been used to prove several realistic programs including VOCAL an OCaml library [9, 42], a certified first-order theorem prover [10], an interpreter for the language CoLiS [32], Strassen's algorithm for matrix multiplication [11] and an efficient arbitrary-precision Integer library [44].

2.3 Consistency Models

2.3.1 Operation Consistency Models

Global distributed data stores with high availability and scalable information require geo-replication in order to be more available in different parts of the globe. The CAP theorem [27] shows that in order to maximize availability and tolerate network failures, these systems must compromise their requirements for the consistency of its data. This theorem says that in a distributed system it is impossible to simultaneously have these three guarantees:

- **Strong consistency:** Every client of the data store observes the effects of the most recent write operation.
- **Availability:** All operations finish eventually.
- **Partition-tolerance:** The system is not vulnerable to network partitions. Network partitions are anomalies that fraction the system into multiple components. In the case of a network partitioning, if a replica from a component sends a message to another replica outside that same component, then that message will be lost.

The impossibility of having these three properties in a distributed system is proved by contradiction. Assuming there is a network consisting of at least two nodes, it is possible to divide it into two non-empty disjoint sets *S1*,*S2*. It is assumed that all the messages between partition *S1* and partition *S2* are lost. If a *write* operation occurs in a node in partition *S1* and then a *read* operation occurs in partition *S2*, then it will not be able to witness the effects of the *write* operation.

In the properties shown above, there is one called strong consistency. However, consistency is a spectrum where in one end there is strong consistency and at the other end there is eventual consistency. Strong consistency is the strictest model and eventual consistency is the less strict model when it comes to consistency guarantees. In strong consistency, the system behaves as if only one replica existed, ensuring a total order of operations in every replica. The data is passed to the replicas as soon as a *write* operation is issued to a replica. However, during the time the replicas are being updated, the system does not process another *write* operation until all the replicas are consistent. This model guarantees consistency of the data but on the other hand, the system suffers a decrease in its availability and an increase of the latency of its operations. On the other end of the spectrum there is eventual consistency. Under this model when a replica receives a *write* operation it executes it and then answers the client. After that, asynchronously, that replica sends the effects of the *write* operation to the remaining replicas. This means that it is possible for a new *write* operation to be processed without consensus with other replicas, hence the possible violation of data integrity. The positives of this model are the high availability of the system and the low latency of the operations. Since eventual consistency is less

strict then strong consistency, then this model belongs to a range of consistency models called weak consistency. Another well known example of a weak consistency model is causal consistency. The causal consistency model is based on Lamport's definition of the happens-before relation [35] which captures the notion of causality by relating an operation to previous operations on the same replica and to other operations performed on other replicas whose effects are propagated using messages. Normally causal consistency is seen as a combination of the following four properties:

- **Read your writes:** A client must always be able to observe the effects of all previous writes issued by itself.
- **Monotonic Reads:** Subsequent reads issued by the same client should observe either the same state or the same state that has been modified by write operations.
- **Monotonic Writes:** The effects of multiple write operations issued by some client must be observed in the same order that they occurred by the remaining clients.
- **Writes follow reads:** This guarantees that write operations come after read operations that have influenced them.

However, causal consistency by itself does not guarantee that the state of the system converges because divergence can last forever without breaking any of its properties. Having this in mind a consistency model has been proposed where geo-replicated systems could use a combination of several consistency models, which is used by many systems nowadays [2, 37, 50]. Specifically, the approach would be to use strong consistency whenever the correction of the application is at risk, and leverage weak consistency when the operations are safe. This way it is possible to take advantage of an asynchronous execution when operations are safe [38], thus increasing availability and at the same time when critical operations are executed, there is a guarantee of correctness of the application. The main challenge regarding this consistency model is achieving the correct balance between weak and strong consistency because the programmer needs to think about the concurrent effects of the operations, and decide which operations require synchronisation to assure the correction of the application. This is one of the main concerns approached by this work.

2.3.2 Transaction Consistency Models

A transaction is an operation that groups multiple read and write operations into a single coarse-grained operation that preserves the ACID properties:

- **Atomicity (All-or-Nothing):** All the operations have their intended effects or none of them have, and this is essential because it prevents updates to the data store from occurring only partially.

- **Correctness:** The state of the data store respects all the invariants before and after the execution of a transaction.
- **Isolation:** Each transaction executes with no interference from other concurrent transactions, which means that either all the effects from a transaction are visible or none of them is.
- **Durability:** The effects of a transaction that terminates successfully are not lost independently of failures that might occur.

Together these properties provide the serializability model, a program semantics in which a transaction executes as if it was the only one accessing the database. This model allows concurrent operations to access the data store and achieve expected results because it restricts the acceptable interactions between concurrent transactions [46].

There are several consistency models that can be used for transactions but the focus in this section goes to the following:

- **Read Committed (RC):** Transactions that use this level of isolation only witness effects of write operations from transactions that committed, thus ensuring atomicity of the transactions. This consistency model does not offer any other guarantees [6].
- **Monotonic Atomic View (MAV):** This consistency model strengthens the Read Committed model by preventing lost updates. MAV also ensures that when an operation from transaction T1 witnesses the effects of a transaction T2, then the subsequent operations from T1 will also witness the effects of T2. This model is helpful when it comes to enforcing the integrity of foreign key constraints, materialized views, indices and secondary updates. For this model to be used, the data store only needs to keep track of the set of transactions that have been witnessed by the running transaction, and before the execution of an operation it needs to be ensured that the replica includes all the transactions in the set [1].
- **Repeatable Read (RR):** The semantics of this consistency model requires that the transaction witnesses a snapshot of the state of the data store, and this snapshot can be obtained from any replica of the system [6].

2.4 CRDTs

As it was mentioned in Section 2.3 with replication comes the issue of the consistency of the stored data. Eventual consistency is an example of a model that can provide high availability but at the same time introduces a lack of consistency guarantees, because it does not guarantee that replicas will converge. Shapiro et. al [47] proposed a theoretically-sound approach to eventual consistency called strong eventual consistency. This system model proposes conditions to achieve convergence and avoid the complexity of ad-hoc

conflict resolution and roll-back [47]. The data types that satisfy the previous conditions are called Conflict-free Replicated Data Types (CRDTs). These data types guarantee convergence in a self-established way, despite failures. Also, these data types have a deterministic conflict resolution policy by construction. CRDTs do not require synchronization for an update to happen, and they remain responsive, available and scalable despite high latency, faults or disconnection from the network [47]. CRDTs are mutable objects that are replicated in a set of replicas that are interconnected by an asynchronous network, which can partition and recover. In this network, processes that do not crash are considered as correct, and in the case of a crash of a process, when it recovers it comes back with its memory intact.

A client of a CRDT object has two types of operations he can call in order to communicate with it: *read* and *update*. *Read* operations are called to read the current state of the object and the *update* operations serve to update the state of the object. Either one of these methods are only applied in one replica and then the effects are propagated to the remaining replicas asynchronously.

CRDTs can be divided into two categories: operation-based CRDTs and state-based CRDTs. With state-based CRDTs, also known as Convergent Replicated Data Types (CvRDT), every replica often sends its state to another replica in order to propagate the updates that occurred to its state. The replica that receives the state from a remote replica then merges it with its own state. This merge operation needs to be idempotent, commutative and associative. The new state then has all the updates that occurred in the replica that sent the state and the updates that occurred in the replica that received the state and merged it. This kind of CRDTs only requires eventual consistency for all the replicas to know the effects of an operation. Since this kind of CRDTs only requires eventual consistency, there is the issue of inconsistency of the data since the replicas only eventually receive the effects of a *write* operation executed in another replica. The advantages of state based CRDTs are the high availability of the system and the low latency of operations. The other kind of CRDTs are the Operation-based CRDTs, also know as Commutative Replicated Data Type (CmRDT). Contrarily to the state-based CRDTs, operation-based CRDTs do not provide the *merge* operation. With this kind of CRDTs, the *update* operation is split in two: a *prepare-update* method and an *effect-update* method. The *prepare-update* is executed at the replica where the operation was invoked, and the *effect-update* is executed afterwards. The *effect-update* is executed at all the remaining replicas that received this operation from the replica where the *prepare-update* was executed. An update is only considered as being delivered at a replica when it is included in its causal history [47]. Since this type of CRDTs requires causal consistency for updates, then there is also the issue of not guaranteeing the consistency of the data.

There are several data structures that can be chosen from for the implementation of a CRDT like a flag or a set for example. A set is an abstract data type that can only store unique values without any specific order. Considering a set that has operations *add* and *remove*, there is the possibility of originating sequences of operations that do

```
type remove_wins_set 'a = {
  mutable remove_wins_add: fset 'a;
  mutable remove_wins_removes: fset 'a;
}

let ghost predicate equal (s1 s2: remove_wins_set 'a) =
  s1.remove_wins_add     == s2.remove_wins_add &&
  s1.remove_wins_removes == s2.remove_wins_removes

val empty_set () : remove_wins_set 'a
  ensures { is_empty result.remove_wins_add }
  ensures { is_empty result.remove_wins_removes }

predicate in_set (elt: 'a) (s: remove_wins_set 'a) =
  mem elt s.remove_wins_add && not (mem elt s.remove_wins_removes)

val add_element (elt: 'a) (s: remove_wins_set 'a) : unit
  writes  { s.remove_wins_add }
  ensures { s.remove_wins_add = add elt (old s).remove_wins_add }

val remove_element (elt: 'a) (s: remove_wins_set 'a) : unit
  writes  { s.remove_wins_removes }
  ensures { s.remove_wins_removes = add elt (old s).remove_wins_removes }

```

Figure 2.2: Remove-wins Set CRDT in Why3.

not commute when executed concurrently. For example, if there is a *remove* operation and an *add* operation executed concurrently regarding the same object, more specifically if the *add(o)* operation occurs after the *remove(o)* then the object *o* remains in the Set, but if the operations are executed in the reverse order then object *o* would not be in the set. One way of solving this issue is making sure that a *remove(o)* only affects *add(o)* operations that are visible to *remove(o)*. To achieve this, every element *o* from the set has an associated unique tag *t* and then when an operation *remove(o)* occurs, all the tags from *o* that the local replica knows are removed [47]. This way the set becomes an add-wins set because the precedence is given to an *add* operation when executed concurrently with a *remove* operation. If the precedence is given to the *remove* operation, then it becomes a remove-wins set. An example of remove-wins set is presented in Figure 2.2.

2.5 Hoare's Logic

The concept of Hoare's logic was introduced by Hoare [30] and had the goal of providing a logical basis in order to prove the properties of a computer program using deductive reasoning. Deductive reasoning is the process of relating premises with conclusions such that if all premises are true, the terms are not ambiguous, and the rules of deductive logic are followed, then the conclusion is true.

One of the most important aspects in analysing a computer program, is checking

if given its properties and the consequences of its execution, the program reaches a state that respects the expected behaviour of said program. The expected behaviour of a program can be specified as a set of assertions regarding properties of a program that need to be preserved before and after its execution. These assertions are expressed using mathematical symbols and logic. Hoare starts by dividing the assertions in two categories: preconditions and postconditions. The preconditions specify the properties a program must respect prior to its execution. If the properties specified are not preserved prior to the execution of the program, then it is impossible to prove anything about its execution. The postconditions express the properties that the program needs to respect after its execution. With these two kinds of assertions, a notation is proposed in order to illustrate the connection between them and the program. In this notation, we have the program (Q), the preconditions (P) and the postconditions (R) resulting in $\{P\}\ Q\ \{R\}$, which can be translated to "If the assertions in P are true before the execution of Q, then the assertions in R will be true after its completion" [30]. In the case where no preconditions are provided, then the notation writes $\{\boldsymbol{true}\}\ Q\ \{R\}$, because there are no restrictions on the properties to be preserved by the program prior to its execution.

The axioms and rules of inference that were presented together with this notation in Hoare's paper, do not give a basis to guarantee that a program successfully terminates [30]. A possible cause of non-termination of a program is an infinite loop for example. Since this logic does not guarantee the successful termination of a program, it only guarantees *partial correctness*. Thus, the notation $\{P\}\ Q\ \{R\}$ should be interpreted as "provided that the program successfully terminates, the properties of its results are described in R" [30].

2.6 Design by Contract

Following the efforts of Dijkstra [17], Floyd [51], and Hoare [30] Meyer also addressed the use of assertions in an object-oriented language [7]. Meyer specified some guidelines regarding object-oriented construction in order to improve its reliability using assertions. The term reliability is often associated with a programming technique called *defensive programming*, which consists of including as many checks as possible, even if they are redundant, to protect every module of a program from possible anomalies. The problem of adding redundant checks to a software module is that it goes against the principle of improving its quality, since it causes an increase of its runtime complexity. This happens because when more checks are added more software is added, which implies more locations where things can go wrong so they also need to be verified, thus creating an "infinite loop" where we are adding new software and checks for the software that was just introduced.

Since defensive programming is not a very systematic approach to guaranteeing reliability, another solution is proposed more specifically, *contract theory*. This method presents the notion of a contract for software development, which has an inspiration from contracts that exist between people, where every party involved has obligations and

benefits. In a contract, a benefit for one of the involved parties is usually an obligation for another involved party. Specifying a contract in software can be achieved using assertions which define the relation between the *caller* of the routine and the called routine. The types of assertions that are used are the following:

- **Preconditions:** These assertions specify the requirements for a call of a routine, more specifically the obligations for the callers of the routine. If there are no preconditions on a routine then any state of computation is accepted before the execution of the routine. When there is a violation of a precondition then there is an indication that a bug exists in the caller's code or there is an error in its specification, because it failed with its obligations.

- **Postconditions:** These assertions specify the properties that are ensured after the execution of a routine, more specifically the obligations of the routine. If there are no postconditions any state of computation is accepted after the execution of the routine. When there is a violation of a postcondition then there is an indication of a bug in the routine's code since it failed with its obligations.

- **Class Invariants:** The invariant must be satisfied after the creation of every instance of a class and it must also be preserved by every routine that is available for the clients to call.

As previously mentioned, in defensive programming the focus is on using as many checks as possible even if they are redundant. Using preconditions, on the other hand, helps reducing the redundancy in tests. The stronger the precondition the bigger the burden is to the client calling the routine making this routine more demanding, but the stronger the postcondition is, the bigger the burden on the routine making the routine more tolerant.

2.7 Predicate Transformers

The concept of a predicate transformer was introduced by Dijkstra [16]. A predicate transformer consists of a mapping from one predicate to another predicate of a programming language. Regarding predicate transformers, there are two kinds: the weakest precondition and the strongest postcondition.

The terms "weakest" and "strongest" can be explained with a simple example: if A implies B but B does not imply A, then B is "weaker" than A and A is "stronger" than B. To better understand the previous explanation, here it is an example:

$\{x > 0\} \quad x := x + 1 \quad \{x > 0\}$

$\{x > -1\} \quad x := x + 1 \quad \{x > 0\}$

Above we have a function that increments the value of x which has two different and valid preconditions according to Hoare's logic. On the first example we have the precondition $\{x > 0\}$ and in the second example we have $\{x > -1\}$. The precondition

$\{x > -1\}$ is weaker than $\{x > 0\}$, because the latter implies the first. For this function and its precondition, $\{x > -1\}$ is the weakest precondition.

The weakest precondition predicate transformer (wp) tries to find the weakest precondition for a function S such that S must terminate and an allowed final state is reached. On the other hand, the strongest postcondition predicate transformer (sp) states that if a function S starts at a state that respects its preconditions, then if S terminates a state will be reached that respects the strongest postcondition [26]. The weakest precondition calculus guarantees the termination of the function S, whereas the strongest postcondition calculus does not.

Predicate transformers can be used for the automatic inference of preconditions, postconditions or loop invariants, thus helping reduce the overhead on the part of the programmer when it comes to writing them [25]. Automated program verifiers normally have a verification condition generator (VC gen) as a sub-component. These verification condition generators are considered as predicate transformers in the sense that they apply transformations to the VC's before they are presented to an automated theorem prover.

In the following subsections, we will explain the semantics involving both kinds of predicate transformers.

2.7.1 Weakest Precondition Calculus

For this kind of predicate transformer there are two variants: the weakest conservative precondition calculus and the weakest liberal precondition calculus. The first type is used to find the unique P, which is the weakest precondition for S and Q in the following Hoare's triple $\{P\}$ S $\{Q\}$ such that S terminates and produces a final state that respects Q. The second type is used to find the unique P, which is the weakest precondition for S and Q in the following Hoare's triple $\{P\}$ S $\{Q\}$ such that either S terminates and S produces a final state that respects Q or the execution of S does not terminate. For this section we will focus on the weakest conservative precondition calculus.

The weakest precondition calculus is related to Hoare's logic as we stated before, but there are other comparisons that can be made. Hoare's logic is relational[1] because for each Q there are many P such that $\{P\}$ S $\{Q\}$, whereas WP is functional because for each Q there is only one assertion WP(S,Q). WP respects Hoare's logic because {WP (S,Q)} S $\{Q\}$ is true. Additionally, as it was mentioned Hoare's logic only guarantees partial correctness, but WP guarantees its termination, thus guaranteeing total correctness.

In order to explain the weakest precondition calculus rules, we will consider a simple language, which is the representative of the actual intermediate language used in ESC/Java [45], whose BNF is presented in Figure 2.3.

Every execution of a statement from the language shown in Figure 2.3, either blocks, goes wrong (e.g. failure to terminate caused by an infinite loop) or terminates [36]. The

[1] Not to be confused with Relational Hoare's logic [5]

$$
\begin{array}{rcl}
S,T & ::= & Id := Expr \\
 & | & \textbf{assert}\ Expr \\
 & | & \textbf{assume}\ Expr \\
 & | & S\ ;\ T \\
 & | & S\ or\ T
\end{array}
$$

Figure 2.3: BNF of the ESC/Java language.

Stmt	wp (Stmt,Q)
x := E	Q [x := E]
assert E	E ∧ Q
assume E	E ⇒ Q
S ; T	wp(S, wp(T , Q))
S or T	wp(S,Q) ∧ wp(T,Q)

Table 2.1: Weakest Precondition Calculus Rules.

first statement is the assignment which sets the variable *Id* to the value of *Expr*. The following statements are the assert statement and the assume statement, which only execute if expression *Expr* attached evaluates to true. If the expression of the assert statement evaluates to false then the execution goes wrong and if the expression of the assume statement evaluates to false the execution blocks. The statement *S ; T* is the sequential composition of S and T, where T only executes if S terminates. The last statement is *S or T*, which corresponds to the arbitrary choice between S and T.

In order to execute the weakest precondition calculus we evaluate the provided function S and postconditions *Q*. Depending on which statement is found on the code, a specific transformation is applied in order to achieve the weakest precondition. For each possible statement of the language being analysed, we apply the associated transformation, that can be seen in Table 2.1 [17]. The assignment statement rule, Q [x := E] has the same meaning of **let** x = E **in** Q **end**. The remaining rules presented in Table 2.1 are self-explanatory.

2.7.2 Strongest Postcondition Calculus

The strongest postcondition predicate transformer (SP) is a function that receives as input a precondition *P* and a function S and returns the unique *Q*, which is the strongest postcondition of S with regard to its precondition *P* such that {*P*} S {SP (S,P)} must be satisfied. SP does not guarantee the termination of function S received as an input, thus only guaranteeing partial correctness like Hoare's logic. Similar to the weakest precondition predicate transformer, SP is also functional in the sense that for a given function S and its preconditions *P*, there is only one *Q* that is the strongest postcondition of S.

As it was done in the previous section, we consider the intermediate language used

Stmt	sp (Stmt,P)
x := E	$\exists x'.\ x = E(x \leftarrow x') \wedge P\ (x \leftarrow x')$
assert E	E ∧ P
assume E	E ∧ P
S ; T	sp(T, sp(S,P))
S or T	sp(S,P) ∨ sp(T,P)

Table 2.2: Strongest Precondition Calculus Rules.

in ESC/Java in order to present the semantics of the strongest postcondition calculus. The semantic rules for the strongest postcondition calculus are presented in Table 2.2. The assignment statement rule, states that a there is variable x' such that the logical conjunction of the assignment of E to x with Q is true, by replacing in the expression E and Q every occurrence of x with x'. The remaining rules presented in Table 2.2 are self-explanatory.

2.8 CISE proof rule

CISE [28] presents a proof rule and tool for proving the preservation of integrity invariants in applications built over replicated databases. For this proof rule a generic consistency model is proposed, and it allows the definition of specific consistency models for each operation of the application. This consistency model assumes causal consistency which means that the communication between replicas guarantees causality. This means that if a replica sends a message containing the effect of an operation o2 after it sends or receives a message containing the effect of an operation o1, then no replica will receive the message about o2 before it receives the one about o1. The causal propagation of messages can solve some issues. However, just the use of causal consistency does not ensure the preservation of some integrity invariants [28]. One problem that can occur is when two operations update the database, without being aware of the effects of each other.

CISE's proposed consistency model allows the strengthening of causal consistency, by letting the programmer specify which operations cannot be executed concurrently. This is done using a token system $\mathcal{T}$, containing a non-empty set of tokens *Tokens* and a conflict relation $\bowtie$ over tokens. The set *Tokens* presents the tokens that can be associated with the application's operations and the conflict relation $\bowtie$ states which tokens are conflicting. If two operations with conflicting tokens try to execute concurrently, then they need to be executed in a way that both operations know the effects of one another, making them causally dependent. Since operations with conflicting tokens are causally dependent, causal message propagation ensures that every replica will execute those operations in the same order. CISE's proof rule is *modular*, in the sense that it allows for the tool to reason about the behavior of each operation individually [28]. Also, the proof-rule is *state-based* because it reasons in terms of states obtained by evaluating the effects that

operations have on the state of the application.

Following the proof rule presented by CISE, the CISE tool was developed with the purpose of automating said proof rule. This tool is SMT-based, meaning it uses off-the-shelf SMT solvers to discharge generated verification conditions. The CISE tool has had two iterations [38, 41]. The first version of the tool was developed by Najafzadeh et al. [41], and used Z3's low-level APIs directly [40], making it difficult to use. The second version of the tool was developed by Marcelino et. al. [38], in which the specifications were written in Boogie [4]. Boogie generates a set of verification conditions from the programmer's specification, and then discharges them to an SMT solver. One advantage this tool has over the Z3 version is the fact that the specifications are written in a higher-level specification language. That means the programmer does not have to deal with Z3's low-level API. However, there is a downfall of using Boogie, which is its current lack of active maintenance. Both tools provide a counterexample when the verification process fails.

CISE's proof rule proposes three proof obligations: the safety analysis, the commutativity analysis, and the stability analysis. These three proof obligations can be resumed as follows:

- **Safety Analysis:** Verifies if the effects of an operation, when executed without any concurrency, preserve the invariants of the application.
- **Commutativity Analysis:** Verifies if every pair of different operations commute, *i.e.*, if the operations are executed in any order, then the same final state is reached starting from the same initial state.
- **Stability Analysis:** Verifies if the pre-conditions of an operation are stable under the effects of each operation of the system. If two operations are stable, they can be safely executed concurrently.

If there is a pair of operations where one of them can violate the other's precondition when executed concurrently, then a set of tokens is associated to these operations. The idea is to associate pairs of tokens with the specified conflict relation, thus stating that these operations cannot be executed concurrently. In the Z3 version of the tool the tokens are only associated to the operations, ignoring their parameters, only allowing for a coarse-grained approach [41]. For the Boogie version of the tool the token association mechanism became more fine-grained, testing different values for the parameters of each pair of conflicting operations. This was done to find the cases where the combination of parameters could break the invariants. The programmer associates tokens to arguments of the operations and states, via a conflict relation, which arguments have conflicting tokens. When operations that have arguments with conflicting tokens execute concurrently, then if the arguments have the same value they need to be synchronised in order to be safely executed [38].

Chapter 3

CISE3 Architecture

In this section we discuss how our tool CISE3 works. In Section 3.1 we present an overview of our tool. In Section 3.2 we present how our tool performs the three proof obligations from the CISE proof rule, presented in Section 2.8. In Section 3.3 we discuss the specification of token systems, using our provided DSL. In Section 3.4 we present our CRDTs library for the resolution of commutativity issues. In Section 3.5 we present the implementation of the strongest postcondition calculus from our tool.

3.1 CISE3 Overview

In this thesis we are proposing a static analysis tool, to automatically analyse applications operating over weakly consistent databases with the aid of the deductive verification framework Why3 [21]. This approach follows the proof rule proposed by CISE [28]. Since our approach is based on CISE's proof rule we assume causal consistency as mentioned in Section 2.8. This assumption can be relaxed to eventual consistency, as we discuss later in Section 6.2.

As we mention in Section 2.2, the Why3 framework provides a programming and high level specification language. From programs written and annotated using WhyML, are generated verification conditions, that can be sent to several of the more than 25 supported theorem provers. This diversity allows to rectify some limitations from other proof tools that just support one theorem prover, usually an SMT solver. Simultaneously, the Why3 framework also provides a graphic environment for the development of programs, where we can interact with several theorem provers, perform small interactive proof steps [14], as well as, visualising counter-examples.

The Why3 framework can be extended via plug-ins like in the cases of Jessie [39] and Krakatoa [20]. The integration of new plug-ins into Why3 is relatively simple: we write

a parser for our target language, whose intermediate representation should me mapped to an non-typed AST of the WhyML language. Finally, we use the typing mechanism from Why3 to generate a typed version of the previous AST. The WhyML program that is generated after this typing phase, can now be analysed by the Why3 framework. The CISE3 tool is a plug-in for the Why3 framework whose target language is WhyML which is why we also leveraged the already existing parser for the WhyML language.

The CISE3 architecture presented in Figure 3.1 has four components:

- **Proof Obligations Component:** This component is related with the execution of the three proof obligations from CISE's proof rule. After the execution of this component the programmer knows the pairs of non-commutative and conflicting operations.
- **Token System Component:** This component is related with the specification and validation of a token system defined by the programmer. For this component we provide a DSL that programmers can use in order to specify token systems. When the programmer provides a token system to our tool then the "Proof Obligations Component" suffers some modifications as discussed in Section 3.3.
- **CRDTs library Component:** This component is related with the use of CRDTs for solving commutativity issues found in applications. For this part of our tool, we provide a library of CRDTs implemented and verified using the Why3 framework. After the execution of the "Proof Obligations Component" the programmer is aware of the pairs of non-commutative operations and can use a CRDT from this library to solve this issue as seen in Figure 3.1.
- **SP Component:** This component is related with the automatic generation of strongest postconditions for operations of the applications to be analysed by our tool. For this component we provide a strongest postconditions predicate transformer over a DSL of ours, similar to a subset of the WhyML language.

We believe that our choice of the Why3 framework, and basing our tool on its architecture of plug-ins, has allowed for a reduction of the development and validation effort for CISE3. Also, developing our tool over a mature framework allows us to evolve for the analysis of more realistic examples.

3.2 Proof Obligations Component

As we mention in Section 2.8, the safety analysis from the CISE proof rule consists of verifying if an operation when executed without any concurrency does not break an integrity invariant of the application. As an input, the programmer provides to the tool the specification of the application's state and its invariants, as well as the sequential implementation and specification of each operation. As an example of a generic Why3

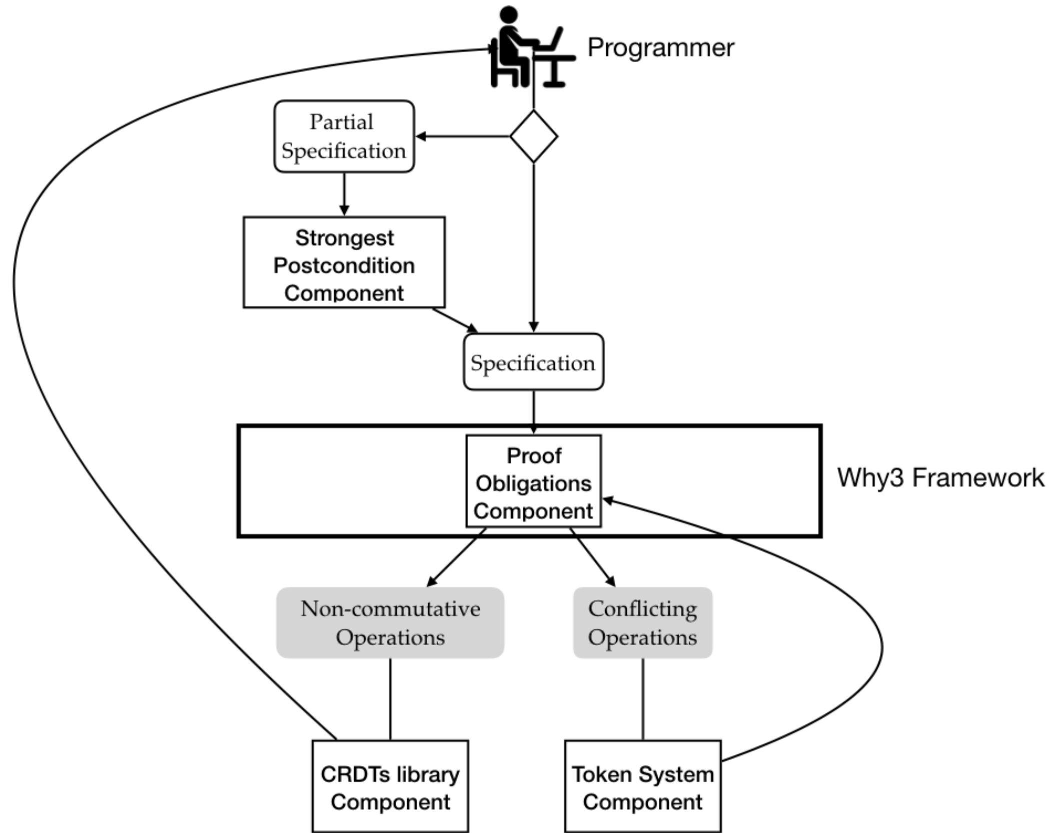

Figure 3.1: CISE3 architecture.

```
1  type τ [@state] = { x:τx }
2  invariant { 𝓘 }
3
4  let f (x:τ1) (state: τ)
5    requires { 𝒫1 }
6    ensures  { ℚ1 }
7  = e1
8
9  let g (y:τ2) (state: τ)
10   requires { 𝒫2 }
11   ensures  { ℚ2 }
12 = e2
13
```

Figure 3.2: Generic program in Why3.

program we consider the program presented in Figure 3.2. This program is composed by operations `f` and `g`, and the type of the state of the application τ, with the invariant $\mathcal{I}$ associated to it.

When the programmer is specifying the type of the state of the application, he needs to associate to it the tag `[@state]`, so it is possible to identify this type as the representative of the state, as it is seen in Figure 3.2. The type τ possesses a set of fields represented

```
1  let ghost f_g_commutativity () : (τ, τ)
2    returns { (s₁, s₂) → eq s₁ s₂ }
3  = val x₁:τ₁ in
4    val state₁ : τ in
5    val x₂:τ₂ in
6    val state₂ : τ in
7    assume { 𝒫₁ (x₁,state₁) ∧ 𝒫₂ (x₂,state₂) ∧
8     eq state₁ state₂ }
9    f x₁ state₁;
10   g x₂ state₁;
11   g x₂ state₂;
12   f x₁ state₂;
13   (state₁, state₂)
14
```

Figure 3.3: Generated commutativity and stability analysis function for operations f and g.

by $\overline{x}$, whose types are represented as $\overline{\tau_x}$. Additionally, in the code shown in Figure 3.2 there is the definition of operations f and g. Every operation must have an instance of the state of the application passed as an argument so it can be used later in the generation of commutativity and stability analysis functions. In the code from Figure 3.2, the preconditions of operation f are represented as $\mathscr{P}_1$, the postconditions are represented as $\mathbb{Q}_1$ and its body is represented as e_1. The specification of the operation g is similar to the one of f, but in this case the preconditions, postconditions and body are represented by $\mathscr{P}_2$, $\mathbb{Q}_2$ and e_2 respectively.

The Why3 framework by itself already has the ability of verifying for each operation, if its implementation adheres to the specification. Also, the programmer by specifying the state of the application with its integrity invariants, can use Why3 to verify if an operation breaks them or not, given its implementation. In the cases where a verification condition from the program is not discharged, the framework can present a counter-example [13]. That said, it is possible to state that Why3 is capable of performing the safety analysis without changing anything within the framework.

For the remaining two proof obligations of the CISE proof rule, our tool automatically generates functions that verify the commutativity and stability between pairs of operations. Given the code from the application, CISE3 uses Why3's parser to obtain an in memory representation of the contents of the WhyML program. After the program is parsed, for each pair of different operations a commutativity analysis function is generated automatically, which also verifies the stability between those operations.

For the generic application presented in Figure 3.2, our tool only generates one function for the commutativity and stability analysis between operations f and g. The generated function is presented in Figure 3.3. If Why3 is able to prove that the function presented in Figure 3.3 respects the specification, then operations f and g commute and do not conflict. This function starts by generating two equal states of the application, as

well as the arguments for each operation in lines 3 to 6. The construction `val..in` is used in WhyML to generate a constant of a certain type. For instance, the expression in line 4 generates a value of type τ, binding such value to the variable $\texttt{state}_1$. An expression of the form (`val x :` γ `in`) is syntactic sugar for (`let x = any` γ `in`). The next step is the `assume` expression in line 7, which has the purpose of restricting the values that the generated arguments can possess in order to preserve the preconditions of the operations being analysed. The `assume` expression also states that both generated states must be the same, because f and g are executed in alternative orders but both orders of execution must start from the same initial state. Then operation f is executed, followed by the execution of operation g over $\texttt{state}_1$, in lines 8 and 9. This execution of f and g is followed by the alternative order of execution but over $\texttt{state}_2$ in lines 10 and 11. After these alternated executions of operations f and g, if the final states are the same then we assume that f and g commute. To verify if two states are the same, the postcondition uses a state equality relation (eq). This equality relation is a field by field comparison of the record type of the state of the application. The equality relation between states can be automatically generated by our tool, if the programmer does not provide one. In Chapter 4 we present several case studies that require an equality relation provided by the programmer.

For the stability analysis in the function of Figure 3.3, we try to prove the preservation of the precondition of an operation after the execution of another operation, (*e.g.*, the precondition of g in the state obtained after the execution of f). So, in the first order of execution over state1 we check if the preconditions of operation g are preserved and in the second order of execution over state2, our tool tries to prove the preservation of the preconditions of operation f. If we are not able to prove one of these two assertions, then we assume that f and g are conflicting and that they cannot be safely executed concurrently. Otherwise, f and g are not conflicting. This way, we can perform the commutativity and stability analysis for each pair of different operations in a single step.

One important remark regarding the generated functions by our tool, is the name of the arguments that are generated inside them. The generated arguments do not have the same names as in the operations signature, because our tool changes them due to possible conflicts that can occur between argument names of different operations. In order to solve this issue, our tool adds a suffix after the name of each argument, thus avoiding any name clashes. Also, in the `assume` expression our tool changes the names of the variables in the preconditions of the analysed operations, so there is a match with the new generated names.

For the remaining stability analysis, our tool generates a function for each operation. Each of these remaining functions have the purpose of checking if there can be multiple concurrent executions of the same operation. Given the generic program presented in Figure 3.2, to check the stability of operations f and g, our tool generates the functions presented in Figure 3.4.

The functions presented in Figure 3.4 start by generating the initial state and the arguments for the operation calls in lines 2 to 5 and in lines 11 to 14. Similar to the

```
1   let ghost f_stability () : ()
2   = val x̄₁:τ̄₁ in
3     val state₁ : τ in
4     val x̄₂:τ̄₁ in
5     assume { 𝒫₁ (x̄₁,state₁) ∧
6              𝒫₁ (x̄₂,state₁)}
7     f x̄₁ state₁;
8     f x̄₂ state₁;
9
10  let ghost g_stability () : ()
11  = val x̄₁:τ̄₂ in
12    val state₁ : τ in
13    val x̄₂:τ̄₂ in
14    assume { 𝒫₂ (x̄₁,state₁) ∧
15             𝒫₂ (x̄₂,state₁)}
16    g x̄₁ state₁;
17    g x̄₂ state₁;
18
```

Figure 3.4: Generated stability analysis function for operations f and g.

commutativity analysis functions, the assume expression in line 6, for example, is also used to restrict the values that the generated arguments and states can take so the preconditions of the operation being analysed can be preserved. Lastly, the operation is executed consecutively in lines 7 to 8 and in lines 16 to 17. To check if an operation is not conflicting with itself, we use our tool to try to prove this stability analysis function. In case we are not able to prove the generated function, due to a violation of a precondition on the second call to the operation, then we assume that there cannot be multiple concurrent executions of that operation. As it was pointed out for the commutativity analysis function, in the stability analysis functions the name of the generated arguments and the assume expression suffer modifications, to avoid name clashes between arguments from different operations.

3.3 Token System Component

After the analyses from our tool are performed, the programmer is aware of the pairs of conflicting operations of the application. With this information the programmer can provide a token system to our tool and check if the consistency model that it represents is sound. This is done by executing CISE3 over the application's specification once again. Given the token system, if we are able to discharge every generated verification condition, then the associated consistency model is considered sound. The programmer can execute our tool several times iteratively refining the token system each time, in order to find the less strict consistency model that guarantees data integrity. The idea of defining a token system to represent the consistency models of an application comes from CISE, as explained in Section 2.8. In Chapter 4 we show some examples of the definition of token

$$
\begin{array}{rcl}
\textit{tokenSystem} & ::= & \textit{tokenDef} \\
 & | & \textit{conflictsDef} \\
\textit{tokensDef} & ::= & \textit{token tokensDef} \\
 & | & \textit{token} \\
\textit{token} & ::= & \textbf{token}\ \textit{opId tokenId}^{+} \\
 & | & \textbf{argtoken}\ \textit{opId argId tokenId} \\
\textit{conflictsDef} & ::= & \textit{conflict conflictsDef} \\
 & | & \textit{conflict} \\
\textit{conflict} & ::= & \textit{tokenId}\ \textbf{conflicts}\ \textit{tokenId}
\end{array}
$$

Figure 3.5: BNF that represents our provided token system specification language.

systems for our case studies, as well as its repercussions in the execution of our tool.

In Figure 3.5 we present the grammar of our DSL for specifying token systems. The first two productions from the grammar are straight forward so firstly we focus on the *token* production. The first rule from the *token* production, describes the declaration of a non-empty list of tokens, associated with an operation. Each token can only be declared once, as it cannot be associated to more than one operation. The second rule from the *token* production describes the association of tokens to arguments of an operation. The last production, illustrates how the programmer can declare two tokens as being in conflict. The tokens that are used in the *conflict* production must both have been defined previously. When two tokens declared with the keyword **token** are conflicting, the operations associated to those tokens cannot be executed concurrently in any situation. In this case, our tool will not generate any analysis function regarding the analysis for the pair of conflicting operations. If two tokens that were declared with the **argtoken** keyword are conflicting, the operations associated to the tokens can only execute concurrently if the value of the arguments are different. In this case, our tool assumes that two operations with conflicting arguments only execute concurrently when the values of the arguments are different.

To illustrate our token system specification language let us consider the following example: our tool is executed over the application from Figure 3.2 and finds out operation `f` is conflicting with itself. Given this information, the programmer can provide the following token system:

```
1  token f tau
2  tau conflicts tau
```

The token system presented above shows a consistency model where operation `f` should not be safely executed in concurrence with itself in any situation. So, given this token system our tool does not generate the stability analysis `f_stability`. However, let us consider that the conflict regarding operation `f` is only related to one of its parameters `arg`. In this case the programmer can provide a more refined token system like the one below:

```
argtoken f arg tau
tau conflicts tau
```

Now, this token system depicts a consistency model where f can only be safely executed concurrently when the value of arg is different in each concurrent execution of f. Given this new token system, our tool changes the assume expression from the f_stability function in Figure 3.4. The new assume expression is presented below:

```
assume { 𝒫₁ (x̄₁,state₁) ∧ 𝒫₁ (x̄₂,state₁) ∧ (arg1 ≠ arg2)}
```

3.4 CRDTs library Component

A mostly frequent issue that can occur in geo-replicated systems is data divergence due to the execution of non-commutative operations, in different orders in different replicas. A possible solution to this issue is the introduction of CRDTs in the application code. As mentioned in Section 2.4, CRDTs guarantee the convergence of data in a self-established way. With this in mind, we present a library of CRDTs implemented and verified using the Why3 framework. The goal of this library is to provide programmers with a verified solution in order to solve commutativity issues that are uncovered by our tool. The programmer can also provide its own CRDT specifications in WhyML and use them to solve commutativity issues. In Chapter 4 we show a case study where a CRDT is used to solve a commutativity issue discovered by our tool. Also, all the CRDTs implemented for our library are presented in Appendix A.

3.5 Strongest Postcondition Component

As previously mentioned the goal of our approach is to help the programmer reason about the correctness of weakly consistent distributed applications. Even if the programmer only has to provide the specification of the application, finding the best specification for an application can be a tough task. So, in order to help the programmer with this chore we provide a strongest postcondition generator. This predicate transformer can be used by the programmer in case she needs help reasoning about the postconditions of each operation, as stated in Section 2.7.

For this component we provide a DSL to write specifications targeting our strongest postcondition calculus. This DSL is similar to a subset of the WhyML language, making it easier for the programmer to integrate the generated postcondition in the application. The input the programmer needs to provide to our predicate transformer consists of the implementation and preconditions of the operation. Our strongest postcondition generator follows the rules presented in Table 2.2. Another important remark about this DSL is that it does not support any form of loop constructors. In Chapter 4 we present some case studies where we applied our strongest postcondition calculus.

Let us discuss some design choices regarding our predicate transformer and its target language. First of all, the case studies evaluated by our tool do not have global variables. This departs away from a more traditional presentation of a strongest postcondition calculus [17]. Every variable used in an operation is passed as a parameter, which can raise the problem of aliasing. In order to explain how we cope with this issue let us first consider the following signature for operation f:

```
1  val f (x:τ1) (state: τ) : τres
```

According to the type system from Why3 [24], all arguments from an operation are separated from each other, using here some Separation Logic vocabulary [43]. By that we mean that there are no two arguments pointing to the same memory location. Since our predicate transformer only has in consideration the arguments of the operation, then the problem of aliasing is mitigated.

Another implementation choice regarding our predicate transformer is related to the integrity invariants over the state. Our predicate transformer does not include the integrity invariants in the generated postconditions. We make this choice because when the programmer writes the specification of an application in Why3, she does not need to explicitly include the state invariants. Why3 automatically adds any type invariants as a precondition and postcondition of operations. So, since Why3 inserts the integrity invariants by itself, our predicate transformer does not need to do it.

Chapter 4

Experimental Evaluation

In this chapter we evaluate our tool over a set of three case studies: a banking application, an auction application, and a school registration application.

4.1 Banking Application

In this section we present a complete case study of a simple banking application. This case study is similar to the example presented in Section 1.2. In this application we have a set of bank accounts, over which a client can only deposit or withdraw money. For this case we abstract the security aspects regarding which client can access which account, with every client being able to access every account. The sequential implementation and specification for this case study are presented in Figure 4.1.

The specification of operations `deposit` and `withdraw` are standard: it is not possible to deposit or withdraw negative values and also a client cannot withdraw more money than what is available in the account. The specified state of the application corresponds to a record with just one field named `balance`, which represents the accounts with each index of the array representing a different account. There is an integrity invariant associated to the state of the application. This invariant states that at any point during the execution of the application, the balance of each account must be non-negative. The proof effort for this sequential program is presented in Figure 4.2. The green cells represent the verification conditions that were discharged automatically by the specified external prover. That being said this table shows that the safety analysis is successfully performed.

For this example the programmer needs to introduce the predicate `state_equality` that represents the equality relation between states, as we can observe in Figure 4.1. In order for our tool to identify `state_equality` as the equality relation between states, the programmer must associate the tag `[@state_eq]`. If the programmer does not specify

```
1   type state [@state] = {
2     balance : array int
3   } invariant{ forall i. 0 ≤ i < length balance → balance[i] ≥ 0 }
4
5   let deposit(accountId amount: int) (state : state): unit
6     requires { amount > 0 }
7     requires { accountId ≥ 0 ∧ accountId < length state.balance }
8     ensures  { state.balance[accountId] =
9               old(state.balance)[accountId] + amount }
10    ensures  { forall i. i ≠ accountId →
11              state.balance[i] = (old state.balance)[i] }
12  = state.balance[accountId] ← state.balance[accountId] + amount
13
14  let withdraw(accountId amount: int) (state : state) : unit
15    requires { amount > 0 }
16    requires { state.balance[accountId] - amount ≥ 0 }
17    requires { accountId ≥ 0 ∧ accountId < length state.balance }
18    ensures  { state.balance[accountId] =
19              old(state.balance)[accountId] - amount }
20    ensures  { forall i. i ≠ accountId →
21              state.balance[i] = (old state.balance)[i] }
22  = state.balance[accountId] ← state.balance[accountId] - amount
23
24  predicate state_equality [@state_eq] (s1 s2 : state)
25  = array_eq s1.balance s2.balance
26
```

Figure 4.1: Specification and implementation of the banking application.

Proof obligations		Alt-Ergo 2.3.0	CVC4 1.5	Z3 4.5.0
lemma `VC for state`		0.00	0.03	0.02
lemma `VC for deposit`		0.01	0.05	0.01
lemma `VC for withdraw`		0.01	0.05	0.01

Figure 4.2: Statistics regarding the proof of the banking application.

the predicate `state_equality`, our tool automatically generates an equality relation, as specified in Section 3.2. For the specific case of comparing arrays, a simple structural comparison would not be sufficient to prove if they are equal. Therefore, in this case there is the need for a point-wise comparison between the elements of the arrays, which is accomplished by using the function `array_eq` from Why3 standard library.

When the programmer is writing the specification, she can use our strongest postcondition predicate transformer to automatically obtain the postconditions for each operation. To execute our predicate transformer for operation `deposit`, the input the programmer needs to provide to our tool is the one below, followed by the generated postcondition:

```
1  deposit (amount, accountId : int) (state : state) : unit
2    requires {amount > 0}
3    requires {accountId ≤ (length (state.balance))}
4    requires {accountId > 0}
5  = state.balance[accountId] ← (state.balance[accountId] + amount)
```

```
1  exists v0. state.balance[accountId] = v0 + amount &&
2    amount > 0 &&
3    accountId ≤ length (state.balance) &&
4    accountId > 0
```

The generated postcondition states that there exists a value `v0` representing the previous balance of the account which is accessed by the index `accountId`. Then, a value `amount` is deposited to the balance of the account `v0`. The remaining information refers to the propagation of the preconditions of operation `deposit`. Now the programmer can repeat the process for operation `withdraw` by providing the specification for the operation. The specification for `withdraw`, followed by the generated postcondition, is presented below:

```
1  withdraw (amount, accountId : int) (state : state) : unit
2    requires {amount > 0}
3    requires {accountId ≤ (length (state.balance))}
4    requires {accountId > 0}
5    requires {(balance[accountId] - amount) ≥ 0 }
6  = state.balance[accountId] ← (state.balance[accountId] + amount)
```

```
1  exists v0 . state.balance[accountId] = v0 + amount &&
2    amount > 0 &&
3    accountId ≤ length (state.balance) &&
4    accountId > 0 &&
5    v0 - amount ≥ 0
```

The generated postcondition states that there exists a value `v0` representing the previous balance of the account being accessed by the index `accountId`. Then, a value `amount` is removed from the balance of the account `v0`. The remaining information from the generated postcondition refers to the propagation of the preconditions from the `withdraw` operation.

```
1   let ghost deposit_withdraw_commutativity () : (state, state)
2     ensures  { match result with
3              | x1, x2 → state_equality x1 x2
4              end }
5   =  val ghost accountId1 : int in
6      val ghost amount1 : int in
7      val ghost state1 : state in
8      val ghost accountId2 : int in
9      val ghost amount2 : int in
10     val ghost state2 : state in
11     assume { ((amount1 > 0 ∧ accountId1 ≥ 0 ∧
12      accountId1 < length (balances state1)) ∧
13      amount2 > 0 ∧
14      ((balances state2)[accountId2] - amount2) ≥ 0 ∧
15      accountId2 ≥ 0 ∧
16      accountId2 < length (balances state2)) ∧
17      state_equality state1 state2 };
18     withdraw accountId2 amount2 state1;
19     deposit accountId1 amount1 state1;
20     deposit accountId1 amount1 state2;
21     withdraw accountId2 amount2 state2;
22     (state1, state2)
23
```

Figure 4.3: Generated commutativity analysis function for the bank application.

Comparing the postconditions of each operation from Figure 4.1 and the generated postconditions, we verify that the latter are more verbose. However, they are automatically generated. This helps the programmer to understand which postcondition must be supplied. In fact, an appropriate witness for the existentially quantified variable `v0` is `(old state).balance`, with which we recover the postconditions supplied in Figure 4.1.

In the implementation presented in Figure 4.1 we can observe that the application only has two operations: `deposit` and `withdraw`. Since there is only one pair of different operations, our tool only generates a single function for the commutativity analysis. The generated function is presented in Figure 4.3.

In the function of Figure 4.3 we start by generating the arguments `amount1`, `accountId1`, `state1`, `amount2`, `accountId2`, and `state2` which are used for the calls to operations `deposit` and `withdraw`. The expression `assume` has the purpose of restricting the space of possible combinations of values for the generated arguments, so that the preconditions of the analysed operations are preserved and both generated states are equal. After that, operation `withdraw` is executed over `state1` followed by the execution of operation `deposit` over `state1`. If we are not able to prove that the preconditions of the call to `deposit` are preserved, then we assume that `deposit` and `withdraw` are conflicting and cannot be safely executed concurrently. Afterwards, operation `deposit` is executed over `state2`, followed by the execution of operation `withdraw` over `state2`. As it was done with the previous order of execution of operations, for this sequence if we are not able to prove

```
let ghost withdraw_stability () : unit
= let ghost accountId1 = any int in
  let ghost amount1 = any int in
  let ghost state1 = any state in
  let ghost accountId2 = any int in
  let ghost amount2 = any int in
  assume { (amount1 > 0 ∧
    ((balances state1)[accountId1] - amount1) ≥ 0 ∧
    accountId1 ≥ 0 ∧ accountId1 < length (balances state1)) ∧
    amount2 > 0 ∧
    ((balances state1)[accountId2] - amount2) ≥ 0 ∧
    accountId2 ≥ 0 ∧ accountId2 < length (balances state1) };
  withdraw accountId1 amount1 state1;
  withdraw accountId2 amount2 state1

let ghost deposit_stability () : unit
= let ghost accountId1 = any int in
  let ghost amount1 = any int in
  let ghost state1 = any state in
  let ghost accountId2 = any int in
  let ghost amount2 = any int in
  assume { (amount1 > 0 ∧
    accountId1 ≥ 0 ∧ accountId1 < length (balances state1)) ∧
    amount2 > 0 ∧
    accountId2 ≥ 0 ∧ accountId2 < length (balances state1) };
  deposit accountId1 amount1 state1;
  deposit accountId2 amount2 state1

```

Figure 4.4: Generated stability analysis functions for the bank application.

the preservation of the preconditions of `withdraw`, then we assume `deposit` and `withdraw` are conflicting and cannot be safely executed concurrently. Lastly, a pair containing both final states `state1` and `state2` is returned, and if they are equal we know that `deposit` and `withdraw` commute. In this case, we are able to prove every verification condition generated by this function, meaning that operations `deposit` and `withdraw` commute and do not conflict. That said, these operations can be safely executed concurrently because they do not put data integrity at risk. Since there are no commutativity issues regarding these operations, there is no need to introduce a CRDT in this application. So, with function `deposit_withdraw_commutativity` we are able to execute the stability and commutativity analysis between `deposit` and `withdraw` in one step.

Now, we proceed to the remaining stability analysis where we verify for each operation if there can be multiple concurrent executions of said operation. For this analysis, our tool generates a stability analysis function for each operation of the application. In this case study, our tool generates the stability analysis functions presented in Figure 4.4.

Initially, every stability analysis function generates the arguments that will be used in the calls to the operation, as we saw in the commutativity analysis function. Also, similar to the commutativity analysis function, the `assume` expression restricts the search space

Proof obligations		Alt-Ergo 2.3.0	CVC4 1.5	Z3 4.5.0
VC for deposit_withdraw_commutativity		0.07	(5s)	FAILURE
VC for withdraw_stability	precondition 1	0.00		
	precondition 2	0.00		
	precondition 3	0.01		
	precondition 4	0.00		
	precondition 5	(1s)	(1s)	FAILURE
	precondition 6	0.00		
VC for deposit_stability		0.00	0.02	0.01

Figure 4.5: Statistics of the proof effort for the generated functions for the bank application.

of values that the arguments can take so that the preconditions of the first operation call are preserved. Additionally, the `assume` expression is used to ensure that both generated states are equal. After that, the operation being analysed is called consecutively, and we are not able to prove the preservation of the preconditions of the operation prior to the second call, then that operation is conflicting with itself. In this case study, every verification condition for function `deposit_stability` is proved automatically, ensuring that `deposit` is not conflicting with itself. However, for the stability analysis function `withdraw_stability`, we are not able to prove the preservation of the preconditions for the second operation call. This means that `withdraw` is conflicting with itself, therefore, there cannot be multiple concurrent executions of this operation. The proof effort for the commutativity and stability analysis functions can be seen in Figure 4.5. In this table we can see that precondition 5 cannot be proved by any external prover available. It is this precondition that makes us assume that `withdraw` is conflicting with itself.

As we observed in Section 2.8, a token system can be used to specify a specific consistency model over an application. Since the analysis phase of our tool is complete, the programmer already knows which operations are conflicting, and in this case there is only one conflict: `withdraw` is conflicting with itself. The programmer must supply a token system, which can then be analysed by our tool. A sound token system for this application can be seen below:

```
1  token withdraw  tau
2  tau conflicts tau
```

This token system defines a token `tau` which is associated to operation `withdraw` and it conflicts with itself. That said, our tool does not generate the stability analysis function for `withdraw`. Consequently, our tool only generates the functions `deposit_withdraw_commutativity` and `deposit_stability`, which we have seen in this section to have been automatically proved. So, every generated verification condition is proved automatically, thus the token

```
1       assume { (amount1 > 0 ∧
2         ((balances state1)[accountId1] - amount1) ≥ 0 ∧
3         accountId1 ≥ 0 ∧ accountId1 < length (balances state1)) ∧
4         amount2 > 0 ∧
5         ((balances state1)[accountId2] - amount2) ≥ 0 ∧
6         accountId2 ≥ 0 ∧ accountId2 < length (balances state1) ∧
7         accountId1 ≠ accountId2 };
8
```

Figure 4.6: Assume expression of `withdraw_stability` given the token system.

system and its underlying consistency model are considered sound.

Realistically, if the application from this case study is executed under the consistency model presented above, when a user is trying to withdraw money from an account, no other client can be withdrawing money from any account. So, this consistency model is too strict for the application and heavily decreases its availability. However, if we analyse the conflict of the `withdraw` operation, then we understand it is related specifically to the argument `accountId` instead of the operation as a whole. That said, the programmer can now provide another (more refined) token system to our tool to check its soundness. This new token system is presented below:

```
1   argtoken withdraw accountId tau
2   tau conflicts tau
```

This new token system states that the argument `accountId` of the operation `withdraw`, that has the token `tau` associated, is conflicting with itself. That said, our tool modifies the `assume` expression from operation `withdraw_stability`, adding the restriction that the argument `accountId` must have different values in each concurrent execution of operation `withdraw`. The modified `assume` expression is presented in Figure 4.6. With this modification in the `assume` expression, every generated verification condition is discharged, meaning that the token system presented above, and its underlying consistency model are sound.

4.2 Auction Application

In this section we present another complete case study, this time of a simple auction application. This application consists of a collection of bids, the winning bid of the auction and a flag that indicates if the auction is still open. For this application, we also abstract from security aspects such as which user of the application can close the auction, with every user being able to do so. The implementation and specification for this application is presented in Figure 4.7.

The auction application has two operations: `place_bid` to place a bid, and `close_auction` to close the auction. The specification for `place_bid` states that the bid that is going to be placed must be non-negative, and the auction must be open. After the execution of

```
1  type state [@state] = {
2    mutable bid : array int;
3    mutable winning_bid : int;
4    mutable open : bool
5  }
6  invariant{ if not(open) then
7                length bid ≥ 0 ∧
8                (forall i. ≥ 0 ∧
9                   i < length bid → winning_bid ≥ bid[i])
10         else winning_bid ≤ 0 }
11 invariant{ if length bid > 0 then
12                 (forall i. i ≥ 0 ∧
13                 i < length bid → bid[i] > 0)
14         else length bid = 0 }
15
16 let place_bid (b : int) (state : state) : unit
17   requires { b > 0 }
18   requires { state.open }
19   requires { state.winning_bid ≤ 0 }
20   ensures  { length state.bid = length (old state.bid) + 1 }
21 = state.bid ← append state.bid (make 1 b)
22
23 let close_auction (state : state) : unit
24   requires { length state.bid ≥ 0 }
25   requires { state.open }
26   requires { state.winning_bid ≤ 0 }
27   ensures  { forall i. i ≥ 0 ∧
28     i < length state.bid → (state.winning_bid ≥ state.bid[i])}
29   ensures{not(state.open) }
30 = for i = 0 to length state.bid - 1 do
31     invariant{ forall j. j ≥ 0 ∧
32      j < i → state.winning_bid ≥ state.bid[j] }
33     if state.bid[i] > state.winning_bid then
34       state.winning_bid ← state.bid[i]
35     done;
36     state.open ← false
37
38 predicate state_equality [@state_eq] (s1 s2 : state)
39 = val predicate (==) (b1 b2: bool)
40     ensures { result ↔ b1 = b2 } in
41   (array_eq s1.bid s2.bid) &&
42   (s1.winning_bid = s2.winning_bid) &&
43   (s1.open == s2.open)
44
```

Figure 4.7: Specification and implementation of the auction application.

Proof obligations		Alt-Ergo 2.3.0	CVC4 1.5	Z3 4.5.0
lemma `VC for state`		0.00	0.03	0.01
lemma `VC for place_bid`		0.03	0.04	0.01
lemma `VC for close_auction`		0.02	0.05	0.02

Figure 4.8: Statistics regarding the proof of the auction application.

operation `place_bid` the new bid is placed in the collection of bids. The specification for `close_bid` states that the auction must be open, and there must be at least one bid. After the execution of operation `close_auction`, the auction is closed and the winning bid of the auction is decided. We associate two integrity invariants to the state of the application. The first invariant states that at any point during the execution, if the auction is open then the winning bid is negative, but when the auction is closed, then it must be greater or equal then any bid from the collection of bids. The second invariant states that at any point during the execution every bid from the collection of bids, is non-negative. The proof effort for this sequential program is presented in Figure 4.8.

When the programmer is writing the specification, she can use our strongest postcondition predicate transformer to automatically obtain the postconditions for each operation. To execute our predicate transformer for operation `place_bid`, the input the programmer needs to provide to our tool is the one below, followed by the generated postcondition:

```
1  place_bid (b : int) (state : state) : unit
2    requires { b > 0 }
3    requires { state.open }
4    requires { state.winning_bid ≤ 0 }
5  = state.bid ← append (state.bid, make (1,b))
```

```
1  exists v0. state.bid = append (v0, make (1,b)) &&
2    b > 0 &&
3    state.open &&
4    state.winning_bid ≤ 0
```

The generated postcondition states that there exists a value `v0` representing the previous collection of placed bids. Then, a bid `b` is inserted in `v0`. The remaining information from the generated postcondition refers to the propagation of the preconditions from the `place_bid` operation. As for operation `close_auction`, we cannot apply our predicate transformer. This happens because our strongest postcondition calculus does not support any kind of loop constructors.

Comparing the postconditions of each operation from Figure 4.7 and the generated postcondition for `place_bid`, we verify that the latter are more verbose. However, they

```
let ghost place_bid_close_auction_commutativity () : (state,state)
  ensures  { match result with
           | x1, x2 → state_equality x1 x2
           end }
=  let ghost b1 = any int in
   let ghost state1 = any state in
   let ghost state2 = any state in
   assume { ((b1 > 0 ∧
    open state1 = True ∧
    winning_bid state1 ≤ 0 ∧
    (forall i:int.
    i ≥ 0 ∧ i < length (bid state1) → (bid state1)[i] > 0)) ∧
    length (bid state2) ≥ 0 ∧
    open state2 = True ∧
    winning_bid state2 ≤ 0 ∧
    (forall i:int.
    i ≥ 0 ∧ i < length (bid state2) → (bid state2)[i] > 0)) ∧
    state_equality state1 state2 };
   close_auction state1;
   place_bid b1 state1;
   place_bid b1 state2;
   close_auction state2;
   (state1, state2)

```

Figure 4.9: Generated commutativity analysis function for the auction application.

are automatically generated. This helps the programmer to understand which postcondition must be supplied. In fact, an appropriate witness for the existentially quantified variable v0 is `(old state).bid`, with which we recover the postconditions supplied in Figure 4.7.

Similar to the banking application, in this case study the programmer also needs to specify the equality relation between sets `state_equality`. Since there is only one pair of different operations, our tool only generates one function for the commutativity analysis for the auction application. The generated function is illustrated in Figure 4.9. In this case, we are not able to prove that the preconditions of operation `place_bid` are preserved after the execution of operation `close_auction`. This happens because by closing the auction, the flag `open` from the state of the application is changed, and in order for a user to place a bid, the auction must be open. That said, we assume that these operations cannot be safely executed concurrently. Since there is a verification condition that is not discharged, prior to the return of the pair of the final states, one cannot check if the states are the same, so we assume they do not commute. Similar to the bank application, in this case study there is not a commutativity issue that can be solved with the introduction of a CRDT in the system.

In this case study, our tool generates the stability analysis functions that are presented in Figure 4.10. Every verification condition generated for function `place_bid_stability` is discharged, ensuring that it is not conflicting with itself. However, for the stability

```
let ghost close_auction_stability () : ()
=  let ghost state1 = any state in
   assume { (length (bid state1) ≥ 0 ∧
    open state1 = True ∧
    w state1 ≤ 0 ∧
    (forall i:int.
    i ≥ 0 ∧ i < length (bid state1) → (bid state1)[i] > 0)) ∧
    length (bid state1) ≥ 0 ∧
    open state1 = True ∧
    w state1 ≤ 0 ∧
    (forall i:int.
    i ≥ 0 ∧ i < length (bid state1) → (bid state1)[i] > 0) };
   close_auction state1;
   close_auction state1

let ghost place_bid_stability () : ()
=  let ghost b1 = any int in
   let ghost state1 = any state in
   let ghost b2 = any int in
   assume { (b1 > 0 ∧
    open state1 = True ∧
    w state1 ≤ 0 ∧
    (forall i:int.
    i ≥ 0 ∧ i < length (bid state1) → (bid state1)[i] > 0)) ∧
    b2 > 0 ∧
    open state1 = True ∧
    w state1 ≤ 0 ∧
    (forall i:int.
    i ≥ 0 ∧ i < length (bid state1) → (bid state1)[i] > 0) };
   place_bid b1 state1;
   place_bid b2 state1

```

Figure 4.10: Generated stability analysis functions for the auction application.

analysis function `close_auction_stability`, we are not able to prove the preservation of the preconditions for the second call to the operation so `close_auction`. This happens because once an auction is closed, then it cannot be closed again. The proof effort for the commutativity and stability analysis functions can be seen in Figure 4.11.

In this application, there are some conflicts between operations, specifically `close_auction` conflicts with `place_bid` and with itself. Since at this point the analysis of the application is done, the programmer is aware of the conflicting operations so now she can provide a token system to our tool, and verify if it is sound. A sound token system for this application can be seen below:

```
token close_auction t1
token place_bid t2
t1 conflicts t1
t1 conflicts t2
```

This token system above states that operation `close_auction` has associated the token

Proof obligations		Alt-Ergo 2.3.0	CVC4 1.5	Z3 4.5.0
lemma VC for place_bid_close_auction_commutativity	lemma precondition	0.01		
	lemma precondition	0.00		
	lemma precondition	0.01		
	lemma precondition	0.00		
	lemma precondition	(1s)	0.03	FAILURE
	lemma precondition	0.01		
	lemma precondition	0.01		
	lemma precondition	0.00		
	lemma precondition	0.01		
	lemma precondition	0.01		
	lemma precondition	0.01		
	lemma precondition	0.01		
	lemma precondition	0.01		
	lemma precondition	0.01		
	lemma postcondition	(1s)	0.04	FAILURE
lemma VC for close_auction_stability	lemma precondition	0.01		
	lemma precondition	0.00		
	lemma precondition	0.00		
	lemma precondition	0.00		
	lemma precondition	(1s)	0.02	FAILURE
	lemma precondition	0.00		
lemma VC for place_bid_stability		0.01		

Figure 4.11: Statistics of the proof effort of the generated functions for the auction application.

`t1` and operation `place_bid` has associated the token `t2`. The conflict relation from the token system says that the operation `close_auction` is conflicting with itself and with operation `place_bid`. As presented in Section 3.3, when two operation-level tokens are declared as conflicting, our tool does not generate any verification condition for its analysis. That said, for this application our tool would only generate the verification conditions for the stability analysis of the `place_bid` operation. So, the consistency model depicted by this token system, states that the only operation that can be safely executed concurrently with another operation is `place_bid`, only being able to be safely executed concurrently with itself. This consistency model is very strict for the application however, it cannot be further refined so it is the less strict consistency model that preserves data integrity.

4.3 Courseware Application

In this section we present a last case study. It consists of a school registration system composed by a set of students, a set of courses and an enrollment relation between students and courses. As we did in the previous case studies, for this application we also abstract from security aspects, such as any user being able to perform any action offered by the application's API. The implementation and specification for this application is illustrated in Figure 4.12.

```
type state [@state] = {
  mutable students : fset int;
  mutable courses  : fset int;
  mutable enrolled : fset (int,int);
}
invariant{ forall i,j. mem (i,j) enrolled →
    mem i students ∧ mem j courses }

let ghost addCourse (course : int) (state : state): unit
  requires { course > 0 }
  ensures  {mem course state.courses}
= state.courses ← add course state.courses

let ghost addStudent (student : int) (state : state): unit
  requires { student > 0 }
  ensures  {mem student state.students}
= state.students ← add student state.students

let ghost enroll (student course : int) (state : state): unit
  requires { student > 0 ∧ course > 0 }
  requires { mem student state.students }
  requires { mem course state.courses }
  ensures  { mem (student,course) state.enrolled }
= state.enrolled ← add (student,course) state.enrolled

let ghost remCourse (course : int) (state : state): unit
  requires { course > 0 }
  requires { forall i. not (mem (i, course) state.enrolled) }
  ensures  { not (mem course state.courses) }
  ensures  { forall c. c ≠ course →
             mem c (old state).courses ↔ mem c state.courses }
= state.courses ← remove course state.courses

predicate state_equality [@state_eq] (s1 s2 : state)
= s1.students == s2.students &&
  s1.courses  == s2.courses  &&
  s1.enrolled == s2.enrolled

```

Figure 4.12: Specification of the courseware application.

Proof obligations		Alt-Ergo 2.3.0	CVC4 1.5	Z3 4.5.0
lemma `VC for state`		0.01	0.07	0.03
lemma `VC for addCourse`		0.01	0.06	0.05
lemma `VC for addStudent`		0.01	0.08	0.03
lemma `VC for enroll`		0.02	0.10	0.08
lemma `VC for remCourse`	lemma `type invariant`	0.02	0.04	0.02
	lemma `postcondition`	0.01	0.03	0.02
	lemma `postcondition`	0.01	0.03	0.03

Figure 4.13: Statistics regarding the proof of the auction application.

In this application, `addCourse` adds a new course to the system, `addStudent` adds a new student to the system, `enroll` registers a student in a course, and `remCourse` removes a course from the system. The specifications of operations `addCourse` and `addStudent` are very similar in the sense that they ensure the course and student, respectively, are inserted in the system. The specification for the `enroll` operation states that both the student and the course that is going to be enrolled, must be in the system prior to its execution. Operation `enroll` ensures that the student will be enrolled in the course after its execution. Lastly, the specification for the `remCourse` operation indicates that the course that is going to be removed must not have a student enrolled in it. Also, this operation ensures that the specified course will be removed from the system, and the remaining courses will stay in the system. Another important thing to notice about the specifications of these operations is that they are all represented as `ghost` code. This happens because functions `add` and `remove` of the `fset` library from Why3, must be executed in a `ghost` environment. There is no issue with using `ghost` code in this case study, because the goal of CISE3 is not to verify the executable code of a distributed application. Instead, we are concerned with analysing relations between operations.

The state of this application is represented by a record with three fields, which are the three sets that store the students, courses and the enrolment relation between students and courses. There is one integrity invariant associated to the state of the application. This invariant states that at any point during the execution, if any student is enrolled in a course, then the student and course must exist in the system. The proof effort for this sequential program can be observed in Figure 4.13.

Similar to the previous two case studies, in this application the programmer also needs to specify a predicate (`state_equality`) used for the comparison between states. In order to compare two sets in Why3, a simple structural comparison is not enough to prove that one set is equal to another. To compare two sets one requires an extensional equality relation over sets, which can be achieved by using the function == from the Why3 `fset` library.

When the programmer is writing the specification, she can use our strongest postcondition predicate transformer to automatically obtain the postconditions for each operation. To execute our predicate transformer over operation `addCourse`, the programmer can provide to our tool the specification below, which is followed by the generated postcondition:

```
1  addCourse (course: int) (state: state) : unit
2    requires {course > 0}
3  = state.courses ← add (state.courses,course)
```

```
1  exists v0. state.courses = add (v0,course) &&
2    course > 0
```

This postcondition states that there exists a value v0 of the type `fset` and that a value course was inserted and the result of that insertion is now the set of courses of the system `state.courses`. The remaining information refers to the preconditions of operation `addCourse`. The input and output for the execution of our predicate transformer for operation `addStudent` is similar to the one we present above. As for operation `enroll`, the programmer provides the specification presented below, which is followed by the generated postcondition:

```
1  enroll (student, course: int) (state: state) : unit
2    requires {student > 0}
3    requires {course > 0}
4    requires {mem (state.courses,course)}
5    requires {mem (state.students,student)}
6  = state.enrolled ← add (state.enrolled,[student,course])
```

```
1  exists v0. state.enrolled = add (v0, (student,course)) &&
2    student > 0 &&
3    course > 0 &&
4    mem (state.courses,course) &&
5    mem (state.students,student)
```

The generated postcondition presented above states that there exists a set v0 and that the tuple (`student,course`) was inserted in it, and the result of that insertion is now the relation enrolled from the system `state.enrolled`. The remaining information from the generated postcondition refers to the preconditions of the `enroll` operation. Laslty, since operation `remCourse` has a `forall` constructor in its preconditions, our strongest postcondition calculus cannot be applied.

In Figure 4.12, we can observe four operations: `addCouse`, `addStudent`, `enroll` and `remCourse`. Since the application has four operations, there are six pairs of different operations. One of the generated functions for this application's commutativity analysis can be seen in Figure 4.14.

```
let ghost enroll_remCourse_commutativity () : (state, state)
  ensures  { match result with
             | x1, x2 → state_equality x1 x2
             end }
= let ghost student1 = any int in
  let ghost course1 = any int in
  let ghost state1 = any state in
  let ghost course2 = any int in
  let ghost state2 = any state in
  assume { (((student1 > 0 ∧ course1 > 0) ∧
         mem student1 (students state1) ∧ mem course1 (courses state1)) ∧
         course2 > 0 ∧
         (forall i:int. not mem (i, course2) (enrolled state2))) ∧
         state_equality state1 state2 };
  remCourse course2 state1;
  enroll student1 course1 state1;
  enroll student1 course1 state2;
  remCourse course2 state2;
  (state1, state2)

```

Figure 4.14: Generated commutativity analysis function for `enroll` and `remCourse`.

In this case we are not able to prove that the preconditions of operation `enroll` are preserved after the execution of `remCourse`. This happens because when `remCourse` removes one course from the system, then that course can be the course that a student is trying to enroll in a concurrent execution of operation `enroll`. That said, we assume that these operations cannot be safely executed concurrently. Since our tool is not able to prove every verification condition prior to the return of the pair of the final states, one cannot check if the states are the same at the end, so we cannot assert anything about their commutativity. Apart from the functions `enroll_remCourse-commutativity` and `addCourse_remCourse_commutativity`, all the remaining functions regarding the commutativity and stability analysis of pairs of different operations are proved automatically, meaning that the operations involved in the remaining pairs commute and do not conflict.

The function `addCourse_remCourse_commutativity` shows us a typical commutativity issue over sets, which is the concurrent removal and addition of elements. Let us consider the following example: if in one replica a user tries to add the student Filipe and consequently removes him from the system, and in another replica the alternative order of execution occurs, then in one replica Filipe is in the system but in the other one he is not. This commutativity issue can be easily solved with a CRDT from our provided library. To solve this issue the programmer needs to replace the Why3 set, that stores the collection of courses, by a Remove-Wins set CRDT. The new code of the application with the introduction of the CRDT can be seen in Figure 4.15. The generated stability analysis functions for this application are presented in Figure 4.16.

In this case study, every verification condition generated for every stability analysis function is proved automatically, allowing us to assume that every operation can be

```
1  type state [@state] = {
2    mutable students : fset int;
3    mutable courses  : remove_wins_set int;
4    mutable enrolled : fset (int,int);
5  }
6  invariant{ forall i,j. mem (i,j) enrolled →
7      mem i students ∧ in_set j courses }
8
9  let ghost addCourse (course : int) (state : state): unit
10   requires { course > 0 }
11   ensures  { state.courses.remove_wins_add =
12       add course (old state).courses.remove_wins_add }
13   ensures  { state.courses.remove_wins_removes ==
14       (old state).courses.remove_wins_removes}
15 = add_element course state.courses
16
17 let ghost addStudent (student : int) (state : state): unit
18   requires { student > 0 }
19   ensures  {mem student state.students}
20 = state.students ← add student state.students
21
22 let ghost enroll (student course : int) (state : state): unit
23   requires { student > 0 ∧ course > 0 }
24   requires { mem student state.students }
25   requires { in_set course state.courses }
26   ensures  { mem (student,course) state.enrolled }
27 = state.enrolled ← add (student,course) state.enrolled
28
29 let ghost remCourse (course : int) (state : state): unit
30   requires { course > 0 }
31   requires { mem course state.courses.remove_wins_add}
32   requires { forall i . not (mem (i,course) state.enrolled)}
33   ensures  { not (in_set course state.courses) }
34   ensures  { forall c. c ≠ course → mem c (old state).courses.
     remove_wins_removes
35     ↔ mem c state.courses.remove_wins_removes}
36 = remove_element course state.courses
37
38 predicate state_equality [@state_eq] (s1 s2 : state)
39 =  s1.students == s2.students &&
40    equal s1.courses s2.courses &&
41    s1.enrolled == s2.enrolled
42
```

Figure 4.15: Courseware application with a CRDT.

```
let ghost remCourse_stability () : ()
= let ghost course1 = any int in
  let ghost state1 = any state in
  let ghost course2 = any int in
  assume { (course1 > 0 ∧
    (forall i:int. not mem (i, course1) (enrolled state1))) ∧
    course2 > 0 ∧
    (forall i:int. not mem (i, course2) (enrolled state1)) };
  remCourse course1 state1;
  remCourse course2 state1

let ghost enroll_stability () : ()
= let ghost student1 = any int in
  let ghost course1 = any int in
  let ghost state1 = any state in
  let ghost student2 = any int in
  let ghost course2 = any int in
  assume { ((student1 > 0 ∧ course1 > 0) ∧
    mem student1 (students state1) ∧ mem course1 (courses state1)) ∧
    (student2 > 0 ∧ course2 > 0) ∧
    mem student2 (students state1) ∧ mem course2 (courses state1) };
  enroll student1 course1 state1;
  enroll student2 course2 state1

let ghost addStudent_stability () : ()
= let ghost student1 = any int in
  let ghost state1 = any state in
  let ghost student2 = any int in
  let ghost state2 = any state in
  assume { state_equality state1 state2 ∧ student1 > 0 ∧ student2 > 0 };
  addStudent student1 state1;
  addStudent student2 state1

let ghost addCourse_stability () : ()
= let ghost course1 = any int in
  let ghost state1 = any state in
  let ghost course2 = any int in
  let ghost state2 = any state in
  assume { state_equality state1 state2 ∧ course1 > 0 ∧ course2 > 0 };
  addCourse course1 state1;
  addCourse course2 state1

```

Figure 4.16: Generated stability analysis functions for the courseware application.

safely executed concurrently. The proof effort for the commutativity and stability analysis functions prior to the introduction of a CRDT, is presented in Figure 4.17.

As mentioned before, in this application there is only one case of conflicting operations: `enroll` and `remCourse`. A possible solution for this conflict is the mutual exclusion of these operations. This means that whenever there is the possibility of a concurrent execution of `enroll` and `remCourse`, they must be executed sequentially to preserve data integrity. One way for the programmer to check if this specific consistency model is sound for this application, is to provide a token system that depicts it to our tool, and check if every generated verification condition is proved. The token system that represents the consistency model specified above is presented below:

```
1  token enroll   t1
2  token remCourse t2
3  t1 conflicts t2
```

In this token system the programmer defines two tokens `t1` and `t2` that are associated with operation `enroll` and `remCourse` respectively. The conflict relation of this token system states that these tokens are conflicting. Given this token system, our tool does not generate the function for the commutativity and stability analysis between `enroll` and `remCourse`. That said, every generated verification condition is proved so the consistency model represented by the token system is considered sound.

The consistency model associated to this token system is too strict for the application, and decreases its availability by a significant amount. However, this consistency model is not the best possible solution for this case study. By analysing the conflict involving operations `enroll` and `remCourse`, we conclude that it is due to the argument `course` from both operations. So, the programmer can refine the previous token system given this new information. This new and more refined token system is shown below:

```
1  argtoken enroll course t1
2  argtoken remCourse course t2
3  t1 conflicts t2
```

The token system above states that the argument `course` from operation `enroll` has associated the token `t1` and the argument `course` from operation `remCourse` has associated the token `t2`, and `t1` and `t2` are conflicting. This means that operations `enroll` and `remCourse` can only be executed if the values of the arguments `course` from each operation are different. This solution is less strict than automatically disallowing their concurrent execution. Regarding the generated verification conditions by our tool, with the newly provided token system, there is a change in the `assume` expression in the function `enroll_remCourse_commutativity`, presented in Figure 4.18. The only difference from the previous `assume` expression is the introduction of the assertion `(not (course1 = course2))`. Given the above token system, every verification condition generated by our tool is proved automatically, so the underlying consistency model is sound.

Proof obligations					Alt-Ergo 2.3.0	CVC4 1.5	CVC4 1.6	Z3 4.5.0
lemma VC for enroll_remCourse_commutativity	lemma precondition					0.02		
	lemma precondition					0.05		
	lemma precondition					0.02		
	lemma precondition					0.04		
	lemma precondition				(1s)	(1s)		(1s)
	lemma precondition					0.02		
	lemma precondition					0.05		
	lemma precondition					0.06		
	lemma precondition					0.02		
	lemma precondition				(1s)	(1s)		(1s)
	lemma postcondition				0.02			
					0.14			
				(1s)	(1s)		(1s)	
lemma VC for addStudent_remCourse_commutativity	lemma precondition					0.02		
	lemma precondition					0.05		
	lemma precondition					0.03		
	lemma precondition					0.02		
	lemma precondition					0.03		
	lemma precondition					0.05		
	lemma postcondition			(1s)	(1s)		(1s)	
					0.08			
					0.04			
lemma VC for addStudent_enroll_commutativity	lemma precondition					0.03		
	lemma precondition					0.10		
	lemma precondition					0.06		
	lemma precondition					0.03		
	lemma precondition					0.03		
	lemma precondition					0.02		
	lemma precondition				(1s)	(1s)		(1s)
	lemma precondition					0.05		
	lemma postcondition			(1s)	(1s)		(1s)	
					0.04			
				(1s)	(1s)		(1s)	
lemma VC for addCourse_remCourse_commutativity	lemma precondition					0.01		
	lemma precondition					0.05		
	lemma precondition					0.02		
	lemma precondition					0.01		
	lemma precondition					0.02		
	lemma precondition					0.05		
	lemma postcondition				0.02			
				(1s)	(1s)		(1s)	
					0.05			
lemma VC for addCourse_enroll_commutativity	lemma precondition					0.03		
	lemma precondition					0.06		
	lemma precondition					0.08		
	lemma precondition					0.01		
	lemma precondition					0.02		
	lemma precondition					0.03		
	lemma precondition					0.05		
	lemma precondition				(1s)	(1s)		(1s)
	lemma postcondition				(1s)	(1s)		(1s)
lemma VC for addCourse_addStudent_commutativity	lemma precondition					0.02		
	lemma precondition					0.02		
	lemma precondition					0.02		
	lemma precondition					0.03		
	lemma postcondition				(1s)	(1s)		(1s)
					(1s)	(1s)		(1s)
						0.04		
lemma VC for remCourse_stability							0.06	
lemma VC for enroll_stability							0.06	
lemma VC for addStudent_stability							0.03	
lemma VC for addCourse_stability							0.03	

Figure 4.17: Statistics of the proof effort of the generated functions for the courseware application.

```
1      assume { not course1 = course2 ∧
2        (((student1 > 0 ∧ course1 > 0) ∧
3        mem student1 (students state1) ∧
4        in_set course1 (courses state1)) ∧
5        course2 > 0 ∧
6        mem course2 (remove_wins_add (courses state2)) ∧
7        (forall i:int. not mem (i, course2) (enrolled state2))) ∧
8        state_equality state1 state2 };
9
```

Figure 4.18: Modified assume expression from `enroll_remCourse_commutativity`.

Chapter 5

Related Work

This chapter presents and discusses other tools used for the verification of weakly consistent applications. We focus on static analysis tools since this is the focus of this thesis. In the end of this chapter we present a table comparing every studied tool with the tool we developed.

5.1 Quelea

Quelea is a declarative programming model and tool for eventually consistent data stores. This tool offers an expressive contract language which allows the programmer to specify fine-grained application consistency properties. It is implemented as an extension of Haskell and runs over Cassandra, an eventually consistent distributed data store. The proof obligations are discharged with the help of the Z3 SMT solver. After a contract classification process occurs the most efficient and sound consistency level is assigned to each operation of the application [49].

For mapping operations to the appropriate consistency levels, the programmer needs to declare application-level consistency constraints on operations, as contracts. These contracts specify the set of allowed operation executions of the operation that has the associated contract. Any execution that does not show an anomaly is considered as a well-formed execution. By specifying the set of allowed executions of an operation, Quelea then uses Z3 to help with the process of contract classification. Contract classification leverages the power of Z3 to verify which of the previous specifications is the weakest and which one is the strongest. To show the comparison a $\leq$ relation is used, which says that if a specification ψx does not guarantee the constraints from specification ψy then: $\psi x \leq \psi y$. The consistency-level that is associated to the operation is the weakest consistency level that enables the preservation of the constraints of the operation's contract [49].

For the process of contract classification there is the need of contracts for each store consistency level and they must also be provided by the programmer. Quelea supports eventual, causal and strong consistency for operations, and for transactions it supports RC, MAV and RR. This feature is implemented on top of the interface exposed by Cassandra [49].

Comparing with CISE3 specification effort, the complexity of specifying contracts in Quelea is high and time consuming, since it makes the programmer reason about possible concurrent interferences. Additionally, there is no guarantee that the contracts are sufficient to assure the preservation of the application's invariants. Also, CISE3 offers a strongest postcondition predicate transformer, which Quelea does not. Due to these reasons, we can state that Quelea requires a more difficult specification process compared with CISE3. One similarity between Quelea and CISE3, is that they both allow the programmer to introduce CRDTs in the specification. Also, Quelea only supports one external prover, Z3, whereas our tool supports a wider variety. Lastly, CISE3 and Quelea are both capable of showing a counterexample whenever an issue is found.

5.2 Q9

Q9 is a programming framework for replicated data types (RDTs), equipped with a bounded verification technique that discovers and fixes weak consistency anomalies automatically [33]. It is embedded in OCaml and its symbolic execution engine is implemented as a compiler pass that follows the typecheking in the OCaml 4.03 compiler. This framework is used to analyse applications that are executed on top of eventually consistent data stores. With the help of a symbolic execution engine it is possible to provide bounded guarantees regarding the correctness of a program. The symbolic execution engine explores a search space of abstract executions of a program. Each member of the search space corresponds to a state of the program parametrized over the bound on the number of concurrent effects [33]. For the verification to be done, the programmer only has to specify the invariants of the program.

The Q9 framework is divided into three components. The first component is a translator that translates high-level programs with implicit effects, to a representation with explicit effects. This first component serves to prepare the high-level program to be analysed and verified. In the second component, the verifier, we have the symbolic execution engine as its main element. It performs the bounded verification technique, given the k-bound as an input, and it works in a loop with an SMT solver, Z3 for example. The k-bound serves to bound the number of concurrent effects that can occur concurrently in a program p. By bounding the number of concurrent effects we also bound the number of replicas that process them. This framework does not tackle the problem of fully unbounded verification, it can only guarantee k-safety of a program [33]. The third and final component, the solver, handles the automatic reparations of the program's consistency anomalies.

The verification process executed in the second component progresses one operation at a time and then one transaction at a time. Each operation/transaction is verified against the current consistency model being analysed, starting at eventual consistency. If the verification determines the invariants are not preserved, then a counterexample is obtained from the solver. After that, the symbolic execution engine in the verifier computes the abstract representation, given the obtained counterexample. Then, the solver tests various consistency models until it finds the less strict model that preserves the invariants, given the abstract representation. Finally the solver informs the verifier of the selected consistency model, and the verification process is repeated until all operations preserve the program's invariants. Otherwise, the strong consistency model is employed in the system.

When the verification process is being carried out, the symbolic execution engine generates verification conditions (VCs) for the SMT solver, based on the k-safety definition [33]. These VCs are then encoded as satisfiability queries in Z3, and if they are satisfiable then the model that is being tested, can be used for the consistency repair process, as stated above.

CISE3 does not restrict the number of concurrent effects that can occur over the state, so CISE3 can guarantee a full verification of the application, instead of a bounded one. However, Q9 has an automatic mechanism to repair issues discovered by the tool, and CISE3 does not. Q9 also does not provide a strongest postcondition predicate transformer but CISE3 does. This feature allows CISE3 to have a lighter specification process for the programmer. Q9 also provides a collection of CRDT specifications, like CISE3. Also, Q9 only supports one external prover, Z3, whereas our tool supports a wider variety. Regarding the exhibition of counterexamples, both Q9 and CISE3 possess that feature.

5.3 Repliss

Repliss [52] is a verification tool used to reason about applications built over weakly consistent databases. This tool allows programmers to write the consistency sensitive code of their application using a domain specific language. Additionally, the programmer also needs to write the specification of the program and its integrity invariants. Given this information, Repliss translates this program into a sequential Why3 program, and if this Why3 program is proved as being correct, then the initial program is ensured to be correct.

The application's code can employ CRDTs that are either provided by the underlying database or by the programmer. This way the programmer delegates all synchronization aspects to the database, which is the central idea of this tool and proof technique. Depending on the application, the programmer must decide which is the most adequate data type, so its access becomes more efficient.

To verify if an application is correct, the tool avoids reasoning about all possible traces. Instead, Repliss's proof technique reasons about one procedure invocation at a time. In

the scope of this tool, the applications offer to the clients an API, which consists of a set of procedures that they can use to communicate with the application. An invocation of a procedure executes code sequentially and interacts with the data store [52]. In order to verify if a procedure invocation is correct, Repliss firstly checks if the initial state maintains the invariant. Then, based on the application's code, it verifies if the procedure invocation maintains the invariant. When verifying a single procedure invocation, Repliss does not consider concurrent executions at all program points. Since Repliss analyses each procedure individually, its technique is modular like CISE's proof rule.

In comparison with our tool, the DSL presented in Repliss is more limited than the WhyML since it does not allow the programmer to write contracts for operations. So, by developing our tool over Why3, we can write better specifications more easily. Other than that, Repliss is similar to our tool in the sense that the reasoning about the operation's relations is done through the analysis of a sequential Why3 program. This allows the programmer to introduce CRDTs in the application. Another difference this tool has comparing with CISE3, is the fact that the latter provides a strongest postcondition predicate transformer to ease the specification process, while Repliss does not. Since Repliss tries to prove a sequential Why3 program with the Why3 framework, then it can access the same external provers as CISE3 and also display counterexamples when an issue is found.

5.4 Hamsaz

Hamsaz [31] is a static analysis tool that given a system's specification and using the CVC4 SMT solver, decides the pairs of conflicting and dependent operations. The specification of an object must include the state type and invariants as well as its operations [31]. The goal of this tool is to automatically synthesize a correct-by-construction replicated system that guarantees integrity and convergence. Also this system avoids unnecessary coordination having in mind the conflict and causal dependency relations between operations. The tool uses a static analysis approach in order to calculate the conflict and dependency relations. The core of Hamsaz's approach is a sufficient condition for integrity and convergence of replicated systems called *well-coordination*. Well-coordination states that conflicting operations need synchronization, while operations that are dependent of each other need causality.

In order to say that two operations are not conflicting they must *S-commute* and *P-concur* with each other. Two operations *S-conflict* if from the same state prior to executing those operations in different orders, results in different final states thus, requiring synchronization. On the other hand, if from the same initial state, both orders of execution of the operations reach the same final state, then they *S-commute* (state commute). Operation *o1 P-concurs* (permissible-concurs) with operation *o2* if *o1* is invariant-sufficient or if *o1 P-R-commutes* with *o2* [31]. An operation is considered as *invariant-sufficient* if it never breaks the invariant when it is executed. However, not all operations are invariant-sufficient, for example in a bank application the *withdraw* operation could put

the balance to a negative value thus, breaking the invariant. Operation *o1 P-R-commutes* (permissible-right-commutes) with operation *o2* if *o1* stays permissible when executed right after *o2* [31]. Some operation *o* is said to be *permissible* in some state σ if it satisfies the invariant in σ and if it still preserves it after its execution. If two operations do not *P-concur* then they *P-conflict* (permissible conflict) and require synchronization.

As mentioned above, invariant-sufficient operations always preserve the invariant. However, there are operations that only preserve the invariant depending on previous operations calls. An operation *o2* is considered as being *independent* of *o1* if *o2* is invariant-sufficient or if it *P-L-commutes* with *o1*. An operation *o2* is said to *P-L-commute* with operation *o1* if it remains permissible if executed prior to *o1* [31]. If *o2* is dependent of *o1* then the execution of *o2* should be postponed, so *o1* can be executed first.

Hamsaz's analysis for the conflicting operations is similar to the the analysis conducted by our tool. However, Hamsaz performs an analysis to find causally dependent operations which CISE3 does not. This analysis allows Hamsaz to only assume eventual consistency. Since CISE3 does not perform this analysis we assume causal consistency, which is a less relaxed assumption. Another similarity between Hamsaz and CISE3 is that both tools allow the use of CRDTs. An advantage that CISE3 has over Hamsaz is that it provides a strongest postcondition predicate transformer that eases the specification process. One disadvantage CISE3 has over Hamsaz is that the latter, offers a set of protocols for the automatic resolution of conflicts. Lastly, CISE3 also supports more than one external prover and is capable of displaying counterexamples whenever an issue is found during the analysis of the application, but Hamsaz only offers the latter feature.

5.5 CISE tool

The first version of the CISE tool proposes an approach where the programmer needs to provide the specification of the application's operations, the state of the application, and the integrity invariants. For each operation of the application the programmer needs to also provide its sequential implementation. The analysis of this tool consists of performing the three proof obligations from the CISE proof rule, like our tool does. This tool automates the proof rule by discharging the generated verification conditions using an SMT solver, in this case Z3. If a verification condition is not discharged then a counter example is provided, that the programmer can visualize in order to understand the source of the issue [41]. However, the counterexamples presented by Z3 are difficult to understand.

For this tool the programmer needs to interact directly with Z3 low level APIs directly, making it difficult to specify the required input. Since CISE3 is implemented over Why3 we provide a high-level programming language (WhyML) making it easier to specify the input for our tool, comparing with the first version of the CISE tool. Additionally, our tool offers the programmer an automatic mechanism for the generation of strongest postconditions, which this tool does not. That being said, the specification effort for the

CISE tool is higher comparing with our tool. Also, the CISE tool only supports one prover, the Z3 SMT solver, but CISE3 is implemented over Why3, so it supports a wider range of external provers. Another advantage of CISE3 is that our token system language allows the programmer to define more fine-grained tokens, instead of only allowing tokens at operation level, like this tool does. Lastly CISE3 allows for the introduction of CRDTs in the application in order to solve commutativity issues and the CISE tool does not.

5.6 CEC tool

The CEC tool is the second version of the CISE tool. This tool follows the same principle as its predecessor however, it provides a high-level verification language, an extension of Boogie [4, 40]. In CEC the programmer writes the specifications in Boogie, which then generates a set of verification conditions. These verification conditions are then sent to the Z3 SMT solver, to be discharged. Also, like its predecessor, the CEC tool is also capable of showing a counterexample whenever a conflict is detected, thus helping the programmer understand what caused the conflict [38].

As an input for the CEC tool the programmer needs to specify the operations, state, and integrity invariants of the application. Each operation is defined as a sequence of *reads* and *updates* and has a precondition associated. In order to define the consistency model to be employed over the application, the programmer provides a token system that consists of a set of tokens and a conflict relation over them, like in our tool and in the CISE tool [38]. The CEC tool allows the programmer to specify parameters from operations that are conflicting.

The specification effort from the CEC tool is similar to the one from CISE3, since in both the programmer writes the specification using a high-level specification language. However, CISE3 has the advantage of offering an automatic generator of strongest post-conditions. Regarding the resolution of commutativity issues using CRDTs, both the CEC tool and CISE3, support this feature. Another similarity is the performed analyses from CEC and CISE3, since they both rely on the three proof obligations provided by the CISE proof rule. Lastly, this tool only supports one external prover, the Z3 SMT solver, whereas CISE3, since is implemented over Why3, supports a wider variety of external provers.

5.7 Tool Comparison

In this section we summarise the comparisons between every studied tool. To resume this section, we also present Table 5.1.

Counterexamples: Every studied tool has the ability of producing a counterexample when an issue is found. However, apart from Repliss and CISE3 the counterexamples that are presented are hard to understand since they are presented in the language of

the SMT solver used by the tool. Since Repliss and CISE3 use Why3 for the verification process, then the counterexamples they produce are more readable.

Bounded Verification: From the studied tools Q9 is the only tool that bounds the number of possible concurrent updates for the verification process. This is a disadvantage from the other studied tools, which do not perform a bounded verification process.

Automatic Resolution: Only Q9 and Hamsaz present an automatic resolution policy.

External Provers Support: Apart from Repliss and CISE3, every studied tool only supports one external prover, an SMT solver. Quelea, Q9, the CISE and CEC tools support Z3 and Hamsaz supports CVC4. Since Repliss and CISE3 use Why3 for the verification process, then they have access to more than 25 external provers.

Consistency Assumptions: CISE3 and its predecessors (CISE and CEC tools) assume causal consistency because they are based on the proof rule from the CISE proof rule. For CISE3 this assumption can be relaxed as discussed in Section 6.2. The remaining analysed tools assume eventual consistency which is more relaxed when comparing to CISE3.

CRDTs support: Only the CISE tool is not capable of supporting the introduction of CRDTs in the application specification. Another positive point from CISE3 is that we also provide a library of verified CRDTs that programmers can use.

Specification Effort: Comparing the specification efforts from all the studied tools, we reach the conclusion that specifying applications for Quelea and the CISE tool is difficult. For Quelea the programmer needs to specify fine-grained contracts about the operations of the application and reason about concurrent interference from other operations. As for the CISE tool, the programmer needs to use the low-level API's from Z3 in order write the specification.

Predicate Transformer: From the studied tools only CISE3 provides a predicate transformer, more precisely a strongest postcondition generator. This metric is also related to the previous one, since the use of a predicate transformer simplifies the specification effort for the programmer. This way we can say that the specification effort for CISE3 is slightly easier comparing to the other studied tools.

	Counter Examples	Bounded Verification	Automatic Resolution	External Provers	Consistency Assumptions	CRDTs Support	Specification Effort	Predicate Transformer
Quelea	Yes	No	No	One	Eventual Consistency	Yes	High	No
Q9	Yes	Yes	Yes	One	Eventual Consistency	Yes	Average	No
Repliss	Yes	No	No	Multiple	Eventual Consistency	Yes	Average	No
Hamsaz	Yes	No	Yes	One	Eventual Consistency	Yes	Average	No
CISE tool	Yes	No	No	One	Causal Consistency	No	High	No
CEC tool	Yes	No	No	One	Causal Consistency	Yes	Average	No
CISE3	Yes	No	No	Multiple	Causal Consistency	Yes	Average	Yes

Table 5.1: Comparison between the studied tools.

Chapter 6

Conclusion

This chapter discusses the contributions from our work, and presents some future work that can be done in order to improve said contributions.

6.1 Discussion

In this thesis we propose an automatic approach for the static analysis of weakly consistent applications using the deductive verification framework Why3. To validate our approach we designed several case studies which we analysed with our tool as seen in Section 4.

The presented approach follows the proof rule proposed by CISE [28] and it takes inspiration from previous tools implemented with the same purpose of trying to automate the mentioned proof rule. Our approach is similar to these tools because it performs the same analysis however, it improves them in the sense that it provides more features and enhances the aspects already presented in the previous versions, as seen in Section 5.

To analyse an application we propose that the programmer provides as an input its sequential specification and implementation. After that, the programmer uses CISE3 to reason about the pairs of conflicting operations from the application. With this information the programmer can then use a CRDT to solve commutativity issues, and specify a token system in order to assess if a specific consistency model is sound over the application. In order to aid the programmer's specification effort, our tool also features a strongest postcondition predicate transformer.

6.2 Future Work

From this work we propose some main directions regarding future work, that can improve our approach and make it more robust.

Firstly we want to add the causal dependency analysis from Hamsaz [31] to our already existing set of analyses. This analysis is capable of finding the set of causally dependent operations from an application. At this moment without this analysis our approach assumes causal consistency however, with the introduction of this analysis we are able to relax that assumption and switch it to eventual consistency. We already reasoned about how to include this analysis and we already have the solution. This is not yet implemented, by the time of writing, but we expect to have it by the time of presentation.

Secondly we want to improve and expand our library of CRDTs. The CRDTs from our current library have very simple implementations, which can be optimized. Also, by expanding our library with more CRDTs, it can be used to solve commutativity issues from a wider variety of examples.

Lastly, we want to expand our target language for our strongest postcondition calculus. Currently our target language does not support any form of loop constructors so, our goal is to make it support *for...each* loops. In applications that operate over replicated databases, *for...each* constructors are the most used kind of loops, so it would be important for our target language to support them. Also, since they are bounded loop constructors it is easier to prove their termination. We intend to follow the approach presented by Filliâtre and Pereira, who proposed a modular specification of iteration regardless of the underlying implementation [22]. In particular, the authors show how to verify correctness of several iteration clients and implementations, based on bounded loops in the style of the *for...each* constructor.

6.3 Final Remarks

Ultimately, we believe the objectives we set for this thesis were fulfilled. We believe our approach represents a valid means of analysing applications operating over replicated databases. The planned future work will allow our approach to become more robust and useful over a wider set of applications.

Bibliography

[1] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Highly Available Transactions: Virtues and Limitations." In: *Proc. VLDB Endow.* 7.3 (Nov. 2013), pp. 181–192. ISSN: 2150-8097. DOI: 10.14778/2732232.2732237. URL: http://dx.doi.org/10.14778/2732232.2732237.

[2] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. "Putting Consistency Back into Eventual Consistency." In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 6:1–6:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741972. URL: http://doi.acm.org/10.1145/2741948.2741972.

[3] V. Balegas, C. Li, M. Najafzadeh, D. Porto, A. Clement, S. Duarte, C. Ferreira, J. Gehrke, J. Leitão, N. Preguiça, R. Rodrigues, M. Shapiro, and V. Vafeiadis. "Geo-Replication: Fast If Possible, Consistent If Necessary." In: *IEEE Data Engineering Bulletin* 39.1 (2016).

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. "Boogie: A Modular Reusable Verifier for Object-oriented Programs." In: *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*. FMCO'05. Amsterdam, The Netherlands: Springer-Verlag, 2006, pp. 364–387. ISBN: 3-540-36749-7, 978-3-540-36749-9. DOI: 10.1007/11804192_17. URL: http://dx.doi.org/10.1007/11804192_17.

[5] N. Benton. "Simple Relational Correctness Proofs for Static Analyses and Program Transformations." In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: ACM, 2004, pp. 14–25. ISBN: 1-58113-729-X. DOI: 10.1145/964001.964003. URL: http://doi.acm.org/10.1145/964001.964003.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. "A Critique of ANSI SQL Isolation Levels." In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. San Jose, California, USA: ACM, 1995, pp. 1–10. ISBN: 0-89791-731-6. DOI: 10.1145/223784.223785. URL: http://doi.acm.org/10.1145/223784.223785.

[7] I. S.E. I. Bertrand Meyer and Usa. “Design by Contract: Making Object-Oriented Programs That Work.” In: *Proceedings of the Technology of Object-Oriented Languages and Systems - Tools-25*. TOOLS ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 360–. ISBN: 0-8186-8485-2. URL: http://dl.acm.org/citation.cfm?id=832251.832695.

[8] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. “Why3: Shepherd Your Herd of Provers.” In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wroclaw, Poland, 2011, pp. 53–64. URL: https://hal.inria.fr/hal-00790310.

[9] A. Charguéraud, J.-C. Filliâtre, M. Pereira, and F. Pottier. *VOCAL – A Verified OCAml Library*. ML Family Workshop 2017. Sept. 2017. URL: https://hal.inria.fr/hal-01561094.

[10] M. Clochard, C. Marché, and A. Paskevich. “Verified Programs with Binders.” In: *Programming Languages meets Program Verification*. San Diego, United States: ACM Press, Jan. 2014. URL: https://hal.inria.fr/hal-00913431.

[11] M. Clochard, L. Gondelman, and M. Pereira. “The Matrix Reproved (Verification Pearl).” In: *Journal of Automated Reasoning* 60.3 (2018), pp. 365–383. ISSN: 1573-0670. DOI: 10.1007/s10817-017-9436-2. URL: https://doi.org/10.1007/s10817-017-9436-2.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. “PNUTS: Yahoo!’s Hosted Data Serving Platform.” In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1277–1288. ISSN: 2150-8097. DOI: 10.14778/1454159.1454167. URL: http://dx.doi.org/10.14778/1454159.1454167.

[13] S. Dailler, D. Hauzar, C. Marché, and Y. Moy. “Instrumenting a weakest precondition calculus for counterexample generation.” In: *J. Log. Algebr. Meth. Program.* 99 (2018), pp. 97–113. DOI: 10.1016/j.jlamp.2018.05.003. URL: https://doi.org/10.1016/j.jlamp.2018.05.003.

[14] S. Dailler, C. Marché, and Y. Moy. “Lightweight Interactive Proving inside an Automatic Program Verifier.” In: *Proceedings of the Fourth Workshop on Formal Integrated Development Environment, F-IDE, Oxford, UK, July 14, 2018*. 2018.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. “Dynamo: Amazon’s Highly Available Key-value Store.” In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: http://doi.acm.org/10.1145/1323293.1294281.

[16] E. W. Dijkstra. "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs." In: *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*. Ed. by D. Gries. New York, NY: Springer New York, 1978, pp. 166–175. ISBN: 978-1-4612-6315-9. DOI: 10.1007/978-1-4612-6315-9_14. URL: https://doi.org/10.1007/978-1-4612-6315-9_14.

[17] E. W. Dijkstra. *A Discipline of Programming*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. ISBN: 013215871X.

[18] F. Elberzhager, J. Münch, and V. T. N. Nha. "A Systematic Mapping Study on the Combination of Static and Dynamic Quality Assurance Techniques." In: *Inf. Softw. Technol.* 54.1 (Jan. 2012), pp. 1–15. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2011.06.003. URL: http://dx.doi.org/10.1016/j.infsof.2011.06.003.

[19] J.-C. Filliâtre. "Deductive software verification." In: *International Journal on Software Tools for Technology Transfer* 13.5 (2011), p. 397. ISSN: 1433-2787. DOI: 10.1007/s10009-011-0211-0. URL: https://doi.org/10.1007/s10009-011-0211-0.

[20] J.-C. Filliâtre and C. Marché. "The Why/Krakatoa/Caduceus Platform for Deductive Program Verification." In: *Computer Aided Verification*. Ed. by W. Damm and H. Hermanns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 173–177. ISBN: 978-3-540-73368-3.

[21] J.-C. Filliâtre and A. Paskevich. "Why3 – Where Programs Meet Provers." In: *ESOP'13 22nd European Symposium on Programming*. Vol. 7792. LNCS. Rome, Italy: Springer, Mar. 2013. URL: https://hal.inria.fr/hal-00789533.

[22] J.-C. Filliâtre and M. Pereira. "A Modular Way to Reason About Iteration." In: *NASA Formal Methods*. Ed. by S. Rayadurgam and O. Tkachuk. Cham: Springer International Publishing, 2016, pp. 322–336. ISBN: 978-3-319-40648-0.

[23] J.-C. Filliâtre, L. Gondelman, and A. Paskevich. "The Spirit of Ghost Code." In: *CAV 2014, Computer Aided Verification - 26th International Conference*. Vienna Summer Logic 2014, Austria, July 2014. URL: https://hal.inria.fr/hal-00873187.

[24] J.-C. Filliâtre, L. Gondelman, and A. Paskevich. "A Pragmatic Type System for Deductive Verification." working paper or preprint. Feb. 2016. URL: https://hal.inria.fr/hal-01256434.

[25] C. Flanagan and S. Qadeer. "Predicate Abstraction for Software Verification." In: *SIGPLAN Not.* 37.1 (Jan. 2002), pp. 191–202. ISSN: 0362-1340. DOI: 10.1145/565816.503291. URL: http://doi.acm.org/10.1145/565816.503291.

[26] G. C. Gannod and B. H. C. Cheng. "Strongest postcondition semantics as the formal basis for reverse engineering." In: *Proceedings of 2nd Working Conference on Reverse Engineering*. 1995, pp. 188–197. DOI: 10.1109/WCRE.1995.514707.

[27] S. Gilbert and N. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services." In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: http://doi.acm.org/10.1145/564585.564601.

[28] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. "'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, 2016, pp. 371–384. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837625. URL: http://doi.acm.org/10.1145/2837614.2837625.

[29] D. Hoang, Y. Moy, A. Wallenburg, and R. Chapman. "SPARK 2014 and GNATprove." In: *International Journal on Software Tools for Technology Transfer* 17.6 (2015), pp. 695–707. ISSN: 1433-2787. DOI: 10.1007/s10009-014-0322-5. URL: https://doi.org/10.1007/s10009-014-0322-5.

[30] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[31] F. Houshmand and M. Lesani. "Hamsaz: Replication Coordination Analysis and Synthesis." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 74:1–74:32. ISSN: 2475-1421. DOI: 10.1145/3290387. URL: http://doi.acm.org/10.1145/3290387.

[32] N. Jeannerod, C. Marché, and R. Treinen. "A Formally Verified Interpreter for a Shell-Like Programming Language." In: Dec. 2017, pp. 1–18. ISBN: 978-3-319-72307-5. DOI: 10.1007/978-3-319-72308-2_1.

[33] G. Kaki, K. Earanky, K. Sivaramakrishnan, and S. Jagannathan. "Safe Replication Through Bounded Concurrency Verification." In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 164:1–164:27. ISSN: 2475-1421. DOI: 10.1145/3276534. URL: http://doi.acm.org/10.1145/3276534.

[34] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. "Frama-C: A Software Analysis Perspective." In: *Form. Asp. Comput.* 27.3 (May 2015), pp. 573–609. ISSN: 0934-5043. DOI: 10.1007/s00165-014-0326-7. URL: http://dx.doi.org/10.1007/s00165-014-0326-7.

[35] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[36] R. Leino. *Efficient Weakest Preconditions*. Tech. rep. MSR-TR-2004-34. 2004, p. 11. URL: https://www.microsoft.com/en-us/research/publication/efficient-weakest-preconditions/.

[37] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. "Making Geo-replicated Systems Fast As Possible, Consistent when Necessary." In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 265–278. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387906.

[38] G. Marcelino, V. Balegas, and C. Ferreira. "Bringing Hybrid Consistency Closer to Programmers." In: *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '17. Belgrade, Serbia: ACM, 2017, 6:1–6:4. ISBN: 978-1-4503-4933-8. DOI: 10.1145/3064889.3064896. URL: http://doi.acm.org/10.1145/3064889.3064896.

[39] Y. Moy. "The Jessie plugin for deductive verification in Frama-C." In: ().

[40] S. S. Nair and M. Shapiro. *Improving the "Correct Eventual Consistency" Tool*. Research Report RR-9191. Sorbonne Université, July 2018. URL: https://hal.inria.fr/hal-01832888.

[41] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro. "The CISE Tool: Proving Weakly-consistent Applications Correct." In: *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*. PaPoC '16. London, United Kingdom: ACM, 2016, 2:1–2:3. ISBN: 978-1-4503-4296-4. DOI: 10.1145/2911151.2911160. URL: http://doi.acm.org/10.1145/2911151.2911160.

[42] M. J. Parreira Pereira. "Tools and Techniques for the Verification of Modular Stateful Code." PhD Thesis. Université Paris-Saclay, Dec. 2018. URL: https://tel.archives-ouvertes.fr/tel-01980343.

[43] J. C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74. ISBN: 0-7695-1483-9. URL: http://dl.acm.org/citation.cfm?id=645683.664578.

[44] R. Rieu-Helft, C. Marché, and G. Melquiond. "How to Get an Efficient yet Verified Arbitrary-Precision Integer Library." In: *9th Working Conference on Verified Software: Theories, Tools, and Experiments*. Vol. 10712. Lecture Notes in Computer Science. Heidelberg, Germany, July 2017, pp. 84–101.

[45] K Rustan, M Leino, J. B. Saxe, and R. Stata. "Checking Java programs via guarded commands." In: 1743 (Aug. 1999).

[46] M. Shapiro and P. Sutra. "Database Consistency Models." In: *CoRR* abs/1804.00914 (2018). arXiv: 1804.00914. URL: http://arxiv.org/abs/1804.00914.

[47] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-Free Replicated Data Types." In: *Stabilization, Safety, and Security of Distributed Systems.* Ed. by X. Défago, F. Petit, and V. Villain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400. ISBN: 978-3-642-24550-3.

[48] J. Signoles. "From Static Analysis to Runtime Verification with Frama-C and E-ACSL." Habilitation Thesis. Doctoral dissertation. Université Paris-Sud, Orsay, France, July 2018. URL: publis/hdr.pdf.

[49] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. "Declarative Programming over Eventually Consistent Data Stores." In: *SIGPLAN Not.* 50.6 (June 2015), pp. 413–424. ISSN: 0362-1340. DOI: 10.1145/2813885.2737981. URL: http://doi.acm.org/10.1145/2813885.2737981.

[50] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. "Transactional Storage for Geo-replicated Systems." In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.* SOSP '11. Cascais, Portugal: ACM, 2011, pp. 385–400. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043592. URL: http://doi.acm.org/10.1145/2043556.2043592.

[51] R. W. Floyd. "Assigning Meanings to Programs." In: *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics* 19 (Jan. 1967). DOI: 10.1090/psapm/019/0235771.

[52] P. Zeller. "Testing Properties of Weakly Consistent Programs with Repliss." In: *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data.* PaPoC '17. Belgrade, Serbia: ACM, 2017, 3:1–3:5. ISBN: 978-1-4503-4933-8. DOI: 10.1145/3064889.3064893. URL: http://doi.acm.org/10.1145/3064889.3064893.

CRDTs library

A.1 Disable Once Flag

It is a flag that once it is disabled, it can never be enabled afterwards. This data structure offers the following operations:

- **initializeFlag ():** Initialises the flag to an enabled state and resets the `was_disabled` variable.
- **enable ():** Enables the flag iff the flag has not been disabled yet.
- **disable ():** Disables the flag.
- **status ():** Returns the status of the flag, either enabled or disabled.
- **equals (doflag):** Checks if two Disable Once Flags are equal.

The specification of this CRDT can be seen in Figure A.1.

A.2 Enable Once Flag

This is a flag similar to the previous one but in this case once it is enabled, it can never be disabled afterwards. This data structure offers the following operations:

- **initializeFlag():** Initialises the flag to a disabled state and resets the `was_enabled` variable.
- **enable():** Enables the flag.
- **disable():** Disables the flag iff the flag has not been enabled yet.

```
1   type ref_bool = { mutable b: bool }
2   val flag : ref_bool
3   val was_disabled : ref_bool
4
5   val initializeFlag () : unit
6     writes  { flag, was_disabled }
7     ensures { flag.b }
8     ensures { not was_disabled.b }
9
10  val enable () : unit
11    requires { not was_disabled.b }
12    writes   { flag }
13    ensures  { flag.b }
14
15  val disable () : unit
16    writes  { flag, was_disabled }
17    ensures { not flag.b }
18    ensures { was_disabled.b }
19
20  val status () : bool
21    ensures { result ↔ flag.b }
22
23  val equals (two : ref_bool) : bool
24    ensures { result ↔ flag.b = two.b }
```

Figure A.1: Disable Once flag in Why3.

- **status():** Returns the status of the flag, either enabled or disabled.
- **equals(eoFlag):** Checks if two Enable Once Flags are equal.

The specification of this CRDT can be seen in Figure A.2.

A.3 Remove-wins Set

It is a set where a precedence is given to the remove operation. This set is able to resolve the commutativity issues related to the addition and removal of elements from a set. This data structure offers the following operations:

- **add_element(elt):** Adds the element `elt` to the `remove_wins_add` set.
- **remove_element(elt):** Adds the element `elt`, if it exists in the set, to the `remove_wins_removes` set. Once this happens, `elt` will never be considered in the set even if it is added later.
- **in_set(elt,rwset):** Checks if the element `elt` is inserted in the Remove-wins set `rwset`.
- **empty_set():** Returns an empty Remove-wins set.

```
type ref_bool = { mutable b: bool }
val flag : ref_bool
val was_enabled : ref_bool

val initializeFlag () : unit
  writes  { flag, was_enabled }
  ensures { not flag.b }
  ensures { was_enabled.b }

val enable () : unit
  writes   { flag, was_enabled }
  ensures  { flag.b }

val disable () : unit
  requires { not was_enabled.b }
  writes  { flag }
  ensures { not flag.b }

val status () : bool
  ensures { result ↔ flag.b }

val equals (two : ref_bool) : bool
  ensures { result ↔ flag.b = two.b }
```

Figure A.2: Enable Once flag in Why3.

- **equals(rwset1, rwset2):** Checks if two Remove-wins sets are equal.

The specification of this CRDT can be seen in Figure A.3.

A.4 Add-wins Set

It is a similar set to the previous one but in this case the precedence is given to the addition operation. This set is able to resolve the commutativity issues related to the addition and removal of elements from a set. This data structure offers the following operations:

- **add_element(elt):** Adds the element `elt` to the `remove_wins_add` set. Once this happens `elt` is considered to be in the set even if it is removed.
- **remove_element(elt):** Adds the element `elt`, if it exists in the set, to the `remove_wins_removes` set.
- **in_set(elt,rwset):** Checks if the element `elt` is inserted in the Add-wins set `awset`.
- **empty_set():** Returns an empty Add-wins set.
- **equals(awset1, awset2):** Checks if two Add-wins sets are equal.

The specification of this CRDT can be seen in Figure A.4.

```
type remove_wins_set 'a = {
  mutable remove_wins_add: fset 'a;
  mutable remove_wins_removes: fset 'a;
}

let ghost predicate equal (s1 s2: remove_wins_set 'a) =
  s1.remove_wins_add     == s2.remove_wins_add &&
  s1.remove_wins_removes == s2.remove_wins_removes

val empty_set () : remove_wins_set 'a
  ensures { is_empty result.remove_wins_add }
  ensures { is_empty result.remove_wins_removes }

predicate in_set (elt: 'a) (s: remove_wins_set 'a) =
  mem elt s.remove_wins_add && not (mem elt s.remove_wins_removes)

val add_element (elt: 'a) (s: remove_wins_set 'a) : unit
  writes  { s.remove_wins_add }
  ensures { s.remove_wins_add = add elt (old s).remove_wins_add }

val remove_element (elt: 'a) (s: remove_wins_set 'a) : unit
  writes  { s.remove_wins_removes }
  ensures { s.remove_wins_removes = add elt (old s).remove_wins_removes }

```

Figure A.3: Remove-wins Set CRDT in Why3.

```
type add_wins_set 'a = {
  mutable add_wins_add: fset 'a;
  mutable add_wins_removes: fset 'a;
}

let ghost predicate equal (s1 s2: add_wins_set 'a) =
  s1.add_wins_add     == s2.add_wins_add &&
  s1.add_wins_removes == s2.add_wins_removes

val empty_set () : add_wins_set 'a
  ensures { is_empty result.add_wins_add }
  ensures { is_empty result.add_wins_removes }

predicate in_set (elt: 'a) (s: add_wins_set 'a) =
  mem elt s.add_wins_add

val add_element (elt: 'a) (s: add_wins_set 'a) : unit
  writes  { s.add_wins_add }
  ensures { s.add_wins_add = add elt (old s).add_wins_add }

val remove_element (elt: 'a) (s: add_wins_set 'a) : unit
  writes  { s.add_wins_removes }
  ensures { s.add_wins_removes = add elt (old s).add_wins_removes }

```

Figure A.4: Add-wins Set CRDT in Why3.