UPMC
SORBONNE UNIVERSITÉS

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Mahsa NAJAFZADEH

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

The Analysis and Co-design of Weakly-Consistent Applications

soutenue le

devant le jury composé de :

M. Marc SHAPIRO	Directeur de thèse
M. Vivien QUEMA	Rapporteur
Mme.Carla FERREIRA	Rapporteur
M. Philippe PUCHERAL	Examinateur
Mme. Beatrice BERARD	Examinateur
M. Pascal MOLLI	Examinateur
M. Pierre-Evariste DAGAND	Examinateur

Abstract

Distributed databases take advantage of replication to bring data close to the client, and to always be available. The primary challenge for such databases is to ensure consistency. The inherent trade-off between consistency, performance, and availability represents a fundamental issue in design of the replicated database serving applications with integrity rules. Recent research provide hybrid consistency models that allow the database supports asynchronous updates by default, but synchronisation is available upon request. To help programmers exploit the hybrid consistency model, we propose a set of useful patterns, proof rules, and tool for proving integrity invariants of applications. The main goal of this thesis is to co-design the application and the associated consistency in order to ensure application invariants with minimal consistency requirements.

In the first part, we study a sound proof rule that enables programmers to check whether the operations of a given application semantics maintain the application invariants under a given amount of parallelism. We have developed a SMT-based tool that automates this proof, and verified several example applications using the tool. A successful analysis proves that a given program will maintain its integrity invariants. If not, the tool provides a counter-example, which the program developer can leverage to adjust the program design, either by weakening application semantics, and/or by adding concurrency control, in order to disallow toxic concurrency.

In the second part, we apply the above methodology to the design of a replicated file system. The main invariant is that the directory structure forms a tree. We study three alternative semantics for the file system. Each exposes a different amount of parallelism, and different anomalies. Using our tool-assisted rules, we check whether a specific file system semantics maintains the tree invariant, and derive an appropriate consistency protocol. Our co-design approach is able to remove coordination for the most common operations, while retaining a semantics reasonably similar to POSIX.

In the third part of this thesis, we present three classes of invariants: equivalence, partial order, and single-item generic. Each places some constraints over the state. Each of these classes maps to a different storage-layer consistency property: respectively, atomicity, causal ordering, or total ordering. Given a class of invariant, we introduce a set of common patterns where synchronisation is not necessary i.e., nothing bad happens for any arbitrary order of operation executions. We also identify patterns where synchronisation is necessary, but can be relaxed in a

disciplined manner.

Acknowledgement

I would like to express my sincere gratitude to my adviser, Marc Shapiro, for his guidance and enthusiastic encouragement. This work would not be possible without his support, patience, and continuous advice.

I am really thankful to all my committee members, especially Vivien Quéma and Carla Ferreira, for accepting to be in my jury without hesitation, and offering me their precious time and comments.

I am very grateful to Alexety Gostman, Carla Ferreira, Pierre Evariste Dagand, Nuno Pregusica, and Valter Balegas for all helpful and inspiring conversations. Thank you Alexey Gotsman and Carla Ferreira for our weakly meetings, and all the ideas they contributed with.

Among many others, I would like to particularly thank Tyler Crain, Alejandro Z. Tomsic, Masoud Saeida Ardekani, and Vinh Tao Thanh. Thank you Tyler for all helpful and inspiring conversations. Thank you Lyes Hamidouche, Marjorie Bourna, Gauthier Voron Laure Mille, and Maxime Lorrillere for all translations.

I want thank my best friends Neda Karimopour and Farzaneh Zareie for their care and precious friendship. No matter how much distance exists between us, they are always there when I need them. My gratitude also to Shahin Mahmoodian for her friendship and spiritual support in particular in my last year of PhD in LIP6.

I would like to express the profound gratitude from my deep heart to my beloved parents, my sister, my brother, and my aunt for their love, continuous support and encouragement in every step of my life. I love all of you so much and appreciate everything you have done to get me where I am today.

To my beloved and encouraging parents, Mahmoud and Farrokh

and to my wonderful siblings, Asma and Pouria

Table of Contents

List of Tables **xiii**

List of Figures **xv**

1 Introduction **1**
- 1.1 Contributions . . . 2
  - 1.1.1 The CISE Tool: Proving Weakly-Consistent Applications Correct . . . 2
  - 1.1.2 A Scalable and Verified Design of a POSIX-Like File System . . . 3
  - 1.1.3 Efficiently Implementable Patterns of Invariants . . . 3
- 1.2 Outline of the thesis . . . 4

I Preliminaries **5**

2 Models and Definitions **7**
- 2.1 Model . . . 8
  - 2.1.1 Database Model . . . 8
  - 2.1.2 System Model . . . 8
- 2.2 Application Invariants . . . 8
- 2.3 Operation Executions in a Replicated Databases . . . 9
  - 2.3.1 Program Execution . . . 11
  - 2.3.2 Happened-Before . . . 11
- 2.4 Correctness Criteria for Replicated Databases . . . 12
  - 2.4.1 State Convergence . . . 12
  - 2.4.2 Safety . . . 12
  - 2.4.3 Conflicting Operations . . . 13
- 2.5 Tokens: Concurrency Control Abstraction . . . 13

3 Replicated Data Types **15**
- 3.1 CRDTs . . . 16
  - 3.1.1 Add-Wins Set . . . 17

3.1.2 Remove-Wins Set . . . 18
3.1.3 Map . . . 19

II The CISE Analysis **21**

4 CISE's Proof Obligations **23**
4.1 Motivation . . . 24
4.2 Consistency Models . . . 26
4.2.1 Sequential Consistency . . . 26
4.2.2 Causal Consistency . . . 26
4.2.3 RedBlue Consistency . . . 26
4.2.4 General Case . . . 27
4.3 CISE Analysis . . . 27
4.3.1 Effector Safety . . . 27
4.3.2 Commutativity Analysis . . . 28
4.3.3 Stability Analysis . . . 29

5 The CISE Tool **33**
5.1 Automatic Solver for CISE Analysis . . . 34
5.2 Application Model . . . 34
5.2.1 Database . . . 34
5.2.2 Operations . . . 36
5.2.3 Invariants . . . 36
5.2.4 Tokens . . . 37
5.3 Solver . . . 38
5.4 CISE Tool's Parser . . . 39

6 The CISE Proof Tool's Application **41**
6.1 Application/Consistency Co-design . . . 42
6.2 Bank Application . . . 42
6.3 Counter With Escrow . . . 45
6.4 Courseware Application . . . 47
6.5 Auction Application . . . 51
6.5.1 Database . . . 51
6.5.2 Invariant . . . 53
6.5.3 Operations . . . 54

7 Related Work **57**
7.1 Related Work . . . 58

7.1.1 Consistency Models . . . 58
7.1.2 Reasoning About Consistency in Distributed Systems and Databases . . . 60
7.2 Conclusion . . . 61
7.3 Future Work . . . 62

III Verifying and Co-designing File System Semantics 63

8 A Scalable and Verified Design of a POSIX-Like File System 65
8.1 Motivation . . . 66
8.2 Definitions and Database Model . . . 66
8.3 A Formal Model of a Replicated File System Semantics . . . 68
8.4 Correctness Criteria . . . 71
8.5 Verifying Sequential Correctness of the File System . . . 72
8.6 Replicated File System With Concurrency Control . . . 73
8.7 Fully Asynchronous Replicated File System . . . 75
8.7.1 Name Conflict . . . 78
8.7.2 Remove/Update Conflict . . . 79
8.8 Mostly Asynchronous Replicated File System . . . 82

9 Related Work 87
9.1 Related Work . . . 88
9.1.1 Formal Reasoning about File Systems . . . 88
9.1.1.1 First-Order Logic Reasoning . . . 88
9.1.1.2 Separation Logic Reasoning . . . 89
9.1.2 Conflict Resolution in File systems . . . 90
9.2 Conclusion . . . 91
9.3 Future Work . . . 91

IV Safe Applications on the Cheap with Invariant Patterns 93

10 Efficiently Implementable Patterns of Invariants 95
10.1 Classes of Invariant . . . 97
10.2 Generic Invariants on a Single Item (Gen1) . . . 98
10.2.1 Protocols and Mechanisms for Gen1-Invariants . . . 98
10.2.2 Total-Order . . . 99
10.3 EQ Invariants . . . 100
10.3.1 Protocols and Mechanisms for EQ-Invariants . . . 100
10.3.2 EQ Invariants and Convergence Resolutions . . . 101

10.4 PO Invariants 101
10.4.1 Protocols and Mechanisms for PO-invariants 102
10.5 Composite Invariant Patterns 103
10.5.1 Composing Gen1-Invariants 103
10.5.2 Composing Gen1-Invariant with EQ-Invariant 104
10.5.3 Composing Gen1-Invariant with PO-Invariant 104
10.5.4 Composing EQ-Invariants 104
10.5.5 Composing EQ-Invariant with PO-Invariant 105
10.5.6 Composing PO-Invariants 105
10.6 Consistency Models 106
10.6.1 Strict Serialisability (SSER) 106
10.6.2 Serialisability (SER) 106
10.6.3 Snapshot Isolation (SI) 107
10.6.4 Causal Consistency (CC) 108
10.6.5 Eventual Consistency (EC) 108
10.6.6 Invariant Anomaly Comparison 108
10.7 Conclusion and Future work 109

Bibliography **111**

List of Tables

5.1 Some member functions of Z3 context. 35

6.1 Auction invariants. 50

7.1 A summery of applications verified by CISE analysis 61

8.1 File system commands and their effectors. 70
8.2 Corrected preconditions of file system operations after effector safety analysis. 72
8.3 Combined effect of concurrent operations using different convergence semantics. . . . 76

10.1 Some consistency models and their invariant guarantees. 107

List of Figures

2.1 A simple banking application (incorrect). 9
2.2 A causally-consistent program execution. 11

3.1 Counter-example: set with concurrent add and remove. 16
3.2 An implementation of add-wins set(AWset). 17
3.3 A program execution using an add-wins set. 17
3.4 A implementation of remove-wins set(RWset). 18
3.5 A program execution using a remove-wins set. 18
3.6 An implementation of map (AWmap). 19

4.1 Stability analysis for bank application: counter-example. 29
4.2 Corrected bank account application. 30
4.3 Execution illustrating the unsoundness of the roof rule for non-commutative operations. 31

5.1 CISE stability rule code. 38

6.1 A simple bank application (incorrect). 43
6.2 Corrected bank application differs from Figure 6.1 as follows: improving precondition of operations and using tokens. 44
6.3 Counter With Escrow (incorrect). 45
6.4 Corrected Counter With Escrow differs from Figure 6.3 as follows: using tokens and improving preconditions. 47
6.5 Courseware application (incorrect). 48
6.6 Corrected courseware application differs from Figure 6.5 as follows: using tokens and improving preconditions. 49
6.7 Auction database state 51
6.8 Auction application (incorrect). 52
6.9 Corrected auction application. 55

8.1 Example of a directory tree structure. 67
8.2 A simple file system application (incorrect). 71

8.3 Corrected file system application with mutually exclusive tokens. 73
8.4 Asynchronous file system design using add-wins semantics. 77
8.5 Asynchronous file system design using remove-wins semantics. 78
8.6 Example of concurrent creating directories with the same name. 80
8.7 Counter-example for stability analysis of concurrent moves. 81
8.8 Mostly asynchronous and corrected file system (add-wins). 83
8.9 Counter-example: the parent relation anomaly. 84

10.1 Atomicity and different concurrent set semantics 100

Chapter 1

Introduction

To achieve high availability and responsiveness, many distributed systems rely on *replicated databases* that maintain copies (replicas) of data at multiple servers [33, 70, 93]. A user can access a local replica and perform operations locally, without synchronising with others. This local access avoids the cost of network latency between replicas, and is insensitive to failures, such as replica disconnection [33, 35, 70, 91].

A major challenge in the design of replicated databases is consistency [46]. *Strong consistency* provides a familiar sequential model to application developers, but it requires to execute operations in a global total order at all replicas (synchronous replication). In contrast, a weak consistency model, e.g., *eventual consistency* [35, 81], guarantees immediate response at all times, but it is prone to application bugs, such as state divergence or invariant violation [33].

Designers of a replicated database face a vexing choice between strong consistency, which guarantees a large class of application invariants, but is slow and fragile, and asynchronous replication, which is highly available and responsive, but exposes the programmer to concurrency anomalies. To bypass this conundrum, some research [18, 69, 93, 97] and commercial [8, 21, 73] databases now provide *hybrid* consistency models that allow the programmer to request stronger consistency for certain operations and thereby introduce synchronisation.

Unfortunately, using hybrid consistency models effectively is far from trivial. Requesting stronger consistency in too many places may hurt performance and availability, and requesting it in too few places may violate correctness. Striking the right balance requires the programmer to reason about the application behaviour above the subtle semantics of the consistency model, taking into account which anomalies are disallowed by a particular consistency strengthening and whether disallowing these anomalies is enough to ensure correctness.

This thesis studies the analysis and co-design of the application and the associated consistency in order to ensure application invariants with minimal consistency requirements.

1.1 Contributions

This thesis makes three main contributions:

1. We propose the first static analysis tool for proving integrity invariants of applications using databases with hybrid consistency models.

2. We present a case study of the application of our analysis tool for designing an efficient file system semantics that provides a choice of behaviours similar to POSIX at a reasonable cost.

3. We propose a set of useful patterns, which help application developers to implement common application invariants.

In the remainder of this section, we review our contributions in more detail.

1.1.1 The CISE Tool: Proving Weakly-Consistent Applications Correct

To exploit hybrid consistency models, we propose to apply a polynomial static analysis for causally-consistent distributed databases, called CISE ("Cause I'm Strong Enough") [45]. The CISE analysis checks whether an application is correct under a given synchronisation protocol in the following sense. An application consists of some data items, a set of operations, and invariants over the data items. The semantics of each operation is given by its *effect*, and *preconditions*. This model is formally described in first-order logic. Two operations may execute concurrently, unless this is disallowed by an abstract concurrency control mechanism, called *token*.

The application is *correct* if every concurrent execution of its operations, allowed by its tokens, eventually results in the same state across all replicas, and if every state transition maintains its invariant. The CISE analysis relies on the following three proof rules:

- *effector safety analysis*: verifies that each individual effector maintains the given invariant.
- *commutativity analysis*: verifies that any two concurrent operations commute.
- *stability analysis*: verifies that every operation's precondition is stable under concurrent updates.

The CISE analysis is sound [45]. A successful CISE analysis proves that any execution of the given application, under the given synchronisation protocol, maintains the given invariants. If unsuccessful, the application developer can fix either the application by weakening the updates or the invariants, or its consistency requirements by adding synchronisation, at the expense of availability and performance. After this refinement, the developer repeats the analysis; and so on, until the analysis succeeds.

1.1.2 A Scalable and Verified Design of a POSIX-Like File System

We apply the CISE analysis to co-design a widely-replicated file system. Initially, the file system specification models the POSIX file system API. The main invariant is that the file system structure must be shaped as a *tree*. From the invariant, we derive the sequential precondition for every file system operation. For instance, the precondition of a move operation forbids to move a directory underneath itself.

We extend the sequential semantics to support concurrent users. We study several alternative semantics for the file system. Each exposes a different amount of parallelism, and different anomalies. The strict POSIX specification disallows many concurrent updates, such as writes to the same file, and it therefore requires a lot of synchronisation.

Using the co-design approach, we carefully remove synchronisation on most operations while retaining a semantics reasonably similar to POSIX. The application of the CISE analysis proves that the precondition of a move operation is not stable under concurrent move operation: it follows that no file system can support an unsynchronised move without anomalies, such as loss or duplication. One of our file system models allows all operations to be concurrent except a small fraction of moves. The CISE analysis proves the correctness of this model.

1.1.3 Efficiently Implementable Patterns of Invariants

Our experience of analysing applications shows that many applications share common styles of invariants. Understanding these common patterns could minimise the cost of verifying a large variety of programs. We present the following three major classes of invariants: *Generic1(Gen1)*, *Equivalence(EQ)*, and *Partial-Order(PO)* invariants. A Gen1 invariant specifies a constraint over the value of a single data item. PO and EQ invariants relate the state of different data items.

Given a class of invariant, we first introduce a set of common program execution patterns, which always maintain the invariant, called *safe* patterns.

In the case of concurrent execution of operations, the sequential invariants might be violated. A replicated database may need to provide a certain consistency guarantee in order to ensure the invariant. The EQ invariant requires the "all-or-nothing" concept in ACID transactions (atomicity), ensuring that all replicas see either the effect of all operations over EQ-dependent data items as a unit, or none of them. The PO invariant requires causal delivery, ensuring that all replicas in the system see causally dependent operations in the order of the happened-before relation [64]. The Ge1 invariant requires total ordering, which serialises all operations. These three consistency guarantees are mostly independent. Given an invariant class, we integrate its consistency requirement with operation executions, and propose program patterns, which maintain the invariant.

The invariant patterns can be combined using conjunction ($\wedge$), and disjunction ($\vee$). We discuss the protocol for maintaining different combination of the invariant patterns. Implementing

disjunction needs at least one of the sub-patterns is satisfied, but conjunction entails that the program execution preserves all sub-patterns.

1.2 Outline of the thesis

The thesis is divided into four parts. Part I presents our replicated database model, and then reviews specification model of some useful replicated data types.

The second part presents a set of proof obligations and tool for analysing and co-designing applications using a replicated database. In Chapter 4, we formulate the analysis, and demonstrate its poof rules. In Chapter 6, we describe our SMT-based tool developed to discharge the analysis, and verify several example applications using the tool. We discuss related work, and summarise our analysis results in Chapter 7,

The third part presents the design of a replicated file system. In Chapter 8, we apply the CISE analysis to study three alternative semantics for the file system. We discuss related work in Chapter 9.

The forth part of the thesis identifies three common classes of invariants, and present a set of distinct patterns to preserve each of them.

Part I

Preliminaries

Chapter 2

Models and Definitions

Contents

2.1 Model 8
2.1.1 Database Model 8
2.1.2 System Model 8
2.2 Application Invariants 8
2.3 Operation Executions in a Replicated Databases 9
2.3.1 Program Execution 11
2.3.2 Happened-Before 11
2.4 Correctness Criteria for Replicated Databases 12
2.4.1 State Convergence 12
2.4.2 Safety 12
2.4.3 Conflicting Operations 13
2.5 Tokens: Concurrency Control Abstraction 13

In this chapter, we model our replicated database and its assumptions. We focus on two main correctness properties: *convergence*, and *safety*.

2.1 Model

2.1.1 Database Model

A database contains a set of mutable, replicated data items. Each data item x has a *value* $\in \mathsf{Val}$ at any one time, and a *type* $type(x) \in \mathsf{Type}$ that determines the operations $\mathsf{Op} = \{o, \ldots\}$ that can be invoked on the object. We assume that operation arguments are a part of the operation name. For instance, we can define a bank account with an integer balance. The database stores the balance of the account that clients can read, make deposits to, withdrawals from, and compute interest.

Let State be the set of possible states of the data managed by the database. The initial state is given by σ_{init}.

2.1.2 System Model

We assume a distributed system composed of n processes $P = \{p_1, \ldots, p_n\}$ that communicate through asynchronous and reliable channels. We assume that replicas never fail.

We rely on a *full replication* model where every process or *replica* in the system stores a full copy of the database. For instance, in a bank application, the data for a bank account is replicated at all the bank's branches. The replication model uses a Read-One-Write-All (ROWA) approach [24]. Users interact with the database system through running applications. An application invokes a set of query and update operations on the data stored in the database. A query will be executed against one of the copies, called its *origin* replica. An update operation invoked at some origin replica *eventually* execute at all replicas. There is no *global* notion of state: each replica changes its local database state independently by the (replicated) execution of operations.

A simple query returns the value of objects without changing the database state, but update operations modify the state.

2.2 Application Invariants

An application might have a set of invariants, which *explicitly* stated. An invariants is a predicate over the database state that restricts the correct values that may be observed by users accessing the database, and determines the safe behaviour of the application. An invariant I deals with one or more objects and their allowed states. For instance, the bank application may require an invariant I stating that the account balance must be non-negative:

$$I = \mathsf{balance} \geq 0$$

$$
\begin{aligned}
\mathsf{State} &= \mathbb{N} \\
\sigma_{init} &= 0 \\
I &= (\mathsf{balance} \geq 0) \\
\mathit{Token} &= \emptyset \\
\bowtie &= \emptyset \\
\mathcal{F}_{\mathsf{deposit(amount)}}(\mathsf{balance}) &= (\bot, (\lambda \mathsf{balance}'.\, \mathsf{balance}' + \mathsf{amount}), \emptyset) \\
\mathcal{F}_{\mathsf{interest()}}(\mathsf{balance}) &= (\bot, (\lambda \mathsf{balance}'.\, (1.05 * \mathsf{balance}')), \emptyset) \\
\mathcal{F}_{\mathsf{query()}}(\mathsf{balance}) &= (\mathsf{balance}, \mathsf{skip}, \emptyset) \\
\mathcal{F}_{\mathsf{withdraw(amount)}}(\mathsf{balance}) &= (\bot, (\lambda \mathsf{balance}'.\, \mathsf{balance}' - \mathsf{amount}), \emptyset)
\end{aligned}
$$

Precondition	Operation
amount ≥ 0	deposit(amount)
true	interest()
amount ≥ 0	withdraw(amount)

Figure 2.1: A simple banking application (incorrect).

2.3 Operation Executions in a Replicated Databases

Following Hoare logic [52], an operation $o \in \mathsf{Op}$ can be represented as follows:

$$\{P_o\}\ o_{eff}\ \{Q_o\}$$

where P_o and Q_o denote the *precondition* and *post-condition* of operation o, whose effect is o_{eff}. The precondition is an assertion about the conditions that must hold of the database state while post-condition is an assertion that must to be true after executing the operation. The effect o_{eff} describes the changes done by the operation to the database state. For instance, consider the operation deposit in the bank application, which guarantees that the account balance is increased by the positive amount specified as its argument. This operation can be represented as follows:

$$\{\mathsf{amount} \geq 0\}\ \{\ \mathsf{balance} := \mathsf{balance} + \mathsf{amount}\}\ \{\ \mathsf{balance} = \mathsf{balance} + \mathsf{amount}\}$$

In this section, we provide a generic execution model for operations running on replicated databases based on the formal model proposed by Gotsman et al. [45]. The execution of each operation consists of two phases: *generator* and *deliver*. The generator executes on origin replica, and the deliver phase executes at all replicas. Assume that a user submits an operation $o \in \mathsf{Op}$ at some origin replica. The generator includes first reading the state $\sigma \in \mathsf{State}$, and then mapping the operation o to a *return value* $value \in \mathsf{Val}$, a state *transformation* function, called *effector*, and a set of *tokens*. More precisely, the semantics of operations is defined by a function

$$\mathcal{F} \in \mathsf{Op} \to (\mathsf{State} \to (\mathsf{Val} \times (\mathsf{State} \to \mathsf{State}) \times \mathsf{set}(\mathit{Token}))). \tag{2.1}$$

The function is formulated as follows for some operation o:

$$\forall \sigma \in \mathsf{State}, o \in \mathsf{Op}, \mathcal{F}_o(\sigma) = (\mathcal{F}_o^{\mathsf{val}}(\sigma), \mathcal{F}_o^{\mathit{eff}}(\sigma), \mathcal{F}_o^{\mathsf{tok}}(\sigma)),$$
$$\mathcal{F}_o^{\mathsf{val}}(\sigma) \in \mathsf{Val}$$
$$\mathcal{F}_o^{\mathit{eff}}(\sigma) \in \mathsf{State} \to \mathsf{State}$$
$$\mathcal{F}_o^{\mathit{eff}}(\sigma) \in \mathsf{set}(\mathit{Token})$$

Given a state $\sigma \in \mathsf{State}$ in which an operation $o \in \mathsf{Op}$ executes at its origin replica,

- $\mathcal{F}_o^{\mathsf{val}}(\sigma)$ is the return value of operation o form a set Val. We use $\perp \in \mathsf{Val}$ for operations that return no value.
- $\mathcal{F}_o^{\mathit{eff}}(\sigma)$ is its effector that will be applied by every replica to its state.
- $\mathcal{F}_o^{\mathsf{tok}}(\sigma)$ is the set of tokens that the operation o requires, and used to introduce synchronisation. We will explain tokens later in Section 2.5. Until then, assume that the token set is empty.

Figure 2.1 illustrates the operational semantics of a simple bank account application using our model. A user can read the balance from the local replica, make deposits to and withdrawals from the account, and compute interest, all without communicating with the other replicas. Each operation is associated with a precondition (P_o), a predicate over the state of its origin replica and parameters that determines when the operation can be safely executed (and the $\mathcal{F}$ function defined). A minimal precondition of the deposit(amount) and withdraw(amount) operations is amount ≥ 0. Their effectors add amount to (respectively, subtract it from) the balance if the preconditions are hold over the origin replica's state. Otherwise, they generate skip as their effect where $\mathsf{skip} = (\lambda \mathsf{balance}'.\, \mathsf{balance}')$.

As an example, consider the semantics of operations for withdrawing an amount, and accruing a 5% interest using the interest operation.

$$\mathcal{F}_{\mathsf{withdraw(amount)}}(\mathsf{balance}) = \mathsf{if\ amount} \geq 0 \ \mathsf{then}\ (\perp, (\lambda \mathsf{balance}'.\, \mathsf{balance}' - \mathsf{amount}), \emptyset)$$
$$\mathsf{else}\ (\perp, (\lambda \mathsf{balance}'.\, \mathsf{skip}), \emptyset)$$
$$\mathcal{F}_{\mathsf{interest}}(\mathsf{balance}) = (\perp, (\lambda \mathsf{balance}'.\, (1.05 * \mathsf{balance}'), \emptyset));$$

The interest operation's precondition is true and its effector multiplies the balance by the interest rate. We will see later, the analysis shows that the precondition of withdraw needs to be strengthened, and that this effector of interest is incorrect (not commutative). Note, in Figure 2.1, we assume that operation's precondition holds at the origin replica and ignore the other case.

The generator does not have any side effects; it executes atomically. After preparing the operation, its effector is sent and eventually delivered to all replicas, including the origin replica. Upon receiving the effect, each replica delivers the effector by applying it to its state.

A replica applies an effect *atomically*, i.e., either all effects of the operation take place, or none. For instance, consider a transfer operation. Bob transfers €100 from his account into Alice's

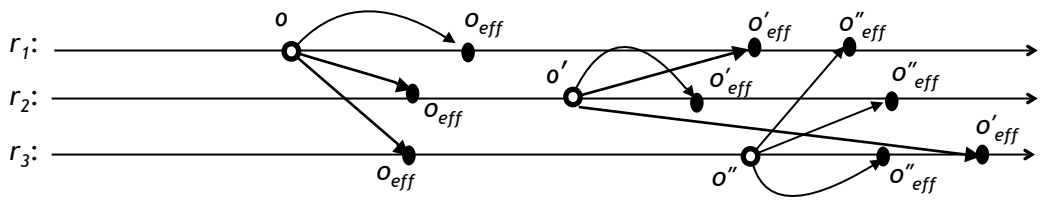


Figure 2.2: A causally-consistent program execution.

account, this includes subtracting €100 from Bob's balance and adding €100 into Alice's balance. Each replica either sees both updates as part of the same operation or none of them. We give the semantics of the transfer operation updating multiple accounts later in Chapter 6.

2.3.1 Program Execution

A program execution H is a tuple $H = (\mathsf{Op}, \mathsf{tokens}, hb)$ such that

- Op is the set of operations appearing in H
- tokens is the set of tokens acquired by the operations Op
- $hb \subseteq \mathsf{Op} \times \mathsf{Op}$, is a strict partial order between operations

where hb is a *happened-before* relation that determines causal dependencies between operations in the program execution.

2.3.2 Happened-Before

We say that operation o' *causally* depends on operation o, noted $o <_{hb} o'$, if and only if:

- Operations o and o' has the same origin replica, in which operation o executes before operation o'.
- The effector of operation o' reads the updates by the effector of operation o over the same object.
- There is an operation o'' such that $o <_{hb} o''$ and $o'' <_{hb} o'$.

Two operations o and o' are *concurrent*, denoted $o \parallel o'$, if neither $o <_{hb} o'$ nor $o' <_{hb} o$. Concurrent operations are causally independent.

Under causal consistency model defined for a shared database [7], all replicas in the system observe the effects of causally-dependent operations in the order in which they happened. However, different replicas may apply effect of causally-independent operations in different orders.

For instance, Alice adds €100 into her account in Europe. She flies to NewYork and withdraws €100 from her account. Causal consistency ensures that all bank branches delivers the effector of deposit before delivering the effector of withdraw.

Figure 2.2 illustrates a causally-consistent program execution in which operations o' and o'' are both causally-dependent on operation o, but mutually concurrent. All three replicas observe the effector of operation o before operations o' and o'', but operations o' and o'' may execute in different orders at the different replicas.

2.4 Correctness Criteria for Replicated Databases

In this thesis, we focus on two main consistency properties that a replicated database must fulfil from the application perspective: 1) *state convergence*: the concurrent execution of operations produce the *same results* at different replicas, and 2) *safety*: any program execution maintains the application invariants, which are typically specified in the form of predicates on the database state.

The replication protocol must forbid operation executions that break these two properties.

2.4.1 State Convergence

A replicated system is said to be *convergent* if all replicas that observed the same set of updates have equivalent state. State convergence is guaranteed if effect of all concurrent operations *commute*, i.e., they are insensitive to the order of execution. Commutativity ensures convergence across replicas despite concurrency [102].

2.4.2 Safety

An application invariant I specifies a predicate over the database state. We say that state σ is *safe* with regard into invariant I iff it satisfies the invariant, denoted by $I(\sigma)$. Program execution is safe if every reachable state is safe.

Definition 2.1 (Safe Update)**.** Given a safe initial state σ_{init} with regard into invariant I, operation o is said to be a *safe update*, iff, after applying the effect of operation o to state σ, invariant I remains true.

To guarantee the application invariant I in any program execution, the replication protocol needs to ensure that:

- Every operation in isolation maintains invariant I.
- Concurrent executions maintain invariant I.

2.4.3 Conflicting Operations

We call a *conflict* execution of two updates o and o' that either do not commute, or whose concurrent execution violates the invariant.

For instance, consider a program execution consisting of the interest and deposit operations, originated at two replicas r_1 and r_2, respectively. The balance is initially €100, balance = 100. The deposit operation adds €20 into the balance, and the interest operation accrues a 5% interest. The operations may execute in different orders at different replicas: replica r_1 first performs interest and then deposit effector, where replica r_2 performs them at opposite order. The result is different; the replicas diverge. Thus, the effector of interest does not commute with deposit effector.

Commutativity alone is not sufficient to verify a program execution. Towards what value the state converges is also important; the state must preserve the application invariant. For instance, in the bank account application with the initial balance of €100, two withdraw(60) operations are conflicting because their concurrent execution would make the balance negative (€-20).

2.5 Tokens: Concurrency Control Abstraction

To address conflicting updates, we introduce a concurrency control abstraction, called *token*. A programmer may associate a set of token $T = \{\tau, \ldots\}$ with an operation to *explicitly* control concurrent accesses over database state. The symmetric *incompatibility relation* between pairs of tokens is noted $\bowtie \subseteq Token \times Token$.

We say a set of tokens T_1 is not compatible with a set of tokens T_2 if there exists token τ_1 in set T_1, and token τ_2 in set T_2 that are not compatible: $\exists \tau_1 \in T_1, \tau_2 \in T_2, \tau_1 \bowtie \tau_2$.

Operations that acquire incompatible tokens according to $\bowtie$ cannot be performed concurrently. For instance, acquiring token τ_1 disallows other replicas to concurrently perform changes that require incompatible tokens.

Definition 2.2 (Incompatibility)**.** For any two operations o and o', if they acquire incompatible tokens according to $\bowtie$, they have to be causally-dependent one way or another:

$$\forall o, o' \in \mathsf{Op}, \mathcal{F}^{\mathsf{tok}}_{o}(\sigma) \bowtie \mathcal{F}^{\mathsf{tok}}_{o'}(\sigma) \implies o <_{hb} o' \vee o' <_{hb} o$$

Definition 2.2 implies causal dependency between operations acquiring incompatible tokens that is one must be causally aware of the other. For instance, if two withdraw operations in the bank application acquire the exclusive token τ with $\tau \bowtie \tau$, they cannot be concurrent.

The token's definition entails that operations to be synchronised [18, 69]. Abstractly, a token is an abstract *lock* [50].

A possible implementation would be the following: before a replica can execute an operation, it acquires the required lock; when the operation returns, it immediately releases the lock.

Chapter 3

Replicated Data Types

Contents

3.1 CRDTs 16
3.1.1 Add-Wins Set 17
3.1.2 Remove-Wins Set 18
3.1.3 Map 19

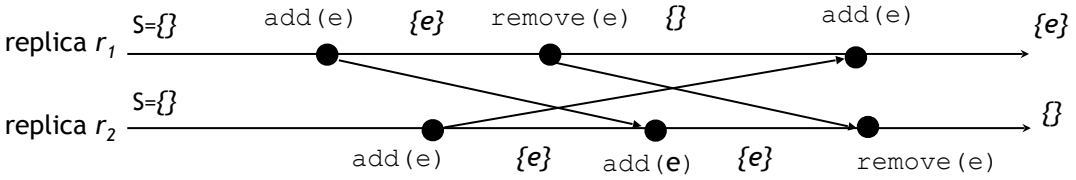


Figure 3.1: Counter-example: set with concurrent add and remove.

This chapter specifies some useful replicated data types: set and map data types. The material in this chapter is useful to following parts of the thesis.

3.1 CRDTs

Commutable updates are desirable. One promising approach is to use *replicated data types* (CRDTs) [91], which encapsulate conflict resolution policies for automatically merging the effects of operations performed concurrently.

CRDTs include many useful data types, such as counters, sets, graphs, and maps. For instance, a CRDT set semantics supports operations to add to or remove an element from the set, and to query.

The sequential semantics of the CRDT set is the classical semantics of a set. Adding an element ensures that it is in the set, whereas removing an element ensures that it is not in the set. Some operations of a sequential set are *commutative*, such as operations on different elements, or *idempotent*, such as adding (or removing) the same element twice. For these commutative or idempotent operations, the concurrent CRDT semantics simply reduces to the sequential set semantics. For instance, users can add elements to or remove elements concurrently from a set when the elements are different; adding element e commutes with deleting element e'.

However, add and remove of the same element is not commutative. Therefore their concurrent execution is problematic. It might result in divergent state across replicas, i.e., the result of concurrent adds and removes depends on the order of execution. To illustrate, consider the simple specification of a replicated set S in Figure 3.1. Set S is initially empty. Replica r_1 adds element e to set S, and then removes element e; its state is again empty. Concurrently, replica r_2 adds the same element e. After both replica r_1 and replica r_2 have applied all operations, the state in replica r_1 and replica r_2 will be different according to sequential specification of the set.

Thus, a CRDT set must specify a commutative semantics for concurrent add and remove of the same element. Many approaches are possible. The most common approaches from an application perspective are called *add-wins* and *remove-wins*. They differ by the result of concurrent add and remove of the same elements. Hereafter, we specify the CRDT set and map data types, using

$$
\begin{aligned}
\text{State} = S &= \text{set}(\mathit{element} \times \mathit{unique} - \mathit{identifier}) \\
\sigma_{init} &= \emptyset \\
\mathcal{F}_{\text{add}(e)}(S) &= (\bot, (\lambda S'.(S' \cup \{(e, \text{unique}(S))\}, \emptyset) \\
\mathcal{F}_{\text{remove}(e)}(S) &= (\bot, (\lambda S'.(S' \setminus \text{lookup}(e,S), \emptyset) \\
\mathcal{F}_{\text{query}}(S) &= (e \mid \exists i, (e,i) \in S, \text{skip}, \emptyset) \\
\text{lookup}(e,S) &= \{(e,i) \mid (e,i) \in S\} \\
\text{unique}(S) &= i \mid i \text{ is } \mathit{unique}
\end{aligned}
$$

Figure 3.2: An implementation of add-wins set(AWset).

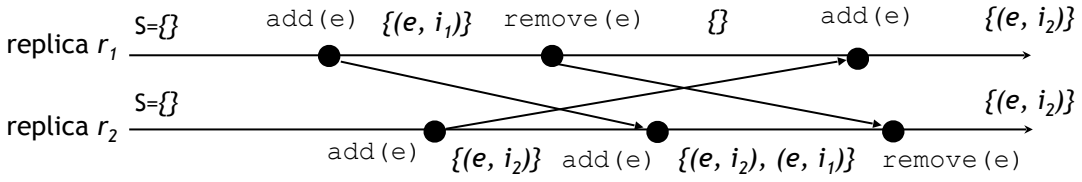


Figure 3.3: A program execution using an add-wins set.

these convergent heuristics.

3.1.1 Add-Wins Set

In the add-wins semantics, sequence of causally-dependent adds and removes should behave according to the sequential specification. When there are concurrent add and remove operations on the same element, the add operation wins and the effects of concurrent remove operations are ignored. Its semantics is defined as:

$$\mathcal{F}_{(\text{AWset})}(\text{query}, H) = \{e \mid \exists o \in H, o = \text{add}(e) \wedge \nexists o' \in H, o' = \text{remove}(e) \wedge o <_{\text{hb}} o'\}.$$

where H is a program execution, and AWset has the same signature as set.

Figure 3.2 presents the basic implementation of an add-wins set by Shapiro et al. [91] using our notation. Each added element is uniquely tagged. The state consists of a set S with pairs $(\mathit{element}, \mathit{unique} - \mathit{identifier})$. A unique() method generates unique identifiers. To add element e to the replicated set, the add operation in the origin replica creates a unique identifier i. The effector is then propagated to all replicas, and adds the pair (e,i) to the set S. Thus, every add is unique.

A lookup(e) method extracts a set of pairs containing element e from the set S. To remove element e, the operation remove computes a set of pairs that contain e in the origin replica using the lookup(e) method, and removes this same set from the set S at all replicas. Since removing

$$
\begin{aligned}
\mathsf{State} = S &= (\mathsf{set} \times \mathsf{set}) \\
S &= (A, T), T = \emptyset, A = \emptyset \\
\mathcal{F}_{\mathsf{add}(e)}(S) &= (\bot, (\lambda S'.\, \mathsf{addToA}(e, \mathsf{lookup}(e, S), S')), \emptyset) \\
\mathcal{F}_{\mathsf{remove}(e)}(S) &= (\bot, (\lambda S'.\, \mathsf{addToT}(e, S')), \emptyset) \\
\mathcal{F}_{\mathsf{query}}(S) &= (\{e \mid (\exists i, (e,i) \in A) \wedge (\nexists i', (e,i') \in T)\}, \mathsf{skip}, \emptyset) \\
\mathsf{addToA}(e, D, (A, T)) &= \text{if } (D = \emptyset) \text{ then } (A \cup \{(e, \mathsf{unique}(S))\}, T) \\
&\qquad \text{else } (A \cup D, T \setminus D) \\
\mathsf{addToT}(e, (A, T)) &= (A, T \cup (e, i)) \\
\mathsf{unique}(S) &= i \mid i \text{ is } \textit{unique} \\
\mathsf{lookup}(e, (A, T)) &= D = \{(e, i) \mid (e, i) \in T\}
\end{aligned}
$$

Figure 3.4: A implementation of remove-wins set(RWset).

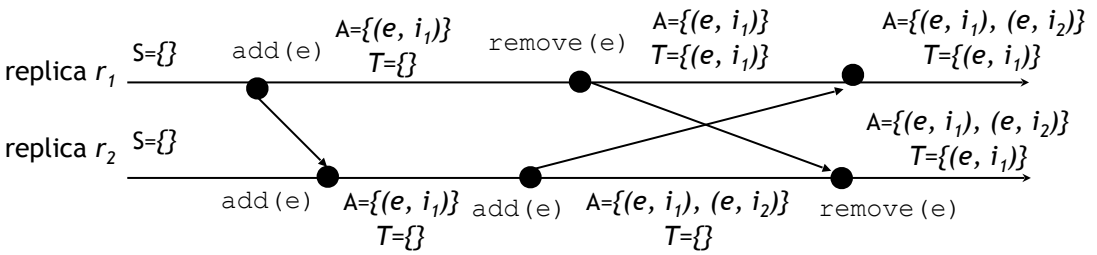


Figure 3.5: A program execution using a remove-wins set.

any disjoint pairs have independent effects, and also removing common pairs have the same effect, concurrent removes commute.

Concurrent adding and removing different elements also commute. When there are concurrent remove and add operations over the same element, the add operation wins, because the new unique tag is not included in the set computed by the remove operation. This behaviour is illustrated in Figure 3.3.

3.1.2 Remove-Wins Set

Similar to the add-wins semantics, the outcome of a sequence of causally-dependent adds and removes is the same as the sequential specification of a set. However, the remove-wins specification has the opposite semantics for handling concurrent add and remove on the same elements: when an element is removed, concurrent adds of the same element are lost. Its semantics is defined as:

$$\mathcal{F}_{(\mathsf{RWset})}(\mathsf{query}, H) = \{e \mid \forall o \in H, o = \mathsf{remove}(e) \implies \exists o' \in H, o' = \mathsf{add}(e) \wedge o <_{\mathsf{hb}} o'\}.$$

where H is a program execution, and RWset has the same signature as set. Element e is in the remove-wins set if all remove(e) operations are covered by add(e) operations, according to the

$$
\begin{aligned}
\mathsf{State} &= M = \mathsf{map}((\mathsf{key} \times \mathit{unique-identifier}) \times \mathsf{value}) \\
\sigma_{init} &= \emptyset \\
\mathcal{F}_{\mathsf{add}(k,v)}(M) &= (\bot, (\lambda M'.\, \mathsf{addToM}(k,v,M', \mathsf{unique}(M))), \emptyset) \\
\mathcal{F}_{\mathsf{remove}(k)}(M) &= (\bot, (\lambda M'.\, M' \setminus \mathsf{lookup}(k,M)), \emptyset) \\
\mathcal{F}_{\mathsf{query}(k)}(M) &= (v \mid \exists i, (k,i,v) \in M, \mathsf{skip}, \emptyset) \\
\mathsf{addToM}(k,v,M',i) &= \mathsf{if}\ (\nexists v', (k,.,v') \in M')\ \mathsf{then}\ (M'[(k,i) \mapsto v]) \\
&\qquad\qquad \mathsf{else}\ (M'[(k,i) \mapsto \mathsf{merge}(v,v')]) \\
\mathsf{lookup}(k,M) &= \{(k,i,v) \mid \exists i,v, (k,i,v) \in M\} \\
\mathsf{unique}(M) &= i \mid i \text{ is } \mathit{unique}
\end{aligned}
$$

Figure 3.6: An implementation of map (AWmap).

happened-before relation.

Figure 3.4 illustrates an implementation of the remove-wins set. A remove-wins se is represented by a pair of disjoint sets: a set of add instances A with pairs (e,i), where e is an element and i is a unique identifier, and a set of remove instances T with the same type of pairs. The query method returns an element e iff there is an add(e) instance and no remove(e) instance.

When an element e is added into the replicated set, the add operation first identifies existing remove instances for element e in the origin replica, and turns them into add instances, and if there is no remove instance, it creates a new add instance for element e and adds to set A. The replication protocol is similar to the add-wins set, it propagates the changes to set A and T to all replicas, which apply the corresponding changes. Thus, concurrent adds of the same element commute because every add is effectively unique. The remove(e) operation removes an element e by adding it to the set T, called the *tombstone* set. Removing an element is allowed only if the element exists, i.e., remove(e) can occur only after add(e). Concurrent removes on the same element commute because they have the same effect. Concurrent add(e) and remove(e') on different elements $e \neq e'$ also commute, and if $e = e'$ the remove wins.

This behaviour is illustrated in Figure 3.5: The set S is initially empty. Replica r_1 adds a to set S using the add(a) operation. Replica r_2 observes the operation and applies its effect. Then replica r_2 adds the same element a using the add(a) operation, concurrently, replica r_1 removes the element a using the remove(a) operation. After exchanging operations, a query returns an empty set in both replicas.

3.1.3 Map

The specification of a CRDT map is based on CRDT sets, as the two set specifications presented earlier can extend to a map. An element is a (key, value) pair. The map semantics supports operations to add to, remove from, and query elements in the map. In an add-wins map, concurrent

$\mathsf{add}(k,v)$ and $\mathsf{remove}(k')$ commute: if $k \neq k'$ they are independent, and if $k = k'$ the add wins. Its semantics is defined as:

$$\mathcal{F}_{(\mathsf{AWmap})}(\mathsf{query}(k), H) = \{v \mid H|_{\mathsf{add}(k,.)} \neq \emptyset \wedge \nexists o' \in H, o' = \mathsf{remove}(k) \wedge o <_{\mathsf{hb}} o' \wedge \\ \forall o = \mathsf{add}(k,v'), o' = \mathsf{add}(k,v''), o \parallel o' \implies v = \mathsf{merge}(v',v'')\}.$$

where H is a program execution, and $H|_o$ is the restriction of operations in H to operations o. The $\mathsf{merge}(v',v'')$ function merges the value of concurrent add operations of the same key.

Figure 3.6 illustrates an implementation of an add-wins map M as follows: The state is a map of pairs $(key, unique-identifier)$ to a value. The $\mathsf{add}(k,v)$ operation creates a new triplet $((k,i),v)$, and adds to map, and if there is a mapping $((k,i'),v')$ in the map, which has created by concurrent add operation, then it uses a merge function to address concurrent adds. We assume that all values in the map are also CRDTs as concurrent adds to the same key can merge. The remove operation computes the set of triplets with the given key, which its effect removes all existing mappings for a given key at all replicas. The precondition is that the corresponding add has been delivered, i.e., the element exists in the origin replica.

Part II

The CISE Analysis

Chapter 4

CISE's Proof Obligations

Contents

4.1 Motivation 24
4.2 Consistency Models 26
4.2.1 Sequential Consistency 26
4.2.2 Causal Consistency 26
4.2.3 RedBlue Consistency 26
4.2.4 General Case 27
4.3 CISE Analysis 27
4.3.1 Effector Safety 27
4.3.2 Commutativity Analysis 28
4.3.3 Stability Analysis 29

In this chapter, we present the CISE analysis, which allows application developers to co-design the application and its consistency model, in order to adjust synchronisation to precisely meet the application's correctness requirements.

Section 4.1 describes the motivation for CISE logic. We outline what problems it addresses, and what is specific to the CISE approach. Section 4.2 defines some well-known consistency models in CISE model using our notations from the previous chapter. In Section 4.3, a formal model of the CISE analysis is presented. The analysis includes three proof obligations. We illustrate each rule by analysing and co-designing the simple bank application example from the previous chapter.

4.1 Motivation

There is growing demand for distributed systems to serve more clients, to minimise response times, and to maximise high availability. To address these requirements, these systems often take advantage of *replicated databases* that replicate data at multiple servers [33, 70, 93]. Thanks to replication, a user can access his own copy of data and perform operations independently from the others. This local access avoids slow long-haul WAN communication between replicas, and improves availability and fault tolerance [33, 35, 70, 91].

A major challenge in the design of replicated databases is consistency [46]. The strongest consistency model (strict serializability) guarantees that a user always accesses the most up-to-date version of data, just like in a centralised database, but requires to execute operations in a global total order at all replicas (synchronous replication) [1]. Many replicated databases choose a weak consistency model, e.g., *eventual consistency* [35, 81], to improve availability (and performance). Under eventual consistency, a replica can execute an update without synchronising a priori with another replicas. The *effect* of the update is propagated to all replicas, and *eventually* every replica performs the same update (asynchronous replication).

Unfortunately, asynchronous replication exposes applications to undesirable concurrency behaviours, which poses a major challenge to the implementation of the applications [33]. Conflicts happen, because concurrent operations may execute in different orders at different replicas. The challenge is then detection of concurrent conflicting updates, and their resolution. For instance, consider a bank account replicated at two different bank branches, where the possible operations are deposit and withdraw, under the application invariant that forbids the account balance to be negative. If the balance is initially €100, and two users, connecting to different bank branches, concurrently withdraw €60, then the balance will become negative(€-20).

Ensuring application invariants entails in the general case that replication protocols introduce synchronisation, which is expensive and not avoidable.

[1] Although the implementation may parallelise some operations when the result remains equivalent to the total order.

The CAP theorem [43] asserts that in the presence of failures, system designers must choose to maintain either availability (and performance) or consistency: both are not possible together. There is no single consistency model best suited for all uses: can provide acceptable application semantics without performance and availability cost [93]. Recent work has begun to provide *hybrid* consistency models that support different consistency guarantees depending on the operation [8, 18, 69, 93, 97]. In this approach, the replicated database supports the asynchronous replication model by default, and adds synchronisation for certain operations when necessary [69]. For instance, to avoid a negative balance in the bank application, a hybrid consistency protocol may execute deposit operations in an asynchronous manner, i.e., users can add to the account in all circumstances, but the bank branches are synchronised before accepting a withdraw operation.

However, using hybrid consistency models effectively in replicated databases is far from trivial. To minimise synchronisation, while ensuring application correctness, the programmer needs to decide under which consistency guarantees each different operation executes. The right decision entails reasoning globally about the application behaviour above the semantics of the consistency model, in order to understand which anomalies can happen and whether a particular consistency strengthening is enough to disallow these anomalies.

To exploit hybrid consistency models, we propose to apply a recently-developed static analysis for causally-consistent distributed databases, called CISE ("Cause I'm Strong Enough") [45]. The CISE analysis checks whether an application is correct under a given synchronisation protocol, in the following sense. An application consists of some data items, a set of operations, some invariants over the data items. The semantics of each operation is given by its *effect*, and by its *preconditions*. This model is formally described in first-order logic. Two operations may execute concurrently, unless this is disallowed by their tokens.

The application is *correct* if every concurrent execution of its operations allowed by its tokens eventually results in the same state across all replicas, and if every state maintains its invariant. The CISE analysis consists of the three following proof rules:

- *effector safety analysis*: verifies that each individual effectors maintains the given invariant.
- commutativity analysis: verifies that any two effectors commute.
- *stability analysis*: verifies that every operation's precondition is stable under concurrent effectors.

The CISE analysis is sound [45]. A successful CISE analysis proves that any execution of the given application, under the given synchronisation protocol, maintains the given invariants. If unsuccessful, the application developer needs to correct this issue either by weakening the updates or the invariants, and/or by adding synchronisation. After this refinement, the developer repeats the analysis; and so on, until the analysis succeeds.

4.2 Consistency Models

We now review, and formally define some well-known consistency models for replicated databases in the CISE model.

4.2.1 Sequential Consistency

Sequential consistency models provide *replication transparency* to the application, i.e., offers applications a single common view of the replicated database. Examples of sequential consistency models include strong consistency, linearisability [51], serialisability [24] or strict serialisability [80]. All replicas agree on the same global ordering of operations. To model the former, every operation is required to acquire a mutual exclusion token:

$$\mathit{Token} = \{\tau\}; \quad \bowtie = \{(\tau,\tau)\}; \quad \forall \sigma \in \mathsf{State}, o \in \mathsf{Op}, \mathcal{F}^{\mathsf{tok}}_o(\sigma) = \{\tau\}$$

Reasoning about replicated databases implementing sequential consistency is easy, because the database behaves *sequentially*. Total order of effectors implies state convergence. If any sequential execution of an application maintains invariant I, any execution of the application under sequential consistency will maintain invariant I.

4.2.2 Causal Consistency

When high performance and availability of update operations are important to the application, an alternative is to use a weaker consistency model that allows concurrent operations. The baseline consistency model of CISE is causal consistency, which does not require any tokens:

$$\mathit{Token} = \emptyset; \quad \bowtie = \emptyset; \quad \forall \sigma \in \mathsf{State}, o \in \mathsf{Op}, \mathcal{F}^{\mathsf{tok}}_o(\sigma) = \emptyset$$

Causal consistency is the strongest achievable model that is available in the presence of network partitions [72]. Under causal consistency, concurrent updates may execute in different orders at different replicas. However, causally-consistent databases guarantee that the effect of an operation is visible only after the effects of operations that it causally depends upon are visible [7].

4.2.3 RedBlue Consistency

RedBlue consistency [69] is a hybrid consistency model that classifies application's operations as *red* and *blue* based on application's invariants: $\mathsf{Op} = \mathsf{Op}_r \uplus \mathsf{Op}_b$. Blue operations commute with all others; they are asynchronous under causal consistency. Red operations must be mutually ordered, requiring system-wide synchronisation; they ensure strong consistency. To express this consistency model, we use a mutual exclusion token τ, where Red operations acquire τ, and blue operations acquire no tokens:

$$\mathit{Token} = \{\tau\}; \quad \bowtie = (\tau,\tau); \quad \forall \sigma \in \mathsf{State}, o \in \mathsf{Op}_b, o' \in \mathsf{Op}_r, \mathcal{F}^{\mathsf{tok}}_o(\sigma) = \emptyset \wedge \mathcal{F}^{\mathsf{tok}}_{o'}(\sigma) = \{\tau\}$$

4.2.4 General Case

We model a general case in which some operations acquire tokens, and their preconditions must be sufficient for invariants. For any two operations o and o', if they acquire incompatible tokens according to $\bowtie$, they have to be mutually ordered,

$$\forall \sigma \in \mathsf{State}, o, o' \in \mathsf{Op}, \mathcal{F}_o^{\mathsf{tok}}(\sigma) \bowtie \mathcal{F}_{o'}^{\mathsf{tok}}(\sigma) \implies o' <_{hb} o \vee o <_{hb} o'$$

4.3 CISE Analysis

The CISE analysis is a static analysis for distributed applications running above a replicated database. A successful CISE analysis proves that any execution of the application under a given synchronisation protocol is correct.

The analysis checks whether an application is correct, in the following sense. An application consists of some data, a set of operations with their preconditions, and some invariants over the data, which are explicitly specified as a set of assertions over database state. This model is described formally in first-order logic. Two operations may execute concurrently, unless this is forbidden by their tokens.

The analysis assumes that the database guarantees causal consistency. The application is correct if every combination of its operations allowed by its tokens ensures that replicas converge, and its invariants are maintained.

The CISE analysis allows to verify any concurrent execution of operations by testing only pairs of concurrent updates, thus it has only polynomial complexity. It consists of three proof obligation rules. The remainder of this chapter describes in detail each CISE's rule.

4.3.1 Effector Safety

The first CISE proof obligation, called *effector safety*, verifies that each effector individually satisfies the desired invariant. It follows that sequential execution of operations also satisfies the invariant. The execution of a set of operations Op is *sequential* if for any two operations o and o' appearing in the execution, either o executes before o' or o' executes before o. Thus, every operation executes to completion before the next operation begins.

An operation has a sufficient precondition if its execution guarantees the correctness [29, 76]. We formulate effector safety proof obligation as follows:

$$\forall o \in \mathsf{Op}, I \in \mathsf{Inv}, \sigma \in \mathsf{State}, I(\sigma) \wedge P_o(\sigma) \implies I(\mathcal{F}_o^{eff}(\sigma)) \tag{4.1}$$

In other words, it must be true that if the initial state σ satisfies invariant I and the operation's precondition P_o, then applying the effector of operation o to state σ maintains the invariant I.

Let's try out effector safety analysis to the simple (incorrect) bank account application presented in Figure 2.1. The database state is a single account balance balance. The integrity invariant requires the balance to be non-negative. The effector for deposit adds some amount to the balance, similarly; the effector for withdraw subtracts some amount from the balance. The operations both have the precondition that the amount must be positive. Applying the effector safety proves that the deposit operation is safe. However, the withdraw effector may violate the non-negatively invariant. This means that its precondition is not strong enough.

We fix the issue by strengthening the precondition of withdraw operation: the amount debited must be positive and less than the current balance. Then, we again perform the effector safety to verify that the precondition is indeed sufficient.

4.3.2 Commutativity Analysis

Concurrent effectors may execute in different orders at different replicas. Two effectors are said to *commute* with one another, if executing them in any mutual order yields the same result, whatever the initial state. For instance, increments to a shared counter x, `inc()`, commute with one another: two effectors `inc(x, 1)` and `inc(x, 2)` can safely be ordered by first executing `inc(x, 1)`, and then `inc(x, 2)` at one replica where another replica performs them at opposite order, in both cases the end result is to increment x by 3.

The second CISE proof obligation, called *commutativity analysis*, is to check that effect of concurrent operations commute, in order to prove that replicas always converge to the same state [61, 102].

$$\forall \sigma, \sigma' \in \mathsf{State}, o, o' \in \mathsf{Op}, (\mathcal{F}_o^{\mathsf{tok}}(\sigma) \bowtie \mathcal{F}_{o'}^{\mathsf{tok}}(\sigma')) \vee (\mathcal{F}_o^{eff}(\sigma).\mathcal{F}_{o'}^{eff}(\sigma') = \mathcal{F}_{o'}^{eff}(\sigma').\mathcal{F}_o^{eff}(\sigma)). \tag{4.2}$$

Informally, the rule is to check all possible concurrent pairs of operations, and to verify that their effect are commutative. Since operations acquiring incompatible tokens are mutually ordered across all replicas, we only require commutativity for operations that do not acquire incompatible tokens, i.e., they can be concurrent.

Recalling the bank operations in Figure 2.1, we check the commutativity rule for all possible concurrent pairs and for all possible initial states and all arguments: deposit ∥ deposit, deposit ∥ withdraw, withdraw ∥ withdraw, interest ∥ interest, deposit ∥ interest, and withdraw ∥ interest. Given the effectors for deposit and withdraw, predictably, applying the commutativity analysis proves that deposit and withdraw operations are commutative, because addition and subtraction commute. However, commutativity is not guaranteed for arbitrary operations. For instance, the interest operation may not commute with other bank operations. Consider a program execution consisting of operations interest and deposit, originating at replicas r_1 and r_2, respectably. The deposit operation adds €20 into the balance. The balance is initially €100. Replica r_1 first performs interest and then deposit(20), where replica r_2 performs them at opposite order. Since Figure 2.1

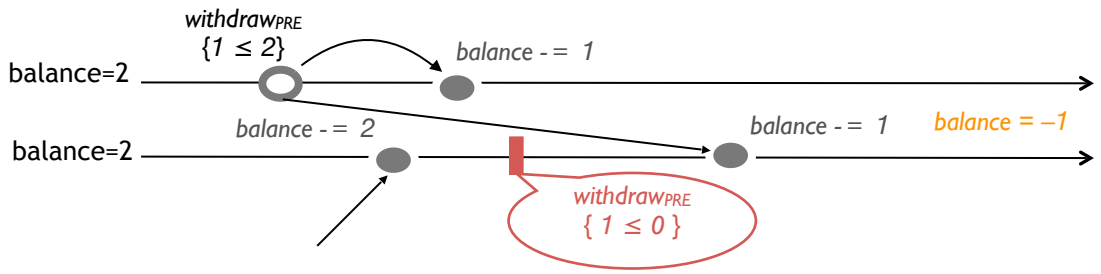


Figure 4.1: Stability analysis for bank application: counter-example.

defines accruing a 5% interest as multiplying the local balance by 1.05, replica r_1 computes 125, whereas replica r_2 computes 126, and hence they diverge.

$$\mathcal{F}^{eff}_{\text{interest}-\text{buggy}}(\text{balance}).\mathcal{F}^{eff}_{\text{deposit}(20)}(\text{balance}) \neq \mathcal{F}^{eff}_{\text{deposit}(20)}(\text{balance}).\mathcal{F}^{eff}_{\text{interest}}(\text{balance})$$
$$100+5+20 \neq 100+20+6$$

This problem can be fixed by implementing the interest operation by computing the amount of interest at the origin replica, and the effector adds that amount to the local balance.

$$\mathcal{F}_{\text{interest}}(\text{balance}) = (\bot, (\lambda \text{balance}'.\ \text{balance}' + 0.05 * \text{balance})), \emptyset). \tag{4.3}$$

Now, with this new semantics, the interest operation commutes with deposit operation. Independent of the order of execution, any concurrent execution of interest and deposit generates the same result.

$$\mathcal{F}^{eff}_{\text{interest}-\text{corrected}}(\text{balance}).\mathcal{F}^{eff}_{\text{deposit}(20)}(\text{balance}')) = \mathcal{F}^{eff}_{\text{deposit}(20)}(\text{balance}).\mathcal{F}^{eff}_{\text{interest}}(\text{balance})$$
$$100+5+20 = 100+20+5$$

The application of commutativity analysis proves that concurrent executions of the deposit and withdraw in Figure 2.1, and the interest defined by equation 4.3 always commute.

4.3.3 Stability Analysis

By applying the effector safety analysis, we verified that in any sequential execution, the invariant I holds. However, concurrently executing operations may violate the invariant. Now, consider the following verification problem: given a set of tokens tokens and their conflict relations captured by $\bowtie$, prove that any possible execution of operations Op maintains integrity invariant I over database states.

The third rule of CISE analysis, called *stability analysis*, is to check if concurrent executions maintain the invariant I. To do this, the stability analysis checks that if operation's precondition is *stable* under concurrent effectors.

The insight behind this proof obligation is that if precondition of an operation remains true after applying concurrent effectors then we know (by the sequential analysis) that applying

$$\begin{aligned}
\mathsf{State} &= \mathsf{balance} \\
\sigma_{init} &= \emptyset \\
\mathit{Token} &= \{\tau\} \\
\bowtie &= \{(\tau, \tau)\} \\
\mathcal{F}_{\mathsf{deposit(amount)}}(\mathsf{balance}) &= (\bot, (\lambda \mathsf{balance}'.\, \mathsf{balance} + \mathsf{amount}), \emptyset) \\
\mathcal{F}_{\mathsf{interest()}}(\mathsf{balance}) &= (\bot, (\lambda \mathsf{balance}'.\, (1.05 * \mathsf{balance}')), \emptyset) \\
\mathcal{F}_{\mathsf{query()}}(\mathsf{balance}) &= (\mathsf{balance}, \mathsf{skip}, \emptyset) \\
\mathcal{F}_{\mathsf{withdraw(amount)}}(\mathsf{balance}) &= (\bot, (\lambda \mathsf{balance}'.\, \mathsf{balance} - \mathsf{amount}), \{\tau\})
\end{aligned}$$

Precondition	Effector
$\mathsf{amount} \geq 0$	$\mathcal{F}_{\mathsf{deposit(amount)}}(\mathsf{balance})$
true	$\mathcal{F}_{\mathsf{interest()}}(\mathsf{balance})$
$\mathsf{amount} \geq 0 \wedge \mathsf{balance} \geq \mathsf{amount}$	$\mathcal{F}_{\mathsf{withdraw(amount)}}(\mathsf{balance})$

Figure 4.2: Corrected bank account application.

effector of the operation is safe, i.e., the invariant remains true.

$$\forall \sigma \in \mathsf{State}, I \in \mathsf{Inv}, o, o' \in \mathsf{Op}, (\mathcal{F}_o^{\mathsf{tok}}(\sigma) \bowtie \mathcal{F}_{o'}^{\mathsf{tok}}(\sigma)) \vee (I(\sigma) \wedge P_o(\sigma) \implies P_o(\mathcal{F}_{o'}^{\mathit{eff}}(\sigma))). \tag{4.4}$$

In other words, if precondition of operation o is *stable* under all concurrent changes allowed, then the invariant is maintained, assuming that the sequential analysis and stability analysis are also verified. Since operations acquiring incompatible tokens are causally dependent, i.e., they cannot be concurrent, we do not need to check the stability analysis for them.

Assume that operation o is submitted at some origin replica r_1. The initial database state σ in replicas r_1 verifies invariant I. The effector of the operation o is generated against state σ, and propagated into all replicas. Upon receiving the effector o, another replica r_2 maybe in a different state σ', due to the effect of operation o' executed at r_2 concurrently with o. The stability analysis checks if o's precondition is true over state σ'. If so, this verification succeeds.

We now use our proof rule to check the stability condition for the bank operations in Figure 2.1. All bank operations except the withdraw pass the stability analysis. We explain the result for the withdraw operation. Applying the stability analysis shows that the precondition of the withdraw operation is stable under the concurrent execution of the deposit and interest operations for all possible initial states and all arguments, but it is not stable under the concurrent execution of another withdraw with the same or different arguments. Figure 4.1 illustrates the counter-example. The balance is initially ∈2. The precondition to withdraw(1) is OK. However, a concurrent withdraw(2) makes the balance zero, thus violating the precondition of withdraw(1). If we were to continue and apply the effect of the first withdrawal operation, the balance will become negative, thus violating the invariant.

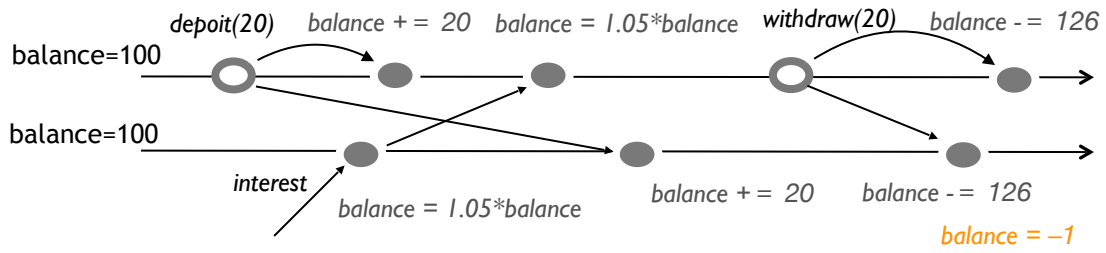


Figure 4.3: Execution illustrating the unsoundness of the roof rule for non-commutative operations.

To fix the problem, different alternatives are possible. If freedom from synchronisation is important, the only alternative is to weaken the invariant e.g., remove the non-negative invariant. The traditional approach is to add some concurrency control in order to disallow the concurrent execution of conflicting operations.

We choose the latter approach for the bank application example. We add an exclusive token τ that disallows any withdraw operation to be concurrent with any other withdraw. Thus, we co-design the bank application by adding the corresponding token to the withdraw operation. Figure 4.2 presents the correct specification of bank account application, which passes all three CISE rules.

Thus, programmer can reason about invariants in the context of different consistency models. Gotsman et al. in [45] have formally proved the soundness of stability analysis assuming *causality* and *commutativity*.

The rule would be unsound over a consistency model that does not guarantee causality. For instance, consider the following scenario in the bank application: $Alice$ has €100 in her account. She deposits €50 and some time later she were to withdraws €120. Then, she reads her account. If the query sees the withdrawal, violating causality, it returns €-20; violating the invariant.

The stability rule is also unsound if effect of operations do not commute. For instance, if we used the non-commutative semantics of interest operation in Figure 2.1, there are program executions that violate the invariant. Consider a shared account, accessed by Alice and Bob, with initial balance €100: Bob computes a 5% interest. Concurrently, Alice deposits €20 into the account. After receiving both updates, Alice withdraws €126. This propagates to Bob's replica. Applying the withdrawal in Bob's replica leaves the balance negative, thus, violating the invariant. The reason is that the effect of withdraw is generated in Alice's replica, which orders deposit(20) before interest, whereas the state in Bob's replica follows the opposite order, resulting in a smaller balance. The execution in Figure 4.3 illustrates the scenario.

The CISE analysis stability rule constitutes of a form of rely-guarantee (RG) reasoning [56]. RG provides a well-known method for verification of concurrent programs using a pair of rely R and guarantee G conditions. The rely condition R specifies the state transitions performed

concurrently by the environment. The guarantee G specifies the state transitions made by the program itself. To prove that the invariant I will always hold, we can introduce a guarantee relation $G(\tau)$ that describes all possible state changes that a concurrent operation acquiring token τ can cause at any replica. We also have a guarantee relation G_0, describing the possible state changes that can be performed by a concurrent operation without acquiring any tokens. Assume that the initial state σ verifies the invariant I, and the guarantees preserve the invariant I, i.e., the changes allowed by G_0 and the changes allowed by $G(\tau)$ over state σ end up in a state in which the invariant I also holds. For each operation $o \in \mathsf{Op}$, separately, we need to prove that the precondition of operation o is stable under changes allowed by G_0 and the changes that other operations whose tokens do not conflict with any of the tokens acquired by the operation o, i.e., $G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp})$. For instance, consider the withdraw operation in the bank application, o = withdraw(balance). The guarantee G_0 allows increasing a non-negative balance, and computing interest. Since $\mathcal{F}_o^{\mathsf{tok}}(\sigma) = \{\tau\}$, we have that $(\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp} = \emptyset$. Applying the rule verifies that o's precondition is stable under changes allowed by the guarantees.

Chapter 5

The CISE Tool

Contents

5.1 Automatic Solver for CISE Analysis 34
5.2 Application Model 34
5.2.1 Database 34
5.2.2 Operations 36
5.2.3 Invariants 36
5.2.4 Tokens 37
5.3 Solver 38
5.4 CISE Tool's Parser 39

In this chapter, we describe our SMT-based tool, developed to automate the CISE analysis.

5.1 Automatic Solver for CISE Analysis

We have developed a tool that automates the CISE analysis by discharging the proof obligations on Satisfiability Modulo Theories (SMT) queries.

A SMT solver checks the satisfiability of first-order formulas with respect to some logical rules. Several SMT solvers have been developed in academia and industry. We built the CISE tool on the Z3 SMT solver [1], developed by Microsoft Research for the verification and analysis of software applications.

Our CISE tool is currently implemented as a few hundred lines of Java code that interface with the Z3 engine via the library provided for Java.

Z3 has a number of built-in functions that operate on Booleans, integers, and more complex data types. We expose these functions directly in our tool development. The main interaction with Z3 happens via Context. A Context manages all Z3 objects, and its global configuration options. After a context *ctx* is created, the configuration cannot be changed.

To prove some claim, the tool negates the claim, and looks for a model satisfying the negation. If found, it constructs a counter-example to the claim.

5.2 Application Model

The main CISE tool interface is application interface. An application is described by its operational semantics (preconditions and effects), its invariants, and its tokens.

5.2.1 Database

An application model starts with a declaration of its database. Each object in the database has a type (aka sort). The underlying Z3 supports primitive data types: *Int*, *Set*, and *Bool*. Every type defines a set of values that an object can have and also operations can be invoked on the object. For instance, the Z3 built-in integer type assigns integer values to data items, and support arithmetic operations such as addition and subtraction.

To interact with Z3 through Java, we need a Context object. Variables and numerals are modelled as expression *Expr* objects. We get objects using member functions in the Context object. For instance, a Z3 context has functions of the form *mksort* to get a type, and function *mkConst* to create an object. Consider, a simple bank account application. Its database stores balance of the account. The balance is integer. In the CISE tool, definition of the balance variable looks like this:

```
IntExpr balance = ctx.mkConst("balance", ctx.mkIntSort());
```

Functions	Definition
mkIntSort()	Create a new integer sort.
mkAdd (ArithExpr... t)	Create an expression representing t[0] + t[1] + ...
mkSub (ArithExpr... t)	Create an expression representing t[0] - t[1]- ...
mkEq(Expr x, Expr y)	Creates the equality x=y.
mkGt (ArithExpr t1, ArithExpr t2)	Create an expression representing $t1 >= t2$
mkSetSort(sort ty)	Create a set of type ty.
mkSetAdd(ArrayExpr set, Expr element)	Add an element to the set.
mkSetMembership(Expr element, ArrayExpr set)	Check for set membership.
mkSetDel(ArrayExpr set, Expr element)	Remove an element from the set.

Table 5.1: Some member functions of Z3 context.

where function *mkConst* of context *ctx* creates an integer variable in Z3 named *balance*, and function *mkIntSort* gets an integer type.

Table 5.1 illustrates a list of common member functions in the Context object for integer and set data types.

Applications may need to declare new data types and relations. A convenient way for specifying new types and relations is using Z3 *sorts* and *tuples*. Each tuple is a finite list of elements. Each element has a name and a type.

For instance, in a file system application, one may define following data types.

```
// A file in the file system
Sort File = ctx.mkUninterpretedSort(ctx.mkSymbol("File"));

// A directory in the file system
Sort Dir = ctx.mkUninterpretedSort(ctx.mkSymbol("Dir"));

// A parent relation in the file system
TupleSort Parent = ctx.mkTupleSort(ctx.mkSymbol("mk_tuple"), // name of tuple
        new Symbol[]{ctx.mkSymbol("parent"), ctx.mkSymbol("child")}, // the name of elements
        new Sort[]{Dir, Node}; // types of elemen

// A set of directories
Expr Dir_set = ctx.mkConst("Dir_set", ctx.mkSetSort(Dir));
```

This example creates two data types: *File* and *Dir*, and a parent relation *Parent*. Function *mkUninterpretedSort* creates an uninterpreted sort. Function *mkTupleSort* creates a tuple. It has three arguments: the name of tuple, the name of its elements, and their type. Using function *mkSetSort*(*Dir*), we create a set of directories.

5.2.2 Operations

An application consists of a set of operations. Each operations is specified by a precondition and an effect.

The precondition of each operation is expressed using a Z3 boolean expression (BoolExpr). Z3 Java API supports the usual Boolean operators *and*, *or*, *xor*, *not*, *implication*, and *ite* (if-then-else).

For instance, in a bank application, the precondition for a withdraw operation is that balance is greater than the debited amount. This is modelled as follows:

```
public BoolExpr precondtion(Context ctx) throws Z3Exception {
    BoolExpr precondition = ctx.mkGe(balance, amount);
    return precondition;
}
```

where function *mkGe* of context creates *balance* greater than or equal to *amount*.

The effect of each operation changes the state of database. For instance, in the bank application, the effect for a withdraw operation is to subtract some *amount* from balance. Using the context's functions for integer day type, we can model the effect as follows:

```
public Expr effect(Context ctx) throws Z3Exception {
    balance = ctx.mkSub(balance, amount);
    return balance;
}
```

where the function *mkSub* of context *ctx* subtracts *amount* from *balance*.

5.2.3 Invariants

An application model contains a set of invariants that restrict the database state. Like preconditions, invariants are boolean expressions over database state. For example, the following invariant expresses that the balance of an account must be positive,

```
public BoolExpr invariant(Context ctx) throws Z3Exception {
    return ctx.mkGe(balance, ctx.mkInt("0"));
}
```

However, an invariant may be more complex. They often require universal and existential quantifiers are denoted by ($\forall$) and ($\exists$), respectively. For instance, consider an invariant that a directory structure may not form a cycle. To model this invariant, we declare a function *ancestor* capturing the ancestor relation between two directories in its argument. The ancestor relation is the transitive closure of parent relation. We say directory u is an ancestor of directory v iff either u is parent of v, or there is a directory w, which is a child of u, and an ancestor of v. The acyclic property implies that if directory u is an ancestor of directory v, then directory v cannot be an ancestor of directory u. The invariant requires that for any pairs of directories, this acyclic property to be true.

```
public BoolExpr invariant(Context ctx) throws Z3Exception {
    FuncDecl ancestor = ctx.mkFuncDecl("ancestor", new Sort[]{(Dit, Dir)};

    Expr u = ctx.mkUninterpretedSort(ctx.mkSymbol("Dir"));
    Expr v = ctx.mkUninterpretedSort(ctx.mkSymbol("Dir"));

    Expr[] argAncestor1 = new Expr[3];
    argAncestor1[0] = u;
    argAncestor1[1] = v;
    argAncestor1[2] = ancestor;
    //creating the ancestor relation between u, and v (u is an ancestor of v)
    Expr ancestorTuple1 = filesystem.Reachability.mkDecl().apply(argAncestor1);

    Expr[] argAncestor2 = new Expr[3];
    argAncestor2[0] = v;
    argAncestor2[1] = u;
    argAncestor2[2] = ancestor;

    //creating the ancestor relation between v, and u (v is an ancestor of u)
    Expr ancestorTuple2 = filesystem.Reachability.mkDecl().apply(argAncestor2);

    Expr acyclic = ctx.mkImplies((BoolExpr) ancestorTuple1, ctx.mknot((BoolExpr) ancestorTuple2));

    //creating (assert(forall((u Dir)(v Dir)) (acyclic u v)))
    Sort[] nodes = new Sort[2];
    nodes[0] = ctx.mkUninterpretedSort(ctx.mkSymbol("Dir"));
    nodes[1] = ctx.mkUninterpretedSort(ctx.mkSymbol("Dir"));
    Symbol[] namess = new Symbol[2];
    namess[0] = ctx.mkSymbol("u");
    namess[1] = ctx.mkSymbol("v");

    //building quntificar
    BoolExpr invariant = ctx.mkForall(nodes, namess, acyclic, 1, null, null, null, null);
    return invariant;
}
```

where *ancestorTuple1* and *ancestorTuple2* specify the ancestor relation between directories u and v. The acyclic property uses an implication function *mkImplies*. The function *mkForall* applies the acyclic property for all pairs of directories.

5.2.4 Tokens

Each operation might be associated with a set of tokens. For instance, consider an account a in the bank application. The withdraw operation might be associated with a token $\tau(a)$, such that $\tau(a) \bowtie \tau(a)$, in order to avoid concurrent withdrawals from the same account. The CISE tool provides a function *conflict* determining the conflict relation between two operations according to their tokens.

```
public BoolExpr conflict(Operation u, Operation v) throws Z3Exception {
    return bowTie(u.tokens, v.tokens);
```

```
public BoolExpr invariant(Context ctx) throws Z3Exception {

    // get precondition of operation op1
    BoolExpr precondition_old = app.getPrecondition(ctx, op1));

    // get invariant
    BoolExpr invariant = app.getInvariants(ctx));

    BoolExpr expr = ctx.mkAnd(invariant, precondition_old);

    // apply effect of operation op2
    Expr effect = app.applyEffect(ctx, op2);

    // get precondition of operation op1 over new state
    BoolExpr precondition_new = app.preCondition(ctx, op1));

    // stabilityRule
    BoolExpr stabilityRule =ctx.mkOr (conflict(op1, op2), ctx.mkImplies(expr, precondition_new));

    //create a new solver
    Solver solver = ctx.mkSolver();

    // solve the negation of proof
    solver.add(ctx.mkNot(correct));

    // find a model(counter−example) that satisfies the solver's rule
    Model model =solver.check(ctx, Status.SATISFIABLE);
}
```

Figure 5.1: CISE stability rule code.

```
}
```

where the function *bowtie* compares sets of tokens for operations u and v and returns true if there is at least one token τ_1 in u's tokens, and one token τ_2 in v's tokens, which are incompatible. Otherwise, it returns false.

5.3 Solver

To verify the application model against a CISE rule, we need to create a Z3 solver using *Context.MkSolver*(). The tool negates the rule, and looks for an operation execution model satisfying the negation using a *Solver.Check*(). If found, it constructs a counter-example to the rule.

For instance, consider the stability analysis that checks the stability of operation's precondition despite concurrent effectors. Figure 5.1 illustrates the code for checking stability of precondition of operation $op1$ against the effect of operation $op2$. Functions *getPrecondition*()

and *getInvariants*() return precondition of operations and application invariants, respectively. Function *applyEffect*() applies the effect of an operation. The *stabilityRule* expression has two boolean expressions, connected by a disjunction. The first expression checks whether tokens of two operations $op1$ and $op2$ are incompatible. The second expression checks whether the precondition of operation $op1$ is true after observing the effect of operation $op2$. To prove a rule, the tool negates the rule. The tool returns a counter-example if there is a program execution satisfying the negation.

5.4 CISE Tool's Parser

Unfortunately, writing the application model in the CISE tool is difficult. In the current version of tool, converting the first-order logic formulas into Z3 format is almost manual. We are developing a translator from some high-level DSL to Z3. Jabczynsk has developed the first version of the Z3 parser for the CISE tool during his master internship in our group. The parser parses Java annotations in a high-level language and translates them to the Z3 internal format. So far, we verified the simple application examples, e.g., bank application using the parser.

Below is a code sample of our tool integrated with the parser for the simple bank account application of previous chapter with two operations deposit and withdraw. The invariant is to keep the balance zero or positive.

```
@XPR("Int balance")
@XPR(value = "balance >= 0", type = XPR.Type.INVARIANT)
@Op(Account.Deposit.class)
@Op(Account.Debit.class)
public class Account extends AnnotatedSchema {

    @XPR(value ={"Int amount","Int balance" },type =XPR.Type.ARGUMENT)
    @XPR(value ="amount >= 0",type = XPR.Type.PRECONDITION)
    @XPR(value ="balance := balance + amount",type =XPR.Type.EFFECT)
    public static class Deposit extends AnnotatedOperation { }

    @XPR(value ={"Int amount","Int balance"},type =XPR.Type.ARGUMENT)
    @XPR(value ="amount >= 0",type = XPR.Type.PRECONDITION)
    @XPR(value ="balance := balance − amount",type =XPR.Type.EFFECT)
    public static class Debit extends AnnotatedOperation { }
}
}
```

To use the tool, users can express the application specification in terms of a set of expressions XPR. Each expression XPR has a value and a type. The value of each expression is given by a high-level DSL. For instance, to write the non-negative invariant, we only need to define an expression with value "*balance* >= *0*" and invariant type.

‘

Chapter 6

The CISE Proof Tool's Application

Contents

6.1 Application/Consistency Co-design 42
6.2 Bank Application 42
6.3 Counter With Escrow 45
6.4 Courseware Application 47
6.5 Auction Application 51
6.5.1 Database 51
6.5.2 Invariant 53
6.5.3 Operations 54

In this chapter, we show how to use the tool to verify and co-design several example applications. These include a generalised banking application, an escrow datatype, a courseware application, and an online auction service. Later, in Chapter 8 we will address a more complex example, the file system.

6.1 Application/Consistency Co-design

We have developed a tool that automates the CISE analysis. The tool checks whether a given application semantics satisfies the three CISE rules, and if it does not, the tool generates a counter-example.

An application developer can leverage the counter-example to identify the source of the problem. The developer corrects the problem either by weakening update semantics or invariants, and/or by strengthening the tokens in order to disallow the execution of conflicting updates. After this refinement, the developer repeats the analysis; and so on, until verification succeeds. Thus, the developer can co-design the application and the consistency protocol, in order to minimise synchronisation, while ensuring correctness.

To illustrate, consider the bank application example. Given its naïve specification in Figure 2.1, our tool finds a counter-example for withdraw. It shows that if the balance is initially zero, withdraw will make it negative. Thus, we derive a sufficient precondition for withdraw, that the balance must be greater than the amount debited. It also finds that this precondition is not stable, generating the following counter-example: if the balance is initially €2, two concurrent debits of €2 each violate the invariant. The bank application developer can then, either disallow concurrent withdrawals with a mutually exclusive token, or remove the non-negative balance invariant. Figure 4.2 shows the co-designed bank specification using tokens, which successfully passes the CISE analysis.

6.2 Bank Application

We first analyse a simple bank application. We extend the example from previous chapter to support multiple bank accounts. The application provides the common bank operations to access and to modify bank accounts. We assume the type Account for account number, and Balance for balance. The database state is a map, named A, of account number $a_1 \in \mathsf{Account}$ to balance $b_1 \in \mathsf{Balance}$. The integrity invariant I that we would like to maintain is that every bank account has a non-negative balance.

$$I = \forall a_1 \in \mathsf{Account}, b_1 \in \mathsf{Balance}, (a_1, b_1) \in A \implies b_1 \geq 0.$$

We define function $\mathsf{balance}(a_1, A)$ to return the current balance for account a_1 stored in map A. We use notation $A[a_1 \mapsto b_1]$ for setting the balance of account a_1 to b_1, while keeping the balance

$$
\begin{aligned}
\mathsf{State} &= A \\
A &= \mathsf{map}(\mathsf{Account}, \mathsf{Balance}) \\
\sigma_{init} &= (\emptyset, \emptyset) \\
\mathit{Token} &= \{\tau_{a_1} \mid a_1 \in \mathsf{Account}\} \\
\bowtie &= \{(\tau_{a_1}, \tau_{a_1}) \mid a_1 \in \mathsf{Account}\} \\
\mathcal{F}_{\mathsf{create}(a_1)}(A) &= (\bot, (\lambda A'.(A' \cup \{(a_1, 0)\})), \emptyset) \\
\mathcal{F}_{\mathsf{deposit}(a_1,v)}(A) &= (\bot, (\lambda A'.(A'[a_1 \mapsto \mathsf{balance}(a_1, A') + v])), \emptyset) \\
\mathcal{F}_{\mathsf{interest}(a_1)}(A) &= (\bot, (\lambda A'.(A'[a_1 \mapsto \mathsf{balance}(a_1, A') + 0.05 * \mathsf{balance}(a_1, A)])), \emptyset) \\
\mathcal{F}_{\mathsf{query}(a_1)}(A) &= (b_1 \mid (a_1, b_1) \in A, \mathsf{skip}, \emptyset) \\
\mathcal{F}_{\mathsf{withdraw}(a_1,v)}(A) &= (\bot, (\lambda A'.(A'[a_1 \mapsto \mathsf{balance}(a_1, A') - v])), \emptyset) \\
\mathcal{F}_{\mathsf{transfer}(a_1,a_2,v)}(A) &= (\bot, (\lambda A'.(A'[a_1 \mapsto \mathsf{balance}(a_1, A') - v][a_2 \mapsto \mathsf{balance}(a_2, A') + v])), \emptyset)
\end{aligned}
$$

Precondition	Operation
true	$\mathcal{F}_{\mathsf{create}(a_1)}(A)$
$v \geq 0$	$\mathcal{F}_{\mathsf{deposit}(a_1,v)}(A)$
true	$\mathcal{F}_{\mathsf{interest}(a_1)}(A)$
true	$\mathcal{F}_{\mathsf{query}(a_1)}(A)$
$v \geq 0$	$\mathcal{F}_{\mathsf{withdraw}(a_1,v)}(A)$
$v \geq 0$	$\mathcal{F}_{\mathsf{transfer}(a_1,a_2,v)}(A)$

Figure 6.1: A simple bank application (incorrect).

of other accounts unchanged:

$$A[a_1 \mapsto b_1] \triangleq A \setminus \{(a_1, b_2) \mid b_2 \in \mathsf{Balance}\} \cup (a_1, b_1). \tag{6.1}$$

The bank application supports operations to access and modify bank accounts. They can create an account $a_1 \in \mathsf{Account}$ with initial balance 0 using $\mathsf{create}(a_1)$ operation. The pair $(a_1, 0)$ is added to the map A. We assume the account numbers are unique. After creating the account, a user can deposit to or withdraw from the account a_1 some positive amount v using $\mathsf{deposit}(a_1, v)$, and $\mathsf{withdraw}(a_1, v)$ operations. A user can transfer some positive amount v from an account a_1 to another account a_2 using the $\mathsf{transfer}(a_1, a_2, v)$ operation; its effect is to withdraw amount v from account a_1, and deposit the same amount v into account a_2. The application defines the $\mathsf{interest}(a_1)$ operation for accruing a 5% interest over account a_1, and $\mathsf{query}(a_1)$ for querying the balance of a_1.

We start with a very weak specification for the bank application, shown in Figure 6.1. Recall that the we assume the state at origin replica satisfies the operation's precondition. Using the CISE tool, we automatically uncover inconsistencies and resolve them. The CISE tool shows that the withdraw and transfer operations violate the effector safety rule, generating counter-examples. For example, if the balance of account a_1 is initially zero, any withdrawal or transfer from the

$$
\begin{aligned}
\mathsf{State} &= A \\
A &= \mathsf{map}(\mathsf{Account}, \mathsf{Balance}) \\
\sigma_{init} &= (\emptyset, \emptyset) \\
\mathit{Token} &= \{\tau_{a_1} \mid a \in \mathsf{Account}\} \\
\bowtie &= \{(\tau_{a_1}, \tau_{a_1}) \mid a \in \mathsf{Account}\} \\
\mathcal{F}_{\mathsf{create}(a_1)}(A) &= (\bot, (\lambda A'.(A \cup \{(a_1, 0)\})), \emptyset) \\
\mathcal{F}_{\mathsf{deposit}(a_1,v)}(A) &= (\bot, (\lambda A'.(A'[a_1 \mapsto \mathsf{balance}(a_1, A') + v])), \emptyset) \\
\mathcal{F}_{\mathsf{interest}(a_1)}(A) &= (\bot, (\lambda A'.(A'[a_1 \mapsto \mathsf{balance}(a_1, A') + 0.05 * \mathsf{balance}(a_1, A)])), \emptyset) \\
\mathcal{F}_{\mathsf{query}(a_1)}(A) &= (b_1 \mid (a_1, b_1) \in A, \mathsf{skip}, \emptyset) \\
\mathcal{F}_{\mathsf{withdraw}(a_1,v)}(A) &= (\bot, (\lambda A'.(A'[a_1 \mapsto \mathsf{balance}(a_1, A') - v])), \{\tau_{a_1}\}) \\
\mathcal{F}_{\mathsf{transfer}(a_1,a_2,v)}(A) &= (\bot, (\lambda A'.(A'[a_1 \mapsto \mathsf{balance}(a_1, A') - v][a_2 \mapsto \mathsf{balance}(a_2, A') + v])), \{\tau_{a_1}\})
\end{aligned}
$$

Precondition	Operation
true	$\mathcal{F}_{\mathsf{create}(a_1)}(A)$
$v \geq 0$	$\mathcal{F}_{\mathsf{deposit}(a_1,v)}(A)$
true	$\mathcal{F}_{\mathsf{interest}(a_1)}(A)$
true	$\mathcal{F}_{\mathsf{query}(a_1)}(A)$
$\mathsf{balance}(a_1, A) \geq v \geq 0$	$\mathcal{F}_{\mathsf{withdraw}(a_1,v)}(A)$
$\mathsf{balance}(a_1, A) \geq v \geq 0$	$\mathcal{F}_{\mathsf{transfer}(a_1,a_2,v)}(A)$

Figure 6.2: Corrected bank application differs from Figure 6.1 as follows: improving precondition of operations and using tokens.

account a_1 will make its balance negative. Therefore, to fix this issue, we add preconditions to ensure that a withdrawal is possible only if the account has sufficient balance.

Then, we run the commutativity test. It shows that all bank operations are commutative. Note that we compute interest at the origin replica, and the effector adds this amount to the local balance at each replica.

Finally, we check the stability rule. It shows that the precondition of the withdraw(a_1,.) operation is not stable, under the concurrent effect of other withdraw(a_1,.) operations, and under the concurrent effect of transfer(a_1, a_2,.) operations. The tool returns the following counter-example. Let balance of account a_1 be initially $\in 2$. The precondition to withdraw($a_1, 1$) is verified. However, a concurrent withdraw($a_1, 2$) (whose precondition is also OK) makes the balance zero, now violating the precondition of withdraw($a_1, 1$).

To fix this problem, different alternatives are possible. If freedom from synchronisation is important, the only alternative is to weaken the invariant, e.g., remove the non-negative invariant. The traditional approach is to add some concurrency control in order to disallow the concurrent execution of conflicting operations. We choose the latter approach. A token τ_{a_1} is associated to a withdrawal from account a_1, such that $\tau_{a_1} \bowtie \tau_{a_1}$. Thus, withdrawals from the same account

$$\begin{aligned}
\mathsf{State} &= \mathsf{Credit} \times \mathsf{map}(\mathsf{ReplicaID} \times \mathsf{Credit}) \\
\mathit{Token} &= \{\tau_r \mid r \in \mathsf{ReplicaID}\} \\
\bowtie &= \{(\tau_r, \tau_r) \mid r \in \mathsf{ReplicaID}\} \\
\mathcal{F}_{\mathsf{increment}(r,k)}(n, C) &= (\bot, (\lambda(n, C').(n, C'[r \mapsto c + k])), \{\tau_r\}) \\
\mathcal{F}_{\mathsf{decrement}(r,k)}(n, C) &= (\bot, (\lambda(n, C').(n, C'[r \mapsto c - k])), \{\tau_r\}) \\
\mathcal{F}_{\mathsf{acquireCredit}(r,k)}(n, C) &= (\bot, (\lambda(n, C').(n - k, C'[r \mapsto c + k])), \emptyset) \\
\mathcal{F}_{\mathsf{releaseCredit}(r,k)}(n, C) &= (\bot, (\lambda(n, C').(n + k, C'[r \mapsto c - k])), \emptyset)
\end{aligned}$$

Precondition	Operation
$k \geq 0$	$\mathcal{F}_{\mathsf{increment}(r,k)}(n, C)$
$k \geq 0$	$\mathcal{F}_{\mathsf{decrement}(r,k)}(n, C)$
$k \geq 0$	$\mathcal{F}_{\mathsf{acquireCredit}(r,k)}(n, C)$
$k \geq 0$	$\mathcal{F}_{\mathsf{releaseCredit}(r,k)}(n, C)$

Figure 6.3: Counter With Escrow (incorrect).

synchronise. However, withdrawals from different accounts can execute without synchronisation, because $\tau_{a_1} \not\bowtie \tau_{a_2}$ for $a_1 \neq a_2$. In order to preserve the non-negativity of balance, both withdraw and transfer must acquire the corresponding token. Figure 6.2 depicts the modified specification of bank account application with the changes outlined above. The application semantics now successfully passes the CISE analysis.

6.3 Counter With Escrow

Counters are useful abstractions in many applications, such as counting ad impressions, or virtual wallets. A counter has a value and supports increment and decrement operations to update its value. The value of the counter should represent the sum of increments minus the decrements. Many applications require the counter to be bounded. For instance, consider an advertisement application counting the number of times a specified ad is displayed. The ad should not be shown any more once it has been displayed some maximum of times.

However, if concurrency is allowed, the counter may violate its limit. The CISE analysis proves that it is not possible to enforce the upper bound of the replicated counter without avoiding concurrent increments. Balegas et al, [19] propose a new replicated data type, called the *bounded counter,* which enforces this kind of invariants, while removing synchronisation from the critical execution path. The authors further extend their work to exploiting reservation techniques in geo-replicated clouds [18]. The main idea behind bounded counters comes from escrow [78], which partitions shares of resource among replicas. Thus, a *credit* is assigned to each replica. The replica can perform an update without synchronising with other replicas, as long as it has sufficient local credit. Otherwise, a demarcation protocol [20] or a synchronous protocol [27] is required to allow

the replica to acquire the more credits.

To help programers to exploit escrow, we design and verify a new data type, called *Counter With Escrow*. It is similar to the bounded counter. We assume a set of local credits Credit, and a set of replica identifiers ReplicaID. The database is a map of local credits to replicas, named C. A replica $r \in \mathsf{ReplicaID}$ is assigned with a local credit $c \in \mathsf{Credit}$, i.e., $(r,c) \in C$. There is a global credit with operations acquireCredit(r,k) and releaseCredit(r,k). The global credit initially has value n. Replica r can send a request to get some k credits of the global credit using the acquireCredit(r,k) operation; its effect is to decrease global credit available n by k if $k \leq n$, and to increase local credit c by amount k. Replica r can release k credits of its local credit using the releaseCredit(r,k) operation; its effect is to decrease local credit by k, and to increase global credit (n) by k. An increment(r,k) operation executed at replica r increases its local credit c by k. For instance, consider a non-negative counter, performing increment operations at a replica increases the replica's local credit. A decrement(r,k) operation executed at replica r decreases its local credit c by k. Each replica is sequential, meaning that all operations within a single replica execute in some sequential order. To capture this behaviour in our specification, we assign an artificial token τ_r to replica r, such that $\tau_r \bowtie \tau_r$. Thus, no concurrent updates to a local credit happens. However, different replicas can modify their local credit independently, i.e., $\tau_r \not\bowtie \tau_{r'}$ for $r \neq r'$.

The invariant I that we must maintain is that the credit is always positive:

$$I = (\forall c \in \mathsf{Credit}, r \in \mathsf{ReplicaID}, (r,c) \in C \implies c \geq 0) \wedge (n \geq 0)$$

where n is the value of the global credit.

We use the CISE analysis to verify the specification of Counter with Escrow in Figure 6.3.

The effector safety analysis returns a counter-example for the acquireCredit operation if sufficient global credit does not exist. The tool returns the following counter-example. Let the global credit be initially zero. Applying the effect of operation acquireCredit$(r,1)$ would make it negative. The analysis also shows that the decrement(r,k) operation requires that sufficient local credit exists in the replica r. Otherwise, applying it's effect might violate the invariant. Assume that replica r has no local credit, any decrement will make its local credit negative. We add the corresponding preconditions.

The commutativity analysis proves that all escrow operations are commutative.

Finally, running the tool for stability analysis returns counter-examples for the acquireCredit operation. Let the global credit be initially 1. Replica r wants to acquire one credit using the acquireCredit$(r,1)$ operation. The precondition of acquireCredit$(r,1)$ is verified. However, a concurrent acquireCredit$(r',1)$ operation makes the global credit zero, and so, the precondition of acquireCredit$(r,1)$ is not true anymore. If replica r was to acquire the credit, then the invariant would become false. To address this problem, we assign a mutually exclusive token τ, to acquireCredit operations, so that they cannot be concurrent. We run the tool again to verify that the problem is resolved.

$$
\begin{aligned}
\mathsf{State} &= \mathsf{Credit} \times \mathsf{map}(\mathsf{ReplicaID} \times \mathsf{Credit}) \\
\mathit{Token} &= \{\tau_r \mid r \in \mathsf{ReplicaID}\} \cup \{\tau\} \\
\bowtie &= \{(\tau_r, \tau_r) \mid r \in \mathsf{ReplicaID}\} \cup \{(\tau, \tau)\} \\
\mathcal{F}_{\mathsf{increment}(r,k)}(n, C) &= (\bot, (\lambda(n, C').(n, C'[r \mapsto c + k])), \{\tau_r\}) \\
\mathcal{F}_{\mathsf{decrement}(r,k)}(n, C) &= (\bot, (\lambda(n, C').(n, C'[r \mapsto c - k])), \{\tau_r\}) \\
\mathcal{F}_{\mathsf{acquireCredit}(r,k)}(n, C) &= (\bot, (\lambda(n, C').(n - k, C'[r \mapsto c + k])), \{\tau\}) \\
\mathcal{F}_{\mathsf{releaseCredit}(r,k)}(n, C) &= (\bot, (\lambda(n, C').(n + k, C'[r \mapsto c - k])), \emptyset)
\end{aligned}
$$

Precondition	Operation
$k \geq 0$	$\mathcal{F}_{\mathsf{increment}(r,k)}(n, C)$
$\mathsf{Credit}(r, C) \geq k \geq 0$	$\mathcal{F}_{\mathsf{decrement}(r,k)}(n, C)$
$n \geq k \geq 0$	$\mathcal{F}_{\mathsf{acquireCredit}(r,k)}(n, C)$
$k \geq 0$	$\mathcal{F}_{\mathsf{releaseCredit}(r,k)}(n, C)$

Figure 6.4: Corrected Counter With Escrow differs from Figure 6.3 as follows: using tokens and improving preconditions.

Note, if we remove the token τ_r assigned into decrement and increment operations, the tool returns another counter-example for two concurrent decrement operations within a replica.

Figure 6.4 gives the design of counter with escrow after applying all changes detailed above. Running the tool again, we verify that the semantics successfully passes all CISE rules.

Returning to the bank application, we can use Counter With Escrow design to implement the balance. Thus, a particular bank branch could acquire a portion of the account's balance, say €1000 out of a balance of €5000. This gives the branch the capability to make any number of debits, up to €1000, without communicating.

6.4 Courseware Application

The next application that we analyse and co-design is a courseware application. Its database stores information about students and courses. The operations are as follows: A user can add a course c using addCourse(c) operation and register a student s using register(s) operation. The registered student s can enroll in the course c using enroll(s, c). The student registration and enrollment can be cancelled using deregister(s), and disenroll(s, c) operations. Course c can be removed using remCourse(c) operation. We also have a query operation.

We assume types course Course and student Student. A database state (S, C, E) consists of a set of students S, a map C for courses, and the enrolment relation E between students and courses. Each course c has a capacity capacity(c), representing the maximum number of students that can be enrolled. The map C maps a course $c \in \mathsf{Course}$ to an integer number $n \in \mathbb{N}$, which counts the number of students enrolled in course c. Counter n is initially zero, and increments by

$$
\begin{aligned}
\mathsf{State} &= S \times C \times E \\
S &= \mathsf{AWset(Student)}, \quad C = \mathsf{AWmap(Course,\mathbb{N})}, \quad E = \mathsf{AWset(Student \times Course)} \\
\sigma_{init} &= (\emptyset_{\mathsf{AWset}}, \emptyset_{\mathsf{AWmap}}, \emptyset_{\mathsf{AWset}}) \\
\mathit{Token} &= \emptyset \\
\bowtie &= \emptyset \\
\mathcal{F}_{\mathsf{register}(s)}(S,C,E) &= (\bot, (\lambda(S',C',E').(S'.\mathsf{add}(s), C', E')), \emptyset) \\
\mathcal{F}_{\mathsf{deregister}(s)}(S,C,E) &= (\bot, (\lambda(S',C',E').(S'.\mathsf{remove}(s), C', E')), \emptyset) \\
\mathcal{F}_{\mathsf{addCourse}(c)}(S,C,E) &= (\bot, (\lambda(S',C',E').(S', C'.\mathsf{add}(c,0), E')), \emptyset) \\
\mathcal{F}_{\mathsf{enroll}(s,c)}(S,C,E) &= (\bot, (\lambda(S',C',E').(S', C'.\mathsf{add}(c,n++), E'.\mathsf{add}(s,c))), \emptyset) \\
\mathcal{F}_{\mathsf{disenroll}(s,c)}(S,C,E) &= (\bot, (\lambda(S',C',E').(S', C'.\mathsf{add}(c,n--), E'.\mathsf{remove}(s,c))), \emptyset) \\
\mathcal{F}_{\mathsf{remCourse}(c)}(S,C,E)) &= (\bot, (\lambda(S',C',E').(S', C'.\mathsf{remove}(c), E')), \emptyset)
\end{aligned}
$$

Precondition	Operation
true	$\mathcal{F}_{\mathsf{register}(s)}$
$s \in S$	$\mathcal{F}_{\mathsf{deregister}(s)}(S,C,E)$
true	$\mathcal{F}_{\mathsf{addCourse}(c)}(S,C,E)$
$s \in S \wedge (c,n) \in C$	$\mathcal{F}_{\mathsf{enroll}(s,c)}(S,C,E)$
$(s,c) \in E$	$\mathcal{F}_{\mathsf{disenroll}(s,c)}(S,C,E)$
$c \in \mathsf{Course}$	$\mathcal{F}_{\mathsf{remCourse}(c)}(S,C,E)$

Figure 6.5: Courseware application (incorrect).

one when a student is enrolled in course c, and decreases if a student cancels her enrolment in the course c. There are two integrity invariants: I_1 and I_2. Invariant I_1 states that the number of students enrolled in a course must not exceed its capacity. Invariant I_2 states that the enrolment relation refers to existing courses and registered students; it is an instance of a foreign key integrity rule in databases, which requires a data item referenced in one part of the database to exist in another.

$$
\begin{aligned}
I_1 &= \forall c \in \mathsf{Course}, (c,n) \in C \implies n \leq \mathsf{capacity}(c) \\
I_2 &= \forall c \in \mathsf{Course}, s \in \mathsf{Student}, (c,s) \in E \implies s \in S \wedge \exists n \in \mathbb{N}, (c,n) \in C
\end{aligned}
$$

We exploit CRDTs to ensure commutative semantics for all operations. See Part I. Figure 6.5 gives the simple specification for the courseware application using an add-wins approach. Operations' effectors use the add and remove operations of a replicated set or map. For instance, consider the replicated map C. The concrete implementation of AWmap data type attaches a unique tag to each added element. To add a course c, the effector $\mathcal{F}_{(addcourse(c))}$ adds the pair (c,n) to C using the $\mathsf{add}(c,n)$ function provided by the AWmap API. The course c is removed by the $\mathsf{remove}(c)$ function provided by the AWmap API.

The effector safety analysis identifies and verifies sufficient preconditions. It shows that the effector safety analysis shows that the effect of the $\mathsf{enroll}(c,s)$ operation may break invariant I_2.

$$
\begin{aligned}
\mathsf{State} &= S \times C \times E \\
S &= \mathsf{AWset(Student)}, \quad C = \mathsf{AWmap(Course,\mathbb{N})}, \quad E = \mathsf{AWset(Student \times Course)} \\
\sigma_{init} &= (\emptyset_{\mathsf{AWset}}, \emptyset_{\mathsf{AWmap}}, \emptyset_{\mathsf{AWset}}) \\
\mathit{Token} &= \{\tau_{e(c)}, \tau_{r(c)}, \tau_c, \tau_{e(s)}, \tau_{r(s)} \mid c \in \mathsf{Course}, s \in \mathsf{Student}\} \\
\bowtie &= \{(\tau_{e(c)}, \tau_{r(c)}), (\tau_{r(c)}, \tau_{e(c)}), (\tau_c, \tau_c), (\tau_{e(s)}, \tau_{r(s)}), (\tau_{r(s)}, \tau_{e(s)}) \mid \\
&\qquad c \in \mathsf{Course}, s \in \mathsf{Student}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}_{\mathsf{register}(s)}(S,C,E) &= (\bot, (\lambda(S',C',E').(S'.\mathsf{add}(s), C', E')), \emptyset) \\
\mathcal{F}_{\mathsf{deregister}(s)}(S,C,E) &= (\bot, (\lambda(S',C',E').(S'.\mathsf{remove}(s), C', E')), \{\tau_{r(s)}\}) \\
\mathcal{F}_{\mathsf{addCourse}(c)}(S,C,E) &= (\bot, (\lambda(S',C',E').(S', C'.\mathsf{add}(c,0), E')), \emptyset) \\
\mathcal{F}_{\mathsf{enroll}(s,c)}(S,C,E) &= (\bot, (\lambda(S',C',E').(S', C'.\mathsf{add}(c,n++), E'.\mathsf{add}(s,c)), \{\tau_{e(c)}, \tau_{e(s)}, \tau_c\}) \\
\mathcal{F}_{\mathsf{disenroll}(s,c)}(S,C,E) &= (\bot, (\lambda(S',C',E').(S', C'.\mathsf{add}(c,n--), E'.\mathsf{remove}(s,c)), \emptyset) \\
\mathcal{F}_{\mathsf{remCourse}(c)}(S,C,E)) &= (\bot, (\lambda(S',C',E').(S', C'.\mathsf{remove}(c), E')), \{\tau_{r(c)}\})
\end{aligned}
$$

Precondition	Operation
true	$\mathcal{F}_{\mathsf{register}(s)}$
$s \in S \wedge \nexists c \in \mathsf{Course}, (s,c) \in E$	$\mathcal{F}_{\mathsf{deregister}(s)}(S,C,E)$
true	$\mathcal{F}_{\mathsf{addCourse}(c)}(S,C,E)$
$s \in S \wedge (c,n) \in C \wedge n < \mathsf{capacity}(c)$	$\mathcal{F}_{\mathsf{enroll}(s,c)}(S,C,E)$
$(s,c) \in E$	$\mathcal{F}_{\mathsf{disenroll}(s,c)}(S,C,E)$
$c \in \mathsf{Course} \wedge \nexists s \in \mathsf{Student}, (s,c) \in E$	$\mathcal{F}_{\mathsf{remCourse}(c)}(S,C,E)$

Figure 6.6: Corrected courseware application differs from Figure 6.5 as follows: using tokens and improving preconditions.

The counter-example is when a student s enrolls in a non-existing course or a non-registered student enrolls in a course. To avoid such anomalies, we add sufficient preconditions, such that $\mathsf{enroll}(s,c)$ operation requires that student s is registered and course c exists.

The commutativity analysis shows that the effect of operations using CRDTs commute. The add-wins CRDT set semantics guarantees the commutativity property. For instance, suppose Alice adds a course c using $\mathsf{addCourse}(c)$ operation, then changes her mind and removes the course using $\mathsf{remCourse}(c)$ operation; concurrently, Bob adds the same course c using $\mathsf{addCourse}(c)$ operation. Independent of the order in which the replicas apply the effects of the concurrent operations $\mathsf{addCourse}(c)$ and $\mathsf{remCourse}(c)$, the result of execution will be the same in both Bob's and Alice's replica. A query operation will return the same result $c \in \mathsf{Course}$.

The concurrent executions of operations may break invariants I_1 and I_2. The stability analysis generates a counter-example, indicating that the precondition of the enroll operation is not stable under concurrent effect of another enroll operation. The counter example is the same as a bounded counter. Thus, to preserve the capacity limit, a mutual token τ_c is associated to a course c, such that $\tau_c \bowtie \tau_c$. Then enrolments in the same course will have to synchronise. However, users can

	Invariant
$I_1 =$	$\forall s \in \mathsf{Seller}, p \in \mathsf{Product}, (s,p) \in O \implies (s,.) \in S \wedge (p,.) \in P$
$I_2 =$	$\forall a \in \mathsf{Auction}, s \in \mathsf{Seller}, (a,s) \in E \implies (a,.) \in A \wedge (s,.) \in S$
$I_3 =$	$\forall b \in \mathsf{Bid}, a \in \mathsf{Auction}, c \in \mathsf{Customer}, (b,a,c,.) \in B \implies (a,.) \in A \wedge c \in C$
$I_4 =$	$\forall l \in \mathsf{Lot}, a \in \mathsf{Auction}, s \in \mathsf{Seller}, p \in \mathsf{Product}, (l,a,s,p,.) \in L \implies (a,s) \in E \wedge (s,p) \in O$
$I_5 =$	$\forall a \in \mathsf{Auction}, c \in \mathsf{Customer}, (a,c) \in W \implies (a,.) \in A \wedge c \in C$
$I_6 =$	$\forall p \in \mathsf{Product}, \mathsf{stock}(p,P) \geq 0$
$I_7 =$	$\forall l \in \mathsf{Lot}, a \in \mathsf{Auction}, s \in \mathsf{Seller}, p \in \mathsf{Product}, (l,a,s,p,k) \in L \implies (s,n) \in S \wedge k \leq n$ $\wedge (p,m) \in S \wedge k \leq m$
$I_8 =$	$\forall a \in \mathsf{Auction}, \mathsf{active}(a) \vee \mathsf{closed}(a) \implies \exists p \in \mathsf{Product}, (l,a,p,k) \in L \wedge k > 0$
$I_9 =$	$\forall a \in \mathsf{Auction}, (a,c) \in W \implies \mathsf{closed}(a)$
$I_{10} =$	$\forall a \in \mathsf{Auction}, \mathsf{closed}(a) \implies \exists c \in \mathsf{Customer}, (a,c) \in W \wedge (b,a,c,.) \in B \wedge b == \max(B)$
$I_{11} =$	$\forall a, a' \in \mathsf{Auction}, (\mathsf{active}(a) \wedge (close(a') \vee \mathsf{open}(a')) \vee (\mathsf{open}(a) \wedge$ $(\mathsf{closed}(a') \vee \mathsf{active}(a')) \vee (\mathsf{closed}(a) \wedge (\mathsf{open}(a') \vee \mathsf{active}(a')) \implies a \neq a'$

Table 6.1: Auction invariants.

enroll in different courses without synchronisation, i.e., $\tau_c \not\bowtie \tau_{c'}$ for $c \neq c'$.

The stability analysis also identifies that concurrent execution of enroll and deregister operations, or of enroll and remCourse operations is not safe, as they may violate invariant I_2. Here is a counter-example: consider that no students enrolled in course c. Alice removes the course while Bob enrolls concurrently into it. The results is that Bob is enrolled into a non-existent course. To disallow such situations, we can define a mutually exclusive token τ_c that both enroll and remCourse operations must acquire. The token τ_c totally orders the enroll and remCourse operations. However, this incurs unnecessary synchronisation. For instance, acquiring this token disallows concurrent removing the same course.

Similar to readers-writer locks from shared memory [18], we can define multi-level tokens in order to reduce the cost of the mutually exclusive token. We provide a multi-level lock abstractions to each data item: token τ_e giving the shared right to forbid removing the data item, and token τ_r giving the shared right to allow removing the data item, such that token τ_e is incompatible with token τ_r over the same data item.

Therefore, we assign a pair of incompatible tokens $\tau_{e(c)}$ and $\tau_{r(c)}$ to a course c, such that $\tau_{e(c)} \bowtie \tau_{r(c)}$, and a pair of incompatible tokens $\tau_{e(s)}$ and $\tau_{r(s)}$ to a student s, such that $\tau_{e(s)} \bowtie \tau_{r(s)}$. Neither of these tokens are incompatible with itself. Operation enroll(s,c) acquires both tokens $\tau_{e(c)}$, and $\tau_{e(s)}$, whereas operation remCourse(c) and deregister(s) acquire token $\tau_{r(c)}$ and token $\tau_{r(s)}$. Then for every pair of operations enroll(s,c) and remCourse(c) (or deregister(s)), either the enrolment operation is aware that the course (or the student) has been removed, or the removal is aware of the enrolment operation; in either case the corresponding operation has no effect.

State	:	set(Customer)	(customer id)
		×map(Seller,ℕ)	(seller id, limit)
		×map(Product,ℕ)	(product id, stock)
		×map(Auction,{open,active,closed})	(auction id, status)
		×set(Seller × Product)	(seller id, product id)
		×set(Auction × Seller)	(auction id, seller id)
		×set(Auction × Customer)	(auction id, customer id)
		×set(Bid × Auction × Customer × ℕ)	(bid, auction id, customer id, price)
		×set(Lot × Auction × Seller × Product × ℕ)	(lot, auction id, seller id, product id, size)

Figure 6.7: Auction database state

However, other pairs of operations can be concurrent, and do not have to synchronise.

Figure 6.6 illustrates the corrected courseware semantics, which includes the sufficient precondition, the set of required tokens, and their incompatibility relation. The CISE analysis verifies that the preconditions and the token assignments are indeed sufficient to ensure the application invariants.

6.5 Auction Application

Our most complex example in this chapter concerns an online auction application similar to eBay. The application maintains information about customers, sellers, products, and auctions. Customers and sellers can register in the application and unregister from it. Registered sellers can create auctions and then add products as lots into the auctions. The status of auction can be one of: open, active, or closed. While an auction is active, a registered customer can place a bid. Auctions may involve one or more product items, one or more sellers, and one or more bidders. Once the auction is closed, the bidder with the highest bid is declared the winner.

6.5.1 Database

We assume the type customers Customer, sellers Seller, products Product, auctions Auction, bids Bid, and lots Lot. The database state is composed of the set of customers C, the map S for sellers, the map P for products, and the map A for auctions. Map S maps a seller $s \in$ Seller to a limit limit(s), the maximum number of products that seller s can auction. Map P maps a product $p \in$ Product to its available quantity, denoted by stock(p,P). Map A stores data about auctions and their status. To store the status of an auction, we use a "linear type" that has successive states S_1 = open, S_2 = active, and S_3 = closed with operations read and advance(S_i), where advance(S_i) has precondition status = $S_{(i-1)}$ and sets state to S_i. Function open(a) states that auction a exists

$$
\begin{aligned}
\text{State} &= C \times S \times P \times A \times O \times E \times W \times B \times L \\
C &= \text{AWset(Customer)}, \quad S = \text{AWmap(Seller,}\mathbb{N}\text{)}, \quad P = \text{AWmap(Product,}\mathbb{N}\text{)}, \\
A &= \text{AWmap(Auction, status)}, \quad O = \text{AWset(Seller} \times \text{Product)}, \\
E &= \text{AWset(Auction} \times \text{Seller)}, \quad W = \text{AWset(Auction} \times \text{Customer)}, \\
B &= \text{AWset(Bid} \times \text{Auction} \times \text{Customer} \times \mathbb{N}\text{)}, \quad L = \text{AWset(Lot} \times \text{Auction} \times \text{Seller} \times \mathbb{N}\text{)} \\
\sigma_{init} &= (\emptyset_{\text{AWset}}, \emptyset_{\text{AWmap}}, \emptyset_{\text{AWmap}}, \emptyset_{\text{AWmap}}, \emptyset_{\text{AWset}}, \emptyset_{\text{AWset}}, , \emptyset_{\text{AWset}}, \emptyset_{\text{AWset}}, \emptyset_{\text{AWset}}) \\
\mathit{Token} &= \emptyset \\
\bowtie &= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}_{\text{regCustomer}(c)}(\sigma) &= (\bot, (\lambda(\sigma'.C).(\sigma'.C.\text{add}(c))), \emptyset) \\
\mathcal{F}_{\text{unRegCustomer}(c)}(\sigma) &= (\bot, (\lambda(\sigma'.C).(\sigma'.C.\text{remove}(c))), \emptyset) \\
\mathcal{F}_{\text{regSeller}(s,n)}(\sigma) &= (\bot, (\lambda(\sigma'.S).(\sigma'.S.\text{add}(s,n))), \emptyset) \\
\mathcal{F}_{\text{unRegSeller}(s)}(\sigma) &= (\bot, (\lambda(\sigma'.S).(\sigma'.S.\text{remove}(s))), \emptyset) \\
\mathcal{F}_{\text{addProduct}(p,m,s)}(\sigma) &= (\bot, (\lambda(\sigma'.P, \sigma'.O).(\sigma'.P.\text{add}(p,m), \sigma'.O.\text{add}(p,s))), \emptyset) \\
\mathcal{F}_{\text{remProduct}(p)}(\sigma) &= (\bot, (\lambda(\sigma'.P, \sigma'.O).(\sigma'.P.\text{remove}(p), \sigma'.O.\text{remove}(p,s))), \emptyset) \\
\mathcal{F}_{\text{ctrAuction}(a,s)}(\sigma) &= (\bot, (\lambda(\sigma'.A, \sigma'.E).(\sigma'.A.\text{add}(a, \text{open}), \sigma'.E.\text{add}(a,s))), \emptyset) \\
\mathcal{F}_{\text{remAuction}(a)}(\sigma) &= (\bot, (\lambda(\sigma'.A, \sigma'.E).(\sigma'.A.\text{remove}(a), \sigma'.E.\text{remove}(a,s))), \emptyset) \\
\mathcal{F}_{\text{addLot}(l,a,s,p,k)}(\sigma) &= (\bot, (\lambda(\sigma'.P, \sigma'.L).(\sigma'.P.\text{add}(p, \text{stock}(p, \sigma'.P) - k), \sigma'.L.\text{add}(l,a,s,p,k))), \emptyset) \\
\mathcal{F}_{\text{remLot}(l,a,s,p,k)}(\sigma) &= (\bot, (\lambda(\sigma'.P, \sigma'.L).(\sigma'.P.\text{add}(p, \text{stock}(p, \sigma'.P) + k), \sigma'.L.\text{remove}(l,a,s,p,k))), \emptyset) \\
\mathcal{F}_{\text{placeBid}(b,a,c,v)}(\sigma) &= (\bot, (\lambda(\sigma'.B).(\sigma'.B.\text{add}(b,a,c,v))), \emptyset) \\
\mathcal{F}_{\text{remBid}(b,a,c,n)}(\sigma) &= (\bot, (\lambda(\sigma'.B).(\sigma'.B.\text{remove}(b,a,c,v))), \emptyset) \\
\mathcal{F}_{\text{startAuction}(a)}(\sigma &= (\bot, (\lambda(\sigma'.A).(\sigma'.A.\text{add}(a, \text{active}))), \emptyset) \\
\mathcal{F}_{\text{closeAuction}(a,c)}(\sigma) &= (\bot, (\lambda(\sigma'.A, \sigma'.W).(\sigma'.A.\text{add}(a, \text{closed}), \sigma'.W.\text{add}(a,c))), \emptyset)
\end{aligned}
$$

Precondition	Operation
true	$\mathcal{F}_{\text{regCustomer}(c)}(\sigma)$
$c \in \sigma.C$	$\mathcal{F}_{\text{unRegCustomer}(c)}(\sigma)$
true	$\mathcal{F}_{\text{regSeller}(s,n)}(\sigma)$
$(s,.) \in \sigma.S$	$\mathcal{F}_{\text{unRegSeller}(s)}(\sigma)$
true	$\mathcal{F}_{\text{addProduct}(p,m,s)}(\sigma)$
$(p,.) \in \sigma.P$	$\mathcal{F}_{\text{remProduct}(p)}(\sigma)$
true	$\mathcal{F}_{\text{ctrAuction}(a,s)}(\sigma)$
$(a,.) \in \sigma.A$	$\mathcal{F}_{\text{remAuction}(a)}(\sigma)$
true	$\mathcal{F}_{\text{addLot}(l,a,s,p,k)}(\sigma)$
$(l,.,.,.) \in \sigma.L$	$\mathcal{F}_{\text{remLot}(l,a,s,p,k)}(\sigma)$
true	$\mathcal{F}_{\text{placeBid}(b,a,c,v)}(\sigma)$
$(b,.,.,.) \in \sigma.B$	$\mathcal{F}_{\text{remBid}(b,a,c,v)}(\sigma)$
$(a,.) \in \sigma.A \wedge \text{open}(a)$	$\mathcal{F}_{\text{startAuction}(a)}(\sigma)$
$(a,.) \in \sigma.A \wedge \text{active}(a)$	$\mathcal{F}_{\text{closeAuction}(a,c)}(\sigma)$

Figure 6.8: Auction application (incorrect).

and is open, i.e., $(a, \text{open}) \in A$. Function active(a) states that auction a exists and is started, i.e., $(a, \text{active}) \in A$. Function closed(a) states that auction a exists and is closed, i.e., $(a, \text{closed}) \in A$. The database also maintains several relations: owner relation O between sellers and products, promoter relation E between auctions and sellers, winner relation W between auctions and customers, bid relation B between auctions and customers, and lot relation L between auctions and products. Bid relation B maps a customer to a price v, which she offers for a particular auction. Lot relation L maps quantity of products to a given auction. Figure 6.7 illustrates the database.

6.5.2 Invariant

This application has eleven integrity rules listed in Table 6.1, which include: foreign key constraints, stock constraints, and restrictions over auction's status. Invariants $I_1 - I_5$ are referential integrity rules over the owner, promoter, bid, lot, and winner relations. Invariant I_1 states that an owner maps to an existing product and a registered seller. Invariant I_2 states that a promoter maps to an existing auction and a registered seller. Invariant I_3 states that a bid maps to a registered customer and an existing auction. Invariant I_4 states that a lot maps to an existing auction, a registered seller, and an existing product. Invariant I_5 ensures that winner of each auction is a customer, who has been registered.

Invariant I_6 ensures that no product is out of stock. Invariants I_7 states that the size of a product lot must be less than the stock, and the seller's limit. Invariant I_8 states that there is at least one lot for any closed or active auction. Invariant I_9 states that if a winner is declared for an auction, then the auction must be closed. Invariant I_{10} states that when an auction is closed, it has a winning bidder, and her bid is the maximal one. Invariant I_{11} states that an auction can be only in one of three statuses.

In addition, there might be some restrictions over state transitions. For instance, the application might require to ensure that lots are not placed or removed from a closed or active auction. For this, when the auction starts, the application stores all its lots in another list, say AL, which can later be checked against the list of lots L. If $(a, l) \in \text{AL}$ asserts that lot l is added in the auction a before it begins, and active(a) asserts that auction a is active, we can introduce following invariant:

$$I_{12} = \forall l \in \text{Lot}, a \in \text{Auction}, \text{active}(a) \wedge (l, a, ., ., .) \in L \implies (l, a) \in \text{AL}$$

The same condition may be required for bids, as customers add bids to or remove from an auction only when the auction is active. We define another invariant I_{13}, where a list BL stores all bids placed in an auction before the auction is closed. To avoid changes into set B for auctions, which are not active, the list BL is checked against the list of bids B:

$$I_{13} = \forall b \in \text{Bid}, a \in \text{Auction}, \text{closed}(a) \wedge (b, a, ., .) \in B \implies (b, a) \in \text{BL}$$

Alternatively, temporal logic expressions can be used to address such forms of temporal specifications [65]. However, temporal logic would entail more complex specification for programmers, and more complex analysis.

6.5.3 Operations

The auction application supports different update operations. For instance, one can register seller s using the regSeller(s,n) operation, which its effect adds (s,n) to map S, and can unregister seller s using the unRegSeller(s) operation, which removes (s,n) from the map S. Once seller s has been registered, she can add or remove a product p with initial stock m, using the addProduct(p,s,m) and remProduct(p) operations, respectively. After adding the product, she can create an auction a, using the ctrAuction(a,s) operation. The addLot(a,s,p,n) operation allows seller s to add some lots of product p to the auction a if its status is open. After activating the auction a using the startAuction(a) operation, no changes are allowed to the auction's lots. A registered customer c can place a bid b valued v over the auction a using the placeBid(b,a,c,v) operation. Until an auction is active, bids can be added to, or removed from the auction. When the auction a is closed by the closeAuction(a,c) operation, customer c is declared the winning bidder if her bid is the maximal of all bids placed on the same auction. As usual, we also have a query operation.

Similar to the courseware application, we exploit CRDTs to ensure *commutative* semantics for all operations of the auction application without introducing synchronisation. We use add-wins approach to implement set or map data items. Figure 6.8 shows simple specification model for the auction application.

Preconditions guarantee sequential invariants. Running the tool for effector safety on the auction specification model enables us to identify and verify sufficient preconditions for each operation. For instance, to ensure invariant I_3 in any sequential execution, the effector safety analysis shows that the placeBid operation may not place a bid into an auction that does not exist. We add the corresponding precondition.

Thanks to CRDTs, the commutativity analysis proves that the effect of operations commute. However, running the CISE tool for stability analysis shows that the precondition of some operations is not stable under concurrent execution. For instance, the stability check for the placeBid operation fails under concurrent remAuction operation. The tool returns the following counter example. Alice removes auction a, which has no bids, and concurrently Bob places a bid on the auction a, thinking that the auction exists. This results in placing bid on a non-existent auction. To avoid such situations, we specify a pair of conflicting tokens for each auction $a \in$ Auction: $\tau_{a(a)}$ and $\tau_{r(a)}$. Thus, the operation placeBid acquires $\tau_{a(a)}$, and the operation remAuction acquires $\tau_{r(a)}$. Using the stability analysis, we identify which pairs of operations must not be concurrent, i.e., those whose precondition is not be stable under concurrent effect of another, and then define sufficient tokens to disallow toxic concurrency. Figure 6.9 illustrates the correct semantics of auction application using the above modifications. Thanks to the tool, we are able to

$$
\begin{aligned}
\text{State} &= C \times S \times P \times A \times O \times E \times W \times B \times L \\
C &= \text{AWset(Customer)}, \quad S = \text{AWmap(Seller,}\mathbb{N}\text{)}, \quad P = \text{AWmap(Product,}\mathbb{N}\text{)}, \\
A &= \text{AWmap(Auction} \times \text{status)}, \quad O = \text{AWset(Seller} \times \text{Product)}, \\
E &= \text{AWset(Auction} \times \text{Seller)}, \quad W = \text{AWset(Auction} \times \text{Customer)}, \\
B &= \text{AWset(Bid} \times \text{Auction} \times \text{Customer} \times \mathbb{N}\text{)}, \quad L = \text{AWset(Lot} \times \text{Auction} \times \text{Seller} \times \mathbb{N}\text{)} \\
\mathit{Token} &= \{\, \tau_{r(a)}, \tau_{a(a)}, \tau_{c(a)}, \tau_{m(a)}, \tau_a \mid a \in \text{Auction}\} \cup \{\, \tau_{r(c)}, \tau_{a(c)} \mid c \in \text{Customer}\} \\
&\quad \{\, \tau_{r(p)}, \tau_{a(p)}, \tau_{p)} \mid p \in \text{Product}\} \cup \{\, \tau_{r(s)}, \tau_{a(s)}, \tau_{s(s)} \mid s \in \text{Seller}\} \\
\bowtie &= \{(\tau_{a(a)}, \tau_{r(a)}), (\tau_{r(a)}, \tau_{a(r)}), (\tau_{m(a)}, \tau_{c(a)}), (\tau_{c(a)}, \tau_{m(a)}), (\tau_a, \tau_a) \mid a \in \text{Auction}\} \\
&\quad \cup (\tau_{r(c)}, \tau_{a(c)}), (\tau_{a(c)}, \tau_{r(c)}) \mid c \in \text{Customer}\} \\
&\quad \cup (\tau_{r(p)}, \tau_{a(p)}), (\tau_{a(p)}, \tau_{r(p)}), (\tau_p, \tau_p) \mid p \in \text{Product}\} \\
&\quad \cup (\tau_{r(s)}, \tau_{a(s)}), (\tau_{a(s)}, \tau_{r(s)}), (\tau_{s(s)}, \tau_{s(s)}) \mid s \in \text{Seller}\} \\
\mathcal{F}_{\text{regCustomer}(c)}(\sigma) &= (\bot, (\lambda(\sigma').(\sigma'.C.\text{add}(c))), \emptyset) \\
\mathcal{F}_{\text{unRegCustomer}(c)}(\sigma) &= (\bot, (\lambda(\sigma'.C).(\sigma'.C.\text{remove}(c))), \{\tau_{r(c)}\}) \\
\mathcal{F}_{\text{regSeller}(s,n)}(\sigma) &= (\bot, (\lambda(\sigma'.S).(\sigma'.S.\text{add}(s,n))), \emptyset) \\
\mathcal{F}_{\text{unRegSeller}(s)}(\sigma) &= (\bot, (\lambda(\sigma'.S).(\sigma'.S.\text{remove}(s))), \{\tau_{r(s)}\}) \\
\mathcal{F}_{\text{addProduct}(p,m,s)}(\sigma) &= (\bot, (\lambda(\sigma'.P, \sigma'.O).(\sigma'.P.\text{add}(p,m), \sigma'.O.\text{add}(p,s))), \{\tau_{a(s)}\}) \\
\mathcal{F}_{\text{remProduct}(p)}(\sigma) &= (\bot, (\lambda(\sigma'.P, \sigma'.O).(\sigma'.P.\text{remove}(p), \sigma'.O.\text{remove}(p,s))), \{\tau_{r(p)}\}) \\
\mathcal{F}_{\text{ctrAuction}(a,s)}(\sigma) &= (\bot, (\lambda(\sigma'.A, \sigma'.E).(\sigma'.A.\text{add}(a, \text{open}), \sigma'.E.\text{add}(a,s))), \{\tau_{a(a)}, \tau_a\}) \\
\mathcal{F}_{\text{remAuction}(a)}(\sigma) &= (\bot, (\lambda(\sigma'.A, \sigma'.E).(\sigma'.A.\text{remove}(a), \sigma'.E.\text{remove}(a,s))), \{\tau_{r(a)}\}) \\
\mathcal{F}_{\text{addLot}(l,a,s,p,k)}(\sigma) &= (\bot, (\lambda(\sigma'.P, \sigma'.L).(\sigma'.P.\text{add}(p, \text{stock}(p, \sigma'.P) - k), \sigma'.L.\text{add}(l,a,s,p,k))), \{\tau_{a(a)}, \\
&\quad \tau_{m(a)}, \tau_{a(p)}, \tau_{a(s)}, \tau_p, \tau_{s(s)}\}) \\
\mathcal{F}_{\text{remLot}(l,a,s,p,k)}(\sigma) &= (\bot, (\lambda(\sigma'.P, \sigma'.L).(\sigma'.P.\text{add}(p, \text{stock}(p, \sigma'.P) + k), \sigma'.L.\text{remove}(l,a,s,p,k))), \{\tau_{m(a)}\}) \\
\mathcal{F}_{\text{placeBid}(b,a,c,v)}(\sigma) &= (\bot, (\lambda(\sigma'.B).(\sigma'.B.\text{add}(b,a,c,v))), \{\tau_{m(a)}, \tau_{a(a)}, \tau_{a(c)}\}) \\
\mathcal{F}_{\text{remBid}(b,a,c,n)}(\sigma) &= (\bot, (\lambda(\sigma'.B).(\sigma'.B.\text{remove}(b,a,c,v))), \{\tau_{m(a)}\}) \\
\mathcal{F}_{\text{startAuction}(a)}(\sigma) &= (\bot, (\lambda(\sigma'.A).(\sigma'.A.\text{add}(a, \text{active}))), \{\tau_{a(a)}, \tau_{m(a)}, \tau_a\}) \\
\mathcal{F}_{\text{closeAuction}(a,c)}(\sigma) &= (\bot, (\lambda(\sigma'.A, \sigma'.W).(\sigma'.A.\text{add}(a, \text{closed}), \sigma'.W.\text{add}(a,c))), \{\tau_{a(a)}, \tau_{m(a)}, \tau_a\})
\end{aligned}
$$

Precondition	Operation
true	$\mathcal{F}_{\text{regCustomer}(c)}(\sigma)$
$c \in \sigma.C \wedge \nexists a \in \text{Auction}, (b,a,c,n) \in \sigma.B \wedge (a,c) \in \sigma.W$	$\mathcal{F}_{\text{unRegCustomer}(c)}(\sigma)$
true	$\mathcal{F}_{\text{regSeller}(s,n)}(\sigma)$
$(s,.) \in \sigma.S \wedge \nexists p \in \text{Product}, a \in \text{Auction}, (s,p) \in \sigma.O \wedge (a,s) \in \sigma.E$	$\mathcal{F}_{\text{unRegSeller}(s)}(\sigma)$
$(s,n) \in \sigma.S$	$\mathcal{F}_{\text{addProduct}(p,m,s)}(\sigma)$
$(p,.) \in \sigma.P \wedge \nexists l \in \text{Lot}, a \in \text{Auction}, (l,a,p,n) \in L$	$\mathcal{F}_{\text{remProduct}(p)}(\sigma)$
$(s,n) \in \sigma.S$	$\mathcal{F}_{\text{ctrAuction}(a,s)}(\sigma)$
$\text{open}(a) \wedge \nexists b \in \text{Bid}, l \in \text{Lot}, (b,a,c,n) \in \sigma.B \wedge (l,a,p,k) \in \sigma.L$	$\mathcal{F}_{\text{remAuction}(a)}(\sigma)$
$\text{open}(a) \wedge (s,p) \in \sigma.O \wedge (a,s) \in \sigma.E \wedge k \leq \text{limit}(s) \wedge k \leq \text{stock}(p, \sigma.P)$	$\mathcal{F}_{\text{addLot}(l,a,s,p,k)}(\sigma)$
$\text{open}(a)$	$\mathcal{F}_{\text{remLot}(l,a,s,p,k)}(\sigma)$
$\text{active}(a) \wedge \exists c \in \text{Customer}, c \in \sigma.C$	$\mathcal{F}_{\text{placeBid}(b,a,c,v)}(\sigma)$
$\text{active}(a) \wedge \nexists c \in \text{Customer}, (a,c) \in \sigma.W$	$\mathcal{F}_{\text{remBid}(b,a,c,v)}(\sigma)$
$\text{open}(a) \wedge \exists l \in \text{Lot}, (l,a,p,k) \in \sigma.L \wedge k > 0$	$\mathcal{F}_{\text{startAuction}(a)}(\sigma)$
$\text{active}(a) \wedge \exists b \in bids, (b,a,c,v) \in \sigma.B \wedge b == \max(\sigma.B)$	$\mathcal{F}_{\text{closeAuction}(a,c)}(\sigma)$

Figure 6.9: Corrected auction application.

verify that this token assignment is indeed sufficient to ensure all invariants for such complex application.

Chapter 7

Related Work

Contents

7.1 Related Work 58
7.1.1 Consistency Models 58
7.1.2 Reasoning About Consistency in Distributed Systems and Databases . . . 60
7.2 Conclusion 61
7.3 Future Work 62

In this chapter, we present the related work, and conclude this part of thesis with a summery of CISE analysis, and developed tool and identify areas for future work.

7.1 Related Work

7.1.1 Consistency Models

Many replicated databases provide only eventual consistency [4, 100], because of the high latency of strong consistency protocols in wide-area networks. However, eventual consistency models expose applications to undesirable concurrency behaviour; they cannot guarantee application invariants. Several previous works aim at designing consistency models that provide meaningful semantics to the application, without compromising availability and incurring high latencies [6, 9, 10, 22, 32, 41, 70, 93]. However, the performance and availability benefits of these consistency protocols are still not well understood.

Haifeng et al. [105] propose three consistency metrics: unseen writes, uncommitted writes and staleness in order to measure consistency. The first metric determines the number of updates not seen by a replica. The uncommitted writes is the number of local updates not been seen by all replicas. The staleness indicates how recent a copy of data is compared to its most current version. Thus, an application can describe its inconsistency thresholds in terms of these three metrics. A replica executes operations of the application asynchronously if the thresholds stated by the application are satisfied. Otherwise, the operations must be synchronised for ensuring consistency promises. However, the three metrics are not expressive enough to capture all the dimensions of consistency required by an application. In particular, they do not consider application invariants.

Sovran et al. [93] propose a hybrid consistency model for key-value store, where some transactions execute under causal consistency and some under parallel snapshot isolation (PSI), a weaker form of snapshot isolation [44]. PSI weakens snapshot isolation semantics by allowing non-conflicting transactions execute in any mutual order among replicas. The hybrid model use two main ideas: preferred replicas and counting set. Each object is assigned to a preferred replica, which is responsible for handling all updates to the object. The counting set, called *cset*, provides commutative and concurrent semantics for set objects. An operation would execute asynchronously in its origin replica if, either the replica is the preferred replica of written objects by the operation, or the operation updates objects of cset type. Otherwise, the origin replica forwards the objects' updates to their preferred replicas using a two phase commit protocol. However, application developers must carefully reason about asynchronous operations in order to avoid invariant violations.

Lioyd et al. [70] provide a key-value geo-replicated system that offers stronger semantics than causal consistency, called $Causal^{+}$ consistency. Like causally-consistent systems, COPS delays updates on a data item until all dependencies are satisfied. Under causal consistency, concurrent users updating the same key in different replicas might observe different values

forever. *Causal*$^+$ extends causal consistency by ensuring state convergence among replicas. For this, the work exploits commutative merge functions, such as last-writer-win rules [54]. However, *causal*$^+$ consistency is still too weak to guarantee application invariants.

PNUTS [32] is a highly available (and scalable) relational database based on asynchronous replication. It provides per-key sequential consistency model to order updates to a key at all replicas. PNUTS relies on a multi-master replication schema, which forwards all updates to a primary replica. The primary replica propagates updates to other replicas in background (based on a centralised pub/sub mechanism). However, an update remains unavailable when a primary replica is unresponsiveness due to network partitions. PNUTS's API supports different primitive calls, in which different levels of consistency are possible. Each record is versioned, so that a desired version can be retrieved. Local reads might return stale data in favour of latency and availability. For write calls, PNUTS provides a test-and-set write interface with version numbers to update data only if it is not staler than the required version. However, PNUTS does not guarantee consistency among keys, as it might violate safety properties such as referential integrity rules.

Consistency rationing [59] provides a hybrid consistency model by categorising data items into three types (A, B and C); each with an appropriate consistency level. Category C contains data for which consistency violation is tolerable; operations on data C are fast and available even when disconnected; it ensures session consistency [96]. For instance, in a web shop application, log data is C, i.e., inconsistencies on log information are acceptable. Category A contains data that require system-wide synchronisation; it ensures strong consistency. In the web shop application, bank transactions must be serialisable, because the system needs to stop operations that would violate system correctness (e.g. negative balance). The consistency level of category B varies over time depending on specific policies including: cost, the probability of conflicts, time constraints and the availability of data. Returning to the example, product inventory is categorised as B. As long as the high amount of the product is available in stock, temporary inconsistency is allowed. Otherwise, it requires serialisability to avoid selling an item that is not in stock. However, categorising data is an error-prone task and difficult to design for and to test. Some complex data structures like streams either cannot be categorised or is too hard to categorise. Moreover, consistency rationing is conservative. Not all transactions over strong data require to be ordered. For instance, deposit operations in bank application are always safe, and hence, they can execute under weaker consistency models.

Li et al. [69] have proposed hybrid consistency model, called RedBlue, that adaptively tunes consistency model based on application requirements. To provide both performance (and availability) and consistency guarantees, RedBlue classifies operations as red and blue. Blue operations commute with all others; they execute asynchronously and quickly even when partitions occur; they ensure causal consistency. For example, in a bank application, deposit operation are blue, i.e., the user can add to his account in all circumstances. Red operations must be mutually

ordered, requiring system wide synchronisation; they ensure strong consistency. In the banking application, withdraw are red, because the system needs to stop a withdrawal that would make the balance negative. However, their model does not have a formal semantics.

7.1.2 Reasoning About Consistency in Distributed Systems and Databases

Li et al. [68] have presented static analysis in order to classify operations into synchronous and asynchronous operations in a hybrid consistency model. The analysis checks that if executing operations on causal consistency preserves a given integrity invariant; if not, the analysis concludes that the operations require synchronisation. However, the analysis does not check that whether the result of operation executions guarantee the invariant. In contrast, the CISE analysis allows to assign a set of tokens into operations and then reason about their correctness.

Bailis et al. [14] have proposed a necessary and sufficient condition, called $I-confluent$ analysis, to check whether operation executions on a replicated database need syntonisation or not. The work analyses various operations of an application, and its desired invariants in order to detect necessary synchronisation. However, the $I-confluent$ analysis is manual, and error-prone for verifying applications with complex and arbitrary invariants. Moreover, it does not address how to strengthen the consistency protocol in the case of invariant violations.

Sivaramakrishnan et al. [92] have proposed a static analysis that automatically maps application requirements to consistency levels in a replicated database. Consistency requirements of each application are captured by some contracts. Each contract specifies the fine-grained application consistency properties, such as the happens-before relation between operations. The analysis verifies that if operation executions satisfy the contracts under a given consistency protocol. However, the contracts are more low-level than invariants, and there is no guideline on how application developers write the right contracts for their application.

Lu et al. [68] have proposed correctness conditions under which transactions can be safely executed at a weaker isolation level than serialisability, such as snapshot isolation. The correctness criteria determines the appropriate isolation level for each type of application's transactions. Under snapshot isolation, read operations can execute concurrently with updates. A read operation may return a stale version of data. Since a single isolation level can be chosen for each transaction, the most conservative isolation level is chosen. In contrast, the CISE analysis focus on analysing and verifying an application running on a replicated database based on its invariants.

To reduce the synchronisation cost, Fekete [38] has proposed a hybrid consists model, where some transactions run under snapshot isolation, but others use two-phase locking for concurrency control. The hybrid model allows that transactions use local snapshots of the database on the replica, while still ensuring serialisable executions. He has proposed sufficient conditions based on analysing read-write and write-write conflicts between transactions in order to determine which transactions in an application need to execute under serialisability. The work shows how an application can be modified to satisfy the conditions. However, the analysis relies on serialisability

Application	#OP	#Tokens	#Invariant	Time(ms)
Bank	5	1	1	385
Escrow	4	2	1	187
Courseware	6	5	2	534
Auction	14	13	13	6542

Table 7.1: A summery of applications verified by CISE analysis

as the correctness criteria for executions of a set of transactions in a replicated database, whereas the CISE analysis allows weaker consistency models as long as the application invariants are preserved.

Roy et al. [89] have presented an analysis algorithm for extracting invariants from application code. Subsequently, they propose a consistency protocol that allows a replica to execute operations independently without any communications with other replicas, as long as the replica can meet a set of local conditions. If an operation cannot execute locally in the replica, a new set of conditions is recomputed using two-phase commit. The work is complementary to the CISE analysis, because the proposed techniques could be used to automatically infer invariants form code.

7.2 Conclusion

We proposed CISE analysis that helps programmers to check whether execution of a given application in replicated databases maintains the application invariants under a given synchronisation protocol. The CISE analysis includes three proof obligations: the *effector analysis* verifies that the operations' preconditions are sufficient for sequential correctness, the *commutativity analysis* ensures that the replicas converge, and the *stability analysis* verifies that preconditions are stable under concurrent updates. The total time complexity of the CISE analysis is $O(m^2)$, where m represents the number of operations. The *effector analysis* takes m steps, whereas *commutativity*, and *stability analysis* each takes m^2 steps, to be completed.

Following the proof rules, we have developed a SMT-based tool that automates the analysis. The tool verifies whether the current application semantics satisfies the CISE rules. If an obligation fails, the tool provides a counter-example, which the developer can use to understand the source of the problem and resolve it either by weakening the updates or invariants, or by adding some tokens to strengthen the synchronisation at the expense of reduced performance and availability.

Using the tool, we have verified several example applications: a bank application, an escrow data type, a courseware application, and an auction service. The tool has particularly verified applications using convergent replicated data types (CRDTs), which encapsulate techniques for merging the effects of operations without synchronisation cost. Table 7.1 summarises the applications verified and the time taken by the tool. The tool was run on a Mac Mini, 3 GHz Intel

Core i7. The numbers of operations and tokens are given without taking into account operation arguments, and tokens associated with different instances of the same object.

7.3 Future Work

The soundness of the CISE analysis relies on fact that the replication protocol must guarantee at least causal consistency. Although causal consistency can be implemented without synchronisation, its implementation entails to explicitly track causal dependencies. Weaker consistency models, which do not preserve causality for all operations, are widely used in many distributed systems. One direction of interest is to propose proof rules that allow to reason about integrity invariants for weaker consistency models.

To support the CISE analysis, one future research direction is to design and implement a replicated database, which uses the analysis. The system provides asynchronous replication by default, and integrates the CISE analysis in order to add extra synchronisations when it is necessary. The challenge of implementing such hybrid model is that synchronous operations may hinder the latency advantage of asynchronous operations.

Although the CISE tool is automatic, the later steps of identifying the sufficient tokens, and then to translate them into an efficient concurrency control protocol, are currently manual, which is tedious and error-prone. They open several avenues for improving the tool. One improvement would be to automate the analysis of counter-examples, in order to explore a correct token assignment.

In the future, we plan to automate the translation of the token assignments into an efficient concrete lock protocol. There are different ways to optimise a lock implementation, each with own cost and complexity. Examples include multi-level locks [71], lock coarsening [62], and early lock release [55]. For instance, each operation may be protected by its own fine-grain lock, at the high cost of acquiring and releasing locks. A well-known optimisation is to coarsen, replacing several fine-grain locks with a single coarse-grain one. Although a coarser lock reduces the synchronisation cost, it also delays (or blocks) concurrent updates, which costs performance too. From a performance perspective, there is no single best locking protocol, since this will depend on dynamic characteristics of the workload, namely on how often updates are blocked (contention) vs. how often locks are acquired (overhead). Combining static and dynamic analysis improves the reliability of system. A future research direction is to develop profiling or monitoring tools in order to measure the efficiency of the concurrency control protocol under different workloads, and heuristics to improve it.

Finally, verifying that the concurrency control protocol does not cause deadlock, through analysis, heuristics, and/or automated testing is another good direction.

Part III

Verifying and Co-designing File System Semantics

Chapter 8

A Scalable and Verified Design of a POSIX-Like File System

Contents

8.1 Motivation 66
8.2 Definitions and Database Model 66
8.3 A Formal Model of a Replicated File System Semantics 68
8.4 Correctness Criteria 71
8.5 Verifying Sequential Correctness of the File System 72
8.6 Replicated File System With Concurrency Control 73
8.7 Fully Asynchronous Replicated File System 75
8.7.1 Name Conflict 78
8.7.2 Remove/Update Conflict 79
8.8 Mostly Asynchronous Replicated File System 82

8.1 Motivation

Distributed file systems take advantage of *replication* to improve performance and be highly available by allowing operations to execute *concurrently* at different replicas. A user can access a file as long as at least one replica is available [48, 90]. Unfortunately, asynchronous replication faces the challenges of replica divergence and violations of invariant due to concurrent updates [81, 100]. These anomalies are undesirable for users, and pose an important challenge to the design of a replicated file system [63].

A conservative solution to this problem is to forbid concurrent updates. Before a replica accepts an update, the replica must synchronise its state with the others (synchronous replication). This approach ensures that operations execute in a global total order at all replicas, and offers applications a single common view of the distributed database [24, 25]. However, it forgoes availability and performance in order to achieve consistency [34].

Experience with real-world file systems has shown that concurrent updates to a shared file or directory occur infrequently [16, 57, 79, 99, 101]. Thus, a file system design relying on the conservative approach causes unnecessary synchronisation. A synchronisation is unnecessary if concurrent execution of operations ensures the application correctness properties.

In order to alleviate the tension between consistency, and availability or performance, we leverage the CISE analysis to adjust a file system design, either by weakening application semantics, and/or by adding concurrency control. The file system design exhibits a behaviour similar to the POSIX specification [83]. Its operational semantics resemble major POSIX commands used for creating, removing, and changing directory entries, as well modifying individual files. The main invariant that the specification must maintain is that the directory structure forms a *tree*.

We study three alternative semantics for the file system. Each exposes a different amount of parallelism, and different anomalies. Using our CISE tool, we check whether a specific file system semantics maintains the tree invariant. We first prove that the sequential execution of various operations of the file system preserves the tree invariant. Then, we extend the sequential semantics to support concurrent users. The underlying consistency model guarantees *causal consistency* by default. The commutativity and stability analyses enable us to verify each three semantics, and to derive an appropriate synchronisation protocol. Application of the CISE analysis confirms that our co-design approach is able to remove synchronisation for the common file system operations, while retaining a semantics reasonably similar to POSIX.

8.2 Definitions and Database Model

The abstract state of a file system consists of a naming tree of *directory*. A directory maps a locally-unique name $n \in \mathsf{Name}$, to a file system object, called a *node*. A node is either a directory

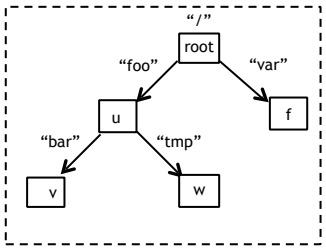


Figure 8.1: Example of a directory tree structure.

or a file,

$$\mathsf{Dir} : \mathsf{Name} \rightarrow \mathsf{Node}$$
$$\mathsf{Node} : \mathsf{File}|\mathsf{Dir}$$

Each node object is identified by a *path*. The path is a sequence of directory names, and possibly a final file name, separated by a separator or delimiter. Following the Unix convention, we use the "/" character as a separator. The origin of this hierarchical file system structure is a single *root* directory. The empty path holds the root of file system. A path is either *relative* or *absolute*. An absolute path starts from the root. A relative path is defined related to the current working directory. We use Greek letters for paths. For instance, in the directory tree shown in Figure 8.1, the path $\pi =$ "*/foo/bar*" is an absolute path representing directory object v, and the path $\pi =$ "*bar*" identifies directory v relative to directory u.

Every node, except the root, has a single *parent* directory.

Definition 8.1 (Parent Relation)**.** Directory u is direct parent of node v, denoted by $u \downarrow v$, if and only if u contains a mapping to v, i.e., there is a name $n \in \mathsf{Name}$, such that $(n, v) \in u$.

Assume that directory u is identified by path π, and node v is identified by path γ. The parent relationship implies that path π is the *longest prefix* of path γ. We call n the unique name of node v relative to the parent directory u.

Definition 8.2 (Path Prefix)**.** Path π is called a *prefix* of path γ, with notation $\pi \sqsubseteq \gamma$, if and only if $\gamma = \pi/\alpha$ for some path α.

The *transitive* closure of parent relation defines the *ancestor relation* in the tree hierarchy. We say directory u is an ancestor of node v, noted by $u \downarrow^+ v$, if and only if:

$$u \downarrow^+ v = \begin{cases} true & \text{if} \quad u \downarrow v \\ \exists w \in \mathsf{Dir}, u \downarrow w \wedge w \downarrow^+ v & otherwise \end{cases}$$

Definition 8.3 (Least Common Ancestor)**.** The Least Common Ancestor of nodes u and v, noted $LCA(u, v)$, is the ancestor of both u and v that is the lowest (i.e., deepest) node in the tree.

Given u's path π and v's path γ, $LCA(u, v)$ is the directory whose path is the *longest common* prefix of π and γ. For instance, the LCA of nodes v and w in Figure 8.1, is node u.

The root is a common ancestor of any pair of nodes.

Definition 8.4 (Validity)**.** Node u said to be *valid* or *reachable* iff the root is an ancestor of u in the tree, i.e., there is a path from the root to u.

A database state (D, F) of the file system consists of the set of directories D, and the set of files F. Each directory maps unique names to its children. We use notation $D.d[n \mapsto e]$ to update directory d in set D, mapping a name n to a node e in directory d, while keeping the other directories in set D unchanged. When the node e is removed (or moved) from directory d, the existing mapping for the node in directory d is removed, noted by $d[n \mapsto \emptyset]$. We also use the notation $F.f.\mathsf{content}(c)$ to update content of a file f in set F, writing c to file f, while keeping the other files in set F unchanged.

A user accesses a node by its path. We compute the path of a node by recursively following name of its ancestors using the parent relation up to the root.

Every node has a *type* assigned to it upon creation. The type defines that whether the node is a directory or a file. We assume a function create(*type*) to create a new and unique node object and add that to the database, where the argument *type* specifies its type. If *type* equals Dir, a new directory object without children is created in set D. If it equals File, an empty file object is created in set F.

8.3 A Formal Model of a Replicated File System Semantics

A large part of POSIX, the Portable *IEEE* Operating System Interface for computing environments, describes the file system. The file system semantics that we study in this chapter consists of a set of commands, which abstract major POSIX commands to manipulate the tree structure and to update file content. They include *creating*, *deleting*, and *renaming* directories or files. Users submit the following commands in a file system interface:

- mkdir(path)

  This command creates a new empty directory identified by the path argument. If the directory is created, the operation returns 0. Otherwise, it returns -1 to indicate an error, for example, attempting to add a directory that already exists is an error.

- rmdir(path)

  This command removes an empty directory, which is addressed by the specified path. The directory must not contain any files or sub-directories. Note, we ignore the special sub-directories . and .. that exist in the Unix file system. Upon successful completion, the directory is removed, and the operation returns 0. Otherwise, it returns -1, and the directory remains unchanged.

- mkfile(path)

  The POSIX *"creat(*path, mode*)"* command creates an open file descriptor referring to the file identified by the path argument. The open file descriptor is a record holding information that controls file accesses, such as the inode and current offset in the file. The mode argument specifies the access modes of the file. The *creat* command has many uses based on its access mode. We abstract the case of creating a file with read and write permissions with a mkfile(path) command. Upon successful execution, it returns a non-negative integer representing a file descriptor for the file, which is used by other I/O functions, such as *read* and *write* to refer to the file. The operation returns -1 on failure, and no file is created.

- write(fd, buf, nbyte)

  This command writes nbyte bytes from the buffer specified by buf argument to the file associated with the open file descriptor fd. Upon successful completion, the command updates the content of the file and returns the number of bytes actually written to the file. Otherwise, it returns -1.

- rmfile(path):

  The POSIX *"unlink(*path*)"* command removes a link to a file. The file is removed when all references to the file are removed, i.e., no process has the file open. We abstract the case of removing a file with a command rmfile(path). Upon successful execution, the operation returns 0, otherwise, it returns -1.

- mvfile(old, new)

  The POSIX *"rename(*old, new*)"* command has different meanings depending on the value of its arguments. We abstract the case where the old argument refers to a file to be moved, and the new argument refers to the new path for the file with a command mvfile(old, new). Upon successful completion, the file is removed from its old parent directory and added to its new parent directory. Otherwise, the operation returns -1, and the file's path does not change. The effect of the move operation is applied *atomically*, i.e., the file is located either in the old path or in the new path, never in both or neither of them.

- mvdir(old, new)

  Assume that the *old* argument of the POSIX rename command is path of a directory. We define another command mvdir(old, new), which resembles the semantics of the POSIX *rename(*old, new*)* command for moving the *source directory*, identified by the old argument, into the *destination directory*, identified by the new argument, either under the same name or different name. Upon successful completion, the source directory moves into the destination directory. Its content (files and sub-directories) is unaffected. Otherwise, the operation returns -1, and nothing changes. The effect of the move operation is applied

Command	Path Resolution	Effector
mkdir(path)	$\mathsf{path} = \pi/n \wedge d' = \mathcal{L}(\pi) \wedge d = \mathsf{create(Dir)}$	$\mathcal{F}_{\mathsf{mkdir}(d',n,d)}$
rmdir(path)	$\mathsf{path} = \pi/n \wedge d = \mathcal{L}(\mathsf{path}) \wedge d' = \mathcal{L}(\pi)$	$\mathcal{F}_{\mathsf{rmdir}(d',n,d)}$
mkfile(path)	$\mathsf{path} = \pi/n \wedge d = \mathcal{L}(\pi) \wedge f = \mathsf{create(File)}$	$\mathcal{F}_{\mathsf{mkfile}(d,n,f)}$
rmfile(path)	$\mathsf{path} = \pi/n \wedge f = \mathcal{L}(\mathsf{path}) \wedge d = \mathcal{L}(\pi)$	$\mathcal{F}_{\mathsf{rmfile}(d,n,f)}$
write(fd, buf, fd)	$\pi/n = \mathsf{getPath(fd)} \wedge c = \mathsf{buf} \wedge f = \mathcal{L}(\pi/n) \wedge d = \mathcal{L}(\pi)$	$\mathcal{F}_{\mathsf{write}(d,n,f,c)}$
mvfile(old, new)	$\mathsf{old} = \pi/n \wedge \mathsf{new} = \gamma/n \wedge f = \mathcal{L}(\mathsf{old}) \wedge d = \mathcal{L}(\pi) \wedge d' = \mathcal{L}(\gamma)$	$\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}$
mvdir(old, new)	$\mathsf{old} = \pi/n \wedge \mathsf{new} = \gamma/n \wedge d = \mathcal{L}(\mathsf{old}) \wedge d' = \mathcal{L}(\pi) \wedge d'' = \mathcal{L}(\gamma)$	$\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}$

Table 8.1: File system commands and their effectors.

atomically, i.e., the directory is located either in the old directory or in the new directory, never in both or neither of them.

Given the natural-language specification of the file system commands, we now present a formal model of the file system semantics. We assume a replicated file system consisting of N *replicas*. Each replica carries a *full* copy of the database state. A replica can fail by crashing, but it eventually recovers. A user can access and modify the file system state using the above commands. A command is initially submitted against the *origin* replica.

A file system command follows paths to access and modify files or directories. We assume that node identifiers are unique across replicas. To ensure that effector for a command produces the same effects as the original command in all replicas, we evaluate the path argument of the command against the database state σ at the origin replica in order to determine the nodes that it refers to. Given a path, the generator of each operation includes a *resolution* function $\mathcal{L}$ to find the node located in the path,

$$\mathcal{L} : \mathsf{Context} \times \mathsf{Path} \rightarrow \mathsf{Node}. \tag{8.1}$$

where the Context determines the starting lookup directory of the resolution function. If the path starts with the "/" character, i.e., it is an absolute path, the starting lookup directory is the root directory. Otherwise, the path is relative to the current directory specified by Context.

The effector takes the node determined by the generator as argument rather than its path in order to refer to the same node at all replicas despite concurrent changes in the node's path. Table 8.1 illustrates the file system commands and their corresponding effector. Function getPath(fd) returns the path of a file referred by the file descriptor *fd*.

For instance, consider that Alice accesses a shared directory located in path "*/share/album*". She wants to create a new directory in the shared directory using the command mkdir("*/share/album/paris*"). The path argument evaluates against Alice's replica to name "*paris*", its parent directory, which we will note d', and a new directory object d, which must be created. Thus, a corresponding effector $\mathcal{F}_{\mathsf{mkdir}(d',"paris",d)}$ is generated, and propagated to all replicas. On delivering the effector, a replica applies its effect, which is to create directory object d in directory

$$
\begin{aligned}
\mathsf{State} &= \mathsf{set}(\mathsf{Dir}) \uplus \mathsf{set}(\mathsf{File}) \\
\sigma_{init} &= (\{root\}, \emptyset) \\
\mathit{Token} &= \emptyset \\
\bowtie &= \emptyset \\
\mathcal{F}_{\mathsf{mkfile}(d,n,f)}((D,F)) &= (fd, \lambda(D',F').(D'.d[n \mapsto f], F' \cup \{f\}), \emptyset) \\
\mathcal{F}_{\mathsf{rmfile}(d,n,f)}((D,F)) &= (\text{"0"}, \lambda(D',F').(D'.d[n \mapsto \emptyset], F' \setminus \{f\}], \emptyset) \\
\mathcal{F}_{\mathsf{write}(d,n,f,c)}((D,F)) &= (nbytes, \lambda(D',F').(D', F'.f.\mathsf{content}(c)), \emptyset) \\
\mathcal{F}_{\mathsf{mkdir}(d',n,d)}(D,F)) &= (\text{"0"}, \lambda(D',F').(D' \cup \{d\} \cup D'.d'[n \mapsto d], F'), \emptyset) \\
\mathcal{F}_{\mathsf{rmdir}(d',n,d)}((D,F)) &= (\text{"0"}, \lambda(D',F').(D' \setminus \{d\} \cup D'.d'[n \mapsto \emptyset], F'), \emptyset) \\
\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}((D,F)) &= (\text{"0"}, \lambda(D',F').(D'.d[n \mapsto \emptyset] \cup D'.d'[n \mapsto f], F'), \emptyset) \\
\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}(D,F)) &= (\text{"0"}, \lambda(D',F').(D'.d'[n \mapsto \emptyset] \cup D'.d''[n \mapsto d], F'), \emptyset)
\end{aligned}
$$

Precondition	Operation
$\nexists e \in \mathsf{Node}, (n,e) \in d$	$\mathcal{F}_{\mathsf{mkfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{rmfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{write}(d,n,f,c)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d'$	$\mathcal{F}_{\mathsf{mkdir}(d',n,d)}(D,F)$
$root \downarrow^+ d$	$\mathcal{F}_{\mathsf{rmdir}(d',n,d)}(D,F)$
$root \downarrow^+ d$	$\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}(D,F)$

Figure 8.2: A simple file system application (incorrect).

set D, and to update the parent directory d' by mapping the name "*paris*" to directory d, denoted by $D.d'[\text{"}paris\text{"} \mapsto d]$.

Figure 8.2 presents the semantics of each effector function of the replicated file system. Note, for simplicity, we only consider the case, where a node moves into another location under the same name. For now, we assume that the set of token is empty.

8.4 Correctness Criteria

In this section, we describe the notion of correctness for the file system, beginning informally with some examples illustrating unintended behaviour, followed by more formal definition.

Consider Alice and Bob both access a shared file f at different replicas r_1 and r_2, respectively. Type of file f is *register* with operations to read and write the value of a register to the file. A write to the register rewrites its last successful written value. Alice writes a to the file. Concurrently, Bob writes b to the same file f. After exchanging the updates, the content of file f will be different at replicas r_1 and r_2. This violates the expectation of *convergence* to the same state. We can ensure that state converges by ensuring that every pair of concurrent operations to *commute*. For

Precondition	Operation
$\nexists e \in \mathsf{Node}, (n,e) \in d \wedge root \downarrow^+ d$	$\mathcal{F}_{\mathsf{mkfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{rmfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{write}(d,n,f,c)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d' \wedge d \downarrow f \wedge root \downarrow^+ f \wedge root \downarrow^+ d'$	$\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d' \wedge root \downarrow^+ d'$	$\mathcal{F}_{\mathsf{mkdir}(d',n,d)}(D,F)$
$\nexists e \in \mathsf{Node}, n' \in \mathsf{Name}, (n',e) \in d \wedge root \downarrow^+ d$	$\mathcal{F}_{\mathsf{rmdir}(d',n,d)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d'' \wedge root \downarrow^+ d \wedge d' \downarrow d \wedge root \downarrow^+ d'' \wedge d \not\downarrow^+ d''$	$\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}(D,F)$

Table 8.2: Corrected preconditions of file system operations after effector safety analysis.

instance, we can replace the register type with any CRDT type that merges concurrent updates (e.g. LWW register).

However, convergence alone is not sufficient. Towards what value the state converges is also important. Consider Alice creates directory d named n in parent directory d', and Bob concurrently deletes the parent directory d'. Each of these two operations are propagated to the other replicas. Although both replicas observe the same database state at the end, i.e., $d \in D \wedge d' \notin D \wedge D.d'[n \mapsto d]$, we know that the state is incorrect: there is a directory whose parent is removed. The database state must satisfy certain desirable file system properties, i.e., its invariants. The main invariant of the file system is that the directory structure forms a *tree*. The tree invariant I includes three main assertions: (1) the root is an ancestor of every node in the tree. (2) every node has exactly one parent except the root. (3) there is no *cycle* in the directory structure. The tree invariant formulated can be formulated as:

$$I = \forall e \in \mathsf{Node}, d, d' \in \mathsf{Dir}, (root \downarrow^+ e) \wedge (d \downarrow e \wedge d' \downarrow e \implies d = d' \wedge e \neq root) \wedge (d \downarrow^+ d' \implies d' \not\downarrow^+ d)$$

Any possible execution of file system operations must ensure that replicas converge, and the tree invariant is maintained. Otherwise, we say a *conflict* occurs.

8.5 Verifying Sequential Correctness of the File System

We apply the CISE effector safety to check that if file system operations illustrated in Figure 8.2 preserve the tree invariant in isolation. Unsuccessful analysis of an operation returns a counter-example, indicating that the operation's precondition is too weak. We leverage the counter-example to identify and solve the problem. We strengthen the precondition accordingly, and repeat the analysis until no counter-example is found.

For instance, the CISE tool found a counter-example for moving directory. The counter-example shows that the source directory must not be an ancestor of the destination directory; otherwise, a cycle would occur after moving the directory. We add the corresponding preconditions.

$$
\begin{aligned}
\mathsf{State} &= \mathsf{set(Dir)} \uplus \mathsf{set(File)} \\
\sigma_{init} &= (\{root\}, \emptyset) \\
\mathit{Token} &= \{\, \tau_e \mid e \in \mathsf{Node} \} \\
\bowtie &= \{(\tau_e, \tau_e) \mid e \in \mathsf{Node}\} \\
\mathcal{F}_{\mathsf{mkfile}(d,n,f)}((D,F)) &= (fd, \lambda(D',F').(D'.d[n \mapsto d], F' \cup \{f\},), \{\, \tau_d\}) \\
\mathcal{F}_{\mathsf{rmfile}(d,n,f)}((D,F)) &= (\text{"0"}, \lambda(D',F').(D'.d[n \mapsto \emptyset], F' \setminus \{f\}), \{\, \tau_f\}) \\
\mathcal{F}_{\mathsf{write}(d,n,f,c)}((D,F)) &= (i, \lambda(D',F').(D', F'.f.\mathsf{content}(c)), \{\, \tau_f\}) \\
\mathcal{F}_{\mathsf{mkdir}(d',n,d)}((D,F)) &= (\text{"0"}, \lambda(D',F').(D' \cup \{d\} \cup D'.d'[n \mapsto d], F'), \{\, \tau_{d'}\}) \\
\mathcal{F}_{\mathsf{rmdir}(d',n,d)}((D,F)) &= (\text{"0"}, \lambda(D',F').(D' \setminus \{d\} \cup D'.d'[n \mapsto \emptyset], F'), \{\, \tau_d\}) \\
\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}((D,F)) &= (\text{"0"}, \lambda(D',F').(D'.d[n \mapsto \emptyset] \cup D'.d'[n \mapsto f], F'), \{\, \tau_f, \tau_{d'}\}) \\
\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}((D,F)) &= (\text{"0"}, \lambda(D',F').(D'.d'[n \mapsto \emptyset] \cup D'.d''[n \mapsto d], F'), \\
&\qquad \{\, \tau_d, \tau_{d''}\} \cup \mathsf{tokens}(d, d'')) \\
\mathsf{tokens}(d, d'') &= \{\tau_e \mid e \in D, e \downarrow^+ d'' \wedge D.LCA(d, d'') \downarrow^+ e\}
\end{aligned}
$$

Precondition	Operation
$\nexists e \in \mathsf{Node}, (n,e) \in d \wedge root \downarrow^+ d$	$\mathcal{F}_{\mathsf{mkfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{rmfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{write}(d,n,f,c)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d' \wedge d \downarrow f \wedge root \downarrow^+ f \wedge root \downarrow^+ d'$	$\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d' \wedge root \downarrow^+ d'$	$\mathcal{F}_{\mathsf{mkdir}(d',n,d)}(D,F)$
$\nexists e \in \mathsf{Node}, n' : \mathsf{Name}, (n',e) \in d \wedge root \downarrow^+ d$	$\mathcal{F}_{\mathsf{rmdir}(d',n,d)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d'' \wedge root \downarrow^+ d \wedge d' \downarrow d \wedge root \downarrow^+ d'' \wedge d \not\downarrow^+ d''$	$\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}(D,F)$

Figure 8.3: Corrected file system application with mutually exclusive tokens.

Table 8.2 illustrates the sufficient preconditions of various file system operations verified by the CISE effector safety.

8.6 Replicated File System With Concurrency Control

A highly available distributed file systems entails that an operation is submitted to an origin replica, without coordination with remote replicas [15]. Thus, update operations can execute concurrently at different replicas and propagate asynchronously.

However, concurrent execution of operations may cause state divergence or the invariant violation. We use tokens in order to disallow toxic concurrent executions.

A developer may associate a set of tokens $T = \{\tau, \ldots\}$ with an operation to *explicitly* control concurrent operations. Operations that acquire tokens incompatible according to $\bowtie$ may not execute concurrently; the token implementation entails that the operations to be synchronised [18, 69].

To ensure the correctness criteria of the file system, we define a mutually exclusive token for each node $e \in \mathsf{Node} : \tau_e$, such that $\tau_e \bowtie \tau_e$. Conflicting operations must acquire the token τ_e before applying any changes over the node e.

Consider operation o that creates a new file f under a parent directory d, and operation o' that removes the directory d. Both operations must acquire token τ_d. The mutually exclusive token τ_d forbids these operations to execute concurrently, i.e., either the operation o is aware that the parent directory d has been removed, or the operation o' is aware that there is a file f added into the directory d; in either case the precondition of corresponding operation fails, and hence the operation cannot generate any effect.

We co-design the semantics of file system operations and their associated consistency requirements by adding tokens to operations appropriately as shown in Figure 8.3. Consider for instance, the effector $\mathcal{F}_{(\mathsf{mvdir}(d',n,d'',d))}$ that moves a directory d from directory d' to a destination directory d''. Its effect is to remove the mapping $n \mapsto d$ from d' and add the mapping to the destination directory d''. The operation is associated with tokens τ_d, and $\tau_{d''}$ over the source and destination directories, and a set of tokens τ_e, for all nodes e that are an ancestor of the destination directory d'' up to $LCA(d,d'')$. Concurrent move operations are allowed as long as their token are compatible.

Lemma 8.1. *Let d be a source directory, d'' be a destination directory, and A be the set of ancestors of the destination directory d'' up to $LCA(d,d'')$. $T = \{\tau_d, \tau_{d''}\} \cup \{\tau_e \mid e \in A\}$ represents necessary and sufficient tokens required by the* mvdir *operation.*

Proof. To show that set T is sufficient for maintaining the tree invariant when concurrent move operations are executing, we explain how the tokens in set T are able to preserve all three conditions of the tree invariant. The invariant implies that every node has exactly a parent. This condition is violated when concurrent operations move the source directory into different locations. Acquiring token τ_d in set T makes this impossible.

The invariant also requires that the root to be an ancestor of every node, and no cycle exists in the tree structure. However, when the source directory moves under itself, i.e., there is a concurrent operation moving the destination directory under the source directory, a cycle happens, and the nodes in the cycle will become disconnected from the root. To forbid such situation, a move operation must acquire tokens over the destination directory and its ancestors. However, it only needs to acquire tokens over ancestors of the destination directory up to the least common ancestor of the source and destination directory.

The intuition behind acquiring tokens over d'''s ancestors until the $LCA(d,d'')$ is: if a directory is a common ancestor of source directory d and destination directory d'', the directory cannot move under source directory d, i.e., it is forbidden by its precondition.

Using the CISE analysis, we show that the set of tokens and their incompatibility relation are indeed sufficient to prevent conflicts for concurrent move operations.

Now, we prove that T contains the minimal set of tokens by contradiction: We assume that T is not minimal, meaning that it includes unnecessary tokens. We remove a token $\tau \in T$, and then check whether concurrent executions of move operations still maintain the tree invariant. If so, set T is not minimal. We consider three cases:

1. τ is the token over the source directory d. Removing token τ from set T allows concurrent operations to moves the same directory d to another destination directory c. If $c \neq d''$, then the source directory d will have two parents; violating the tree invariant.

$$d'' \downarrow d \wedge c \downarrow d$$

2. τ is the token over destination directory. Removing token τ from set T allows another move operation that concurrently moves destination directory d'' to directory c. If $c = d$, or if directory c is a descendent of source directory d, i.e., $d \downarrow^+ c$, then cycles occur.

$$c \downarrow d'' \wedge d'' \downarrow d \wedge d \downarrow^+ c$$

3. τ is the token of one of ancestors, called c, of the destination directory d''. Removing token τ from set T allows another move operation to concurrently move directory c to directory e. If $e = d$, or if directory e is a descendent of source directory d, i.e., $d \downarrow^+ e$, then cycles occur.

$$e \downarrow c \wedge c \downarrow^+ d'' \wedge d'' \downarrow d \wedge d \downarrow^+ e$$

□

Figure 8.3 illustrates a correct semantics of the file system after adding all sufficient tokens and preconditions. The CISE analysis proves that the semantics results in a convergent state and maintains the tree invariant.

8.7 Fully Asynchronous Replicated File System

We cannot expect good performance and high availability from the synchronous file system design. Even if there is an efficient implementation of tokens, synchronisation remains a performance and availability bottleneck. This problem becomes worse when a replica is unavailable, for instance due to crash or a network failure, in which case operations are blocked because they cannot acquire their tokens. Moreover, experience with file accesses by typical users has shown that many files are only accessed by a single user [16, 79], and hence synchronisation may be unnecessary.

To avoid this synchronisation cost, one alternative approach is to *optimistically* accept all concurrent updates, and resolve conflicts. This trades sequential safety semantics for availability. There are three main approaches to resolve conflicts. A common policy is to accept one update of

Executions	Combined Effect (Add-Wins)	Combined Effect (Remove-Wins)
$\mathcal{F}_{\mathsf{mkfile}(d,n,f)}$	$D.d[n \mapsto f]$	$D.\mathsf{remove}(f) \cup D.\mathsf{remove}(d)$
$\parallel$	$\cup$	$\cup$
$\mathcal{F}_{\mathsf{rmdir}(d',n',d)}$	$D.d'[n' \mapsto d]$	$D.d'[n' \mapsto \emptyset]$
$\mathcal{F}_{\mathsf{write}(d,n,f,c)}$	$D.d[n \mapsto f]$	$D.\mathsf{remove}(f) \cup D.\mathsf{remove}(d)$
$\parallel$	$\cup$	$\cup$
$\mathcal{F}_{\mathsf{rmdir}(d',n',d)}$	$D.d'[n' \mapsto d]$	$D.d'[n \mapsto \emptyset]$
$\mathcal{F}_{\mathsf{mkdir}(d',n,d)}$	$D.d'[n \mapsto d]$	$D.\mathsf{remove}(d) \cup D.\mathsf{remove}(d')$
$\parallel$	$\cup$	$\cup$
$\mathcal{F}_{\mathsf{rmdir}(d'',n',d')}$	$D.d''[n' \mapsto d']$	$D.d''[n' \mapsto \emptyset]$
$\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}$	$D.d'[n \mapsto f]$	$D.\mathsf{remove}(f) \cup D.\mathsf{remove}(d')$
$\parallel$	$\cup$	$\cup$
$\mathcal{F}_{\mathsf{rmdir}(d'',n',d')}$	$D.d''[n' \mapsto d']$	$D.d''[n' \mapsto \emptyset]$
$\mathcal{F}_{\mathsf{rmdir}(d''',n',d'')}$	$D.d''[n \mapsto d]$	$D.\mathsf{remove}(d) \cup D.\mathsf{remove}(d'')$
$\parallel$	$\cup$	$\cup$
$\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}$	$D.d'''[n' \mapsto d'']$	$D.d'''[n' \mapsto \emptyset]$

Table 8.3: Combined effect of concurrent operations using different convergence semantics.

the two conflicting updates, and ignore the other, e.g., Thomas's write rule chooses the update with a higher timestamp and ignores older update [54]. For instance, AFS file system employs the *last-writer-wins* approach to resolve concurrent updates on the same file [66].

The second approach is that the replicated file system may support some application-specific resolution strategies. For example, some systems simply store the set of concurrently-written values of an object and it is up to application to resolve the conflict [35, 98]. This approach is used in replicated file systems such as LOCUS, Coda, Ficus, and Roam [57, 82, 85, 86].

Alternatively, the *database* itself may automatically resolve conflicting updates by exploiting object semantics. This assumes that the database has knowledge of resolution semantics in order to integrate the conflict resolution in the replication protocol.

The Fully Asynchronous File System exploits Conflict-Free Replicated Data Types (CRDT) [91] to design an asynchronous file system semantics that behaves similar to the semantics introduced in Section 8.3, and that converges by design. The file system specification includes two replicated sets: one for directories, and another for files. We rely on the specification of CRDT maps and sets presented in Chapter 3. A directory maps unique names to node objects; a node is either a directory or a file. A directory is implemented by a map. The CRDT map supports operations to add and remove nodes in a directory, and to query. When a user wants to add node e with name n to directory d that contains no node with the same name, the pair (n,e) is added to directory d using the map's function $d.\mathsf{add}(n,e)$. Removing node e deletes the mapping from

$$
\begin{aligned}
\mathsf{State} &= (\mathsf{AWset(Dir)} \uplus \mathsf{AWset(File)}) \times \mathsf{map(Node \times Dir)} \\
\sigma_{init} &= (\{(root, \mathsf{Unique-Tag})\}, \emptyset_{\mathsf{AWset}}, \emptyset) \\
Token &= \emptyset \\
\bowtie &= \emptyset \\
\mathcal{F}_{\mathsf{mkfile}(d,n,f)}((D,F,P)) &= (fd, \lambda(D',F',P').(D'.d.\mathsf{add}(n,f) \cup \mathsf{recursiveAdd}(\mathsf{ancestor}(d,P),D') \cup \\
&\quad \mathsf{update}(\mathsf{ancestor}(d,P),D',P'),F'.\mathsf{add}(f),P'[f \mapsto d]), \emptyset) \\
\mathcal{F}_{\mathsf{rmfile}(d,n,f)}((D,F,P)) &= (\text{"0"}, \lambda(D',F',P').(D'.d.\mathsf{remove}(n),F'.\mathsf{remove}(f),P'), \emptyset) \\
\mathcal{F}_{\mathsf{write}(d,n,f,c)}((D,F,P)) &= (nbytes, \lambda(D',F',P').(D'.d.\mathsf{add}(n,f) \cup \mathsf{recursiveAdd}(\mathsf{ancestor}(d,P),D') \cup \\
&\quad \mathsf{update}(\mathsf{ancestor}(d,P),D',P'),F'.\mathsf{add}(f) \cup F'.f.\mathsf{content}(c),P'), \emptyset) \\
\mathcal{F}_{\mathsf{mkdir}(d',n,d)}((D,F,P)) &= (\text{"0"}, \lambda(D',F',P').(\mathsf{addD}(d,D') \cup D'.d'.\mathsf{add}(n,d) \cup \\
&\quad \mathsf{recursiveAdd}(\mathsf{ancestor}(d',P),D') \cup \mathsf{update}(\mathsf{ancestor}(d',D),D',P'),F',P'[d \mapsto d']), \emptyset) \\
\mathcal{F}_{\mathsf{rmdir}(d',n,d)}((D,F,P)) &= (\text{"0"}, \lambda(D',F',P').(\mathsf{remD}(d,D'),F',P'), \emptyset) \\
\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}((D,F,P)) &= (\text{"0"}, \lambda(D',F',P').(D'.d'.\mathsf{add}(n,f) \cup \mathsf{recursiveAdd}(\mathsf{ancestor}(d',P),D') \cup \\
&\quad \mathsf{update}(\mathsf{ancestor}(d',D),D',P') \cup D'.d.\mathsf{remove}(f),F'.\mathsf{add}(f),P'[f \mapsto d']), \emptyset) \\
\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}((D,F,P)) &= (\text{"0"}, \lambda(D',F',P').(\mathsf{addD}(d,D') \cup D'.d''.\mathsf{add}(n,d) \cup \mathsf{recursiveAdd}(\mathsf{ancestor}(d'',D),D') \\
&\quad \cup \mathsf{update}(\mathsf{ancestor}(d'',P),D',P') \cup (A.d'.\mathsf{remove}(d),T),F',P'[d \mapsto d'']), \emptyset) \\
\mathsf{contents}() &= \{d \mid \exists i, (d,i) \in A \wedge (d,i) \notin T\} \\
\mathsf{addD}(d,(A,T)) &= (A \cup (d,i),T) \\
\mathsf{remD}(d,D) &= \text{if } (\exists e \in D.\mathsf{contents}(),(.,e) \in d \vee \exists f \in F,(.,f) \in d) \text{ then } (A,T) \\
&\quad \text{else } (A, T \cup (d,i)) \\
\mathsf{update}(S,D,P) &= (D.d'.\mathsf{add}(n',e) \mid e \in S \wedge e \notin D.\mathsf{contents}() \wedge d' = \mathsf{parent}(e,P) \wedge d'.\mathsf{query}(n') = e) \\
\mathsf{recursiveAdd}(S,(A,T)) &= T \setminus \{(e,i) \mid e \in S \wedge e \notin (A,T).\mathsf{contents}() \wedge (e,i) \in T\} \\
\mathsf{parent}(d,P) &= d' \mid P[d \mapsto d'] \\
\mathsf{ancestor}(e,P) &= \{e\} \cup \{d \mid d \downarrow_P^+ e\}
\end{aligned}
$$

Precondition	Operation
$\nexists e \in \mathsf{Node}, (n,e) \in d \wedge root \downarrow^+ d$	$\mathcal{F}_{\mathsf{mkfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{rmfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{write}(d,n,f,c)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d' \wedge d \downarrow f \wedge root \downarrow^+ f \wedge root \downarrow^+ d'$	$\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d' \wedge root \downarrow^+ d'$	$\mathcal{F}_{\mathsf{mkdir}(d',n,d)}(D,F)$
$\nexists e \in \mathsf{Node}, n' : \mathsf{Name}, (n',e) \in d \wedge root \downarrow^+ d$	$\mathcal{F}_{\mathsf{rmdir}(d',n,d)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d'' \wedge root \downarrow^+ d \wedge d' \downarrow d \wedge root \downarrow^+ d'' \wedge d \not\downarrow^+ d''$	$\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}(D,F)$

Figure 8.4: Asynchronous file system design using add-wins semantics.

directory d using the replicated map's function $d.\mathsf{remove}(n)$.

The Fully Asynchronous File System design must handle several conflict cases as a result of concurrent execution of operations. The remainder of this section will discuss these conflicts and

$$
\begin{aligned}
\mathsf{State} &= \mathsf{RWset(Dir)} \uplus \mathsf{RWset(File)} \\
\sigma_{init} &= (\{(root, \mathsf{Unique-Tag})\}, \emptyset_{\mathsf{RWset}}) \\
Token &= \emptyset \\
\bowtie &= \emptyset \\
\mathcal{F}_{\mathsf{mkfile}(d,n,f)}((D,F)) &= (fd, \lambda(D',F').(\mathsf{addF}(f,n,d,D',F')), \emptyset) \\
\mathcal{F}_{\mathsf{rmfile}(d,n,f)}((D,F)) &= (\text{"0"}, \lambda(D',F').(D'.d.\mathsf{remove}(n), F'.\mathsf{remove}(f)), \emptyset) \\
\mathcal{F}_{\mathsf{write}(d,n,f,c)}((D,F)) &= (nbytes, \lambda(D',F').(D', \mathsf{updateF}(f,c,F')), \emptyset) \\
\mathcal{F}_{\mathsf{mkdir}(d',n,d)}((D,F)) &= (\text{"0"}, \lambda(D',F').(\mathsf{addD}(d,n,d',D'), F'), \emptyset) \\
\mathcal{F}_{\mathsf{rmdir}(d',n,d)}((D,F)) &= (\text{"0"}, \lambda(D',F').(\mathsf{recursiveRem}(d,n,d',D')), \emptyset) \\
\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}((D,F)) &= (\text{"0"}, \lambda(D',F').(\mathsf{moveF}(f,n,d',d,D',F')), \emptyset) \\
\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}((D,F)) &= (\text{"0"}, \lambda(D',F').(\mathsf{moveD}(d,n,d'',D') \cup D.d'.\mathsf{remove}(d), F'), \emptyset) \\
\mathsf{addD}(d,n,d',D) &= \text{if } (d' \in D) \text{ then } (D.\mathsf{add}(d) \cup D.d'.\mathsf{add}(n,f)) \text{ else } D \\
\mathsf{addF}(f,n,d,D,F) &= \text{if } (d \in D) \text{ then } (D.d.\mathsf{add}(n,f), F.add(f)) \text{ else } (D,F) \\
\mathsf{updateF}(f,c,F) &= \text{if } (f \in F) \text{ then } F.f.\mathsf{content}(c) \text{ else skip} \\
\mathsf{recursiveRem}(d,n,d',D) &= \text{if } (\exists e \in D, n' \in \mathsf{Name}, (n',e) \in d) \text{ then } \mathsf{recursiveRem}(e,n',d,D) \\
&\quad \text{else } D.\mathsf{remove}(d) \cup D.d'.\mathsf{remove}(n) \\
\mathsf{moveD}(d,n,d',D) &= \text{if } (d' \in D) \text{ then } (D.d'.\mathsf{add}(n,d)) \text{ else } D.\mathsf{remove}(d) \\
\mathsf{moveF}(f,n,d',d,D,F) &= \text{if } (d' \in D) \text{ then } (D.d'.\mathsf{add}(n,f) \cup D.d.\mathsf{remove}(f), F) \\
&\quad \text{else } (D.d.\mathsf{remove}(f), F.\mathsf{remove}(f))
\end{aligned}
$$

Precondition	Operation
$\nexists e \in \mathsf{Node}, (n,e) \in d \wedge root \downarrow^+ d$	$\mathcal{F}_{\mathsf{mkfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{rmfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{write}(d,n,f,c)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d' \wedge d \downarrow f \wedge root \downarrow^+ f \wedge root \downarrow^+ d'$	$\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d' \wedge root \downarrow^+ d'$	$\mathcal{F}_{\mathsf{mkdir}(d',n,d)}(D,F)$
$\nexists e \in \mathsf{Node}, n' : \mathsf{Name}, (n',e) \in d \wedge root \downarrow^+ d$	$\mathcal{F}_{\mathsf{rmdir}(d',n,d)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d'' \wedge root \downarrow^+ d \wedge d' \downarrow d \wedge root \downarrow^+ d'' \wedge d \not\downarrow^+ d''$	$\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}(D,F)$

Figure 8.5: Asynchronous file system design using remove-wins semantics.

show how the file system design manages them.

8.7.1 Name Conflict

Users may perform concurrent updates to a directory. Concurrently adding or moving nodes under the same name in the same directory is problematic (*name conflict*).

To handle name conflicts, we choose the following merge semantics: concurrently adding or moving two nodes under the same name to the same parent directory merge these nodes. For directories, this means taking their union, and for files, this means merging their contents.

The Fully Asynchronous File System assumes that a file is also implemented by an object of some CRDT type. Thus, concurrent updates on the same file can be merged. The semantics of merging two files is given by the merge semantics of their type. We assume that the type of each file is embedded in its name. This ensures that only files of the same type need to be merged.

Consider replica r_1 creates directory d with name n in the root using command $\mathsf{mkdir}(root, n, d)$. Concurrently, another replica r_2 creates directory d' with the same name n using command $\mathsf{mkdir}(root, n, d')$. After observing both operations, the merge function creates a new directory d'', whose content is the union of contents of directories d and d'; ensuring that each name is mapped into only one directory.

However, a concurrent effector may still use the old directories. Figure 8.6 illustrates the problem in the context of the previous example. Replica r_3 observes directory d', and adds file f to directory d' using command $\mathsf{mkfile}(d', n', f)$, where $n \neq n'$. When the replica r_2 receives effector $\mathcal{F}_{\mathsf{mkfile}(d',n',f)}$, directory d' has been superseeded by d''. To solve this problem, each replica keeps a record of this following merge, in set G containing equivalent pairs, e.g., $(d', d'') \in G$. Thus, when a replica receives an effector with old directories, the replica queries set G to identify the merge directory. We define the merge function as follows:

$$\mathsf{merge}(d, d') = \lambda(D, G).(D.\mathsf{add}(d'') \cup D.d'''[n \mapsto d''], G[d \mapsto d'', d' \mapsto d'']) \tag{8.2}$$

where d'' is a new directory that merges two directories d and d' with the same name n under parent directory d'''. The CISE analysis shows that the merge function resolves the name conflict.

8.7.2 Remove/Update Conflict

A different kind of conflict happens when a replica updates a node, while another replica concurrently removes the node. This kind of conflict is called a *remove/update conflict*. For instance, when a replica receives an operation to add directory u to directory v, if directory v has been removed by a concurrent user, the operation execution results in an unreachable directory u.

The replicated data types support two main approaches, called *add-wins* and *remove-wins*, to address this problem. Both approaches implement the *combined effect* of non-commutative adds and removes. In the add-wins semantics, when there are concurrent add or remove of the same element, add wins and the effects of concurrent removes are ignored. Remove-wins follows the opposite semantics. When a node is removed, any concurrent adds of the same node are lost.

We propose two different replicated file system semantics based on these approaches. Table 8.3 shows the combined effect of conflicting adds (or moves) and remove operations. Since add-wins semantics does not lose the contents of a recently updated node, it is generally considered preferable.

Figure 8.5 illustrates the add-wins specification for the file system application. In the add-wins semantics of file system, set D and F each stores a pair of elements and unique tags, attached to each element. Each directory in set D is a AWmap that stores a set of (n, e) pairs,

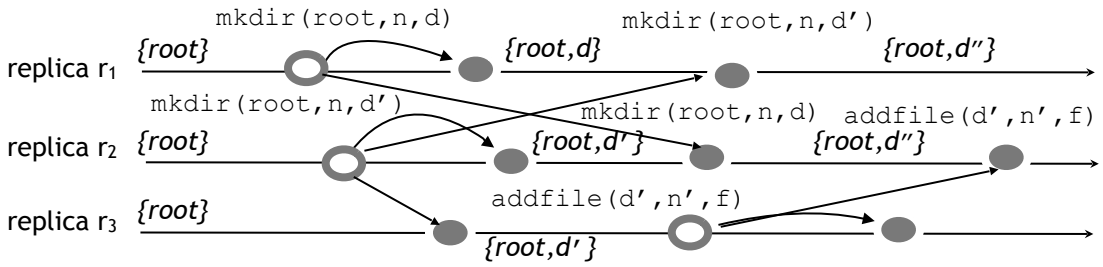


Figure 8.6: Example of concurrent creating directories with the same name.

where a name n is associated with a node e. Given a directory $d' \in D$, $D.d'$.add(n,d) denotes d' is updated by mapping n to d, i.e., adding directory d to d', while $D.d'$.remove(n) denotes d' is updated by removing the pair (n,d) from d', i.e., removing directory d from d'.

The add and remove functions of AWmap semantics, defined in Chapter 3, handle concurrent adds and removes of the same nodes in a directory. Concurrent adds commute since each one is unique. Concurrent removes commute because removing disjoint pairs has independent effects, and removing the same pair has the same effect. Concurrent $D.d'$.add(n,d) commute with $D.d'$.remove(n), i.e., the add wins because the unique tag generated by add cannot be observed by remove.

Writing to a file in a replica and concurrently removing the same file in another replica cause is also a remove/update conflict. The add-wins file system semantics re-creates the removed file. To implement this, each write to a file is considered as an add to the set F, so that it wins over a concurrent remove.

Now, consider the case of concurrent removing a directory and adding a node to the same directory. The add-wins specification of file system re-creates the missing directories. For instance, when directory u is added to directory v, which has been concurrently removed, the missing directory v is reinstated and then directory u is stored in directory v.

To implement this behaviour, the set directory D is represented by a pair of sets A and T. Set A is a set of active directory instance pair (d,i), and set T is a set of removed directory instance pair (d,i), where i is an unique tag for each added directory d. Function addD(d,D) adds an active instance of directory d to set A, and function remD(d,D) adds a removed instance of directory d to set T if there is no file or directory, which is concurrently added to directory d. A directory d is in the set D, i.e., is included in return value of contents, if there is an active instance of the directory in the set $D.A$, which does not exist in the tombstone set $D.T$.

We assume that set S contains all ancestors's of a node in the origin replica. To re-create an ancestor of a node, which has been concurrently removed, each replica stores a relation P that keeps the mapping information between nodes and their parent. For every node e, if any of its ancestors is removed, function recursiveAdd re-creates them by removing all removed instances of e's ancestors from set T. To make a valid path from the root to node e, function update in a

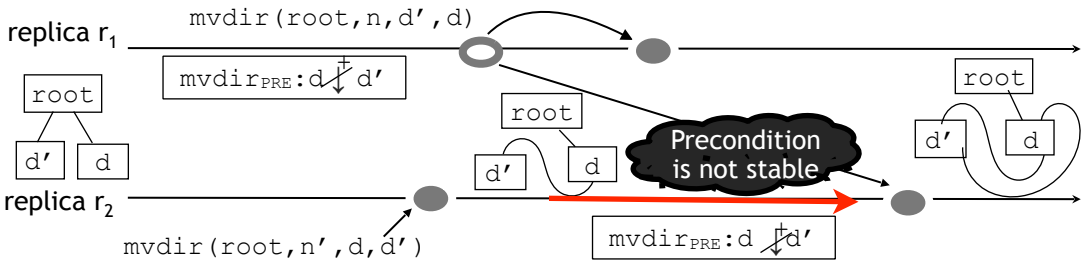


Figure 8.7: Counter-example for stability analysis of concurrent moves.

replica reads the relation P, and then adds e's ancestors to their parent.

For instance, consider a file system with the root, and a directory d, where d is located under the root. The database state is $(\{(d,i_2),(root,i_1)\},\{\},P[(d \mapsto root)])$. Replica r_1 adds a file f to directory d, concurrently replica r_2 removes directory d. The effector $\mathcal{F}_{\mathsf{mkfile}(d,n,f)}$ includes two functions update and recursiveAdd in order to re-create the full path of parent directory d if the directory is concurrently removed. Upon receiving the effector at replica r_2, function update adds directory d to the root again, and function recursiveAdd removes all instances of d from set T.

Concurrent moving and removing of the same node is addressed by considering each move operation as an add operation as it wins over a concurrent remove.

Figure 8.5 illustrates the remove-wins semantics for the file system application. Unlike the add-wins semantics, when there is concurrent update and remove operation on the same element, the remove operation wins and the update operation is lost. For instance, consider a directory d shared by Alice and Bob. Alice adds a file f to directory d in replica r_1, and concurrently Bob removes directory d in replica r_2. When Alice receives the removal, she applies the effector $\mathcal{F}_{(\mathsf{rmdir}(d',n,d))}$ that removes the directory d and file f. On the other hand, when Bob receives the Alice's add operation, he ignores adding file f because its parent directory d does not exist. Thus, the remove-wins semantics of file system entails recursive remove operation. We rely on the add and remove functions of a remove-wins set described in Chapter 3. To create a directory d, function addD(d,...) checks that if its parent directory exists, if so, directory d will be added to set D using the add function, otherwise, the directory is not created. To remove a directory in a replica, if the directory contains sub-directories or files as a result of performing concurrent add or move operations in the replica, the remove effector recursively removes the directory and all its content using the function recursiveRem. Thus, concurrent removes will win over adds. Concurrent moving a node and removing its destination directory addressed by considering each move operation as an add operation as the removal wins over a concurrent add, and hence the node will be removed.

We use the CISE analysis to verify the Fully Asynchronous File System design, considering both the add-wins and the remove-wins approaches. The analyser passes the sequential correctness analysis, verifying that all operation preconditions are sufficient to maintain the

tree invariant. The commutativity analysis verified that the concurrent operations results in a convergent state because all possible pairs of concurrent operations commute.

However, neither the add-wins semantics nor the remove-wins semantics passes the stability analysis when there are two concurrent *move* operations. Figure 8.7 illustrates a counter-example: Consider a file system with three directories, root, d and d', replicated at two replicas. Initially, the root is parent of d and d'. One replica asks to move directory d named n under directory d' using the move operation mvdir(root,n,d',d). The precondition of this *move* operation is true, i.e., the directory d is not an ancestor of directory d'. However, concurrently, other replica moves directory d' under directory d, and hence, the precondition of *move* is not true any more. And indeed, if we were to continue and apply effect of the first *move* operation, we come to the state, with a cycle of d and d', disconnected from the root. Obviously, it is not a tree.

8.8 Mostly Asynchronous Replicated File System

The application of the CISE analysis for the asynchronous file system design verifies that most operations of a replicated file system can execute without synchronisation, and only concurrent *move* operations may violate the tree invariant. The precondition of *move* directory operation is not *stable* when there is another concurrent *move* operation. If high performance and availability of update operations are important to the file system application, a simple approach to fix this issue is to allow operations to execute without restriction, and to repair the tree invariant violations after the fact. For instance, Microsoft One Drive [3] accepts all concurrent operations, and if cycles occur due to concurrent moving two directories d and d', it will lose both directories d and d' with all their contents. Other file systems such as Google Drive [2] or geoFS [95] exhibit other anomalies. The system using GeoFS duplicates all the directories in the cycle and the system using Google Drive puts all the directories in the cycle in root.

Our Mostly Asynchronous File System chooses the alternative approach: to add synchronisation in order to avoid concurrent execution of *move* operations that would violate the tree invariant. Thus, we co-design a file system semantics, in which the common operations run in asynchronous mode, and only some move directory operations need synchronisation.

The CISE analyser helps us to identify which pairs of the *move* operations are conflicting, i.e., whose concurrent execution violates the tree invariant.

To ensure that cycles do not happen, we use a pair of incompatible tokens for each directory $d \in D$: $\tau_{s(d)}$ and $\tau_{d(d)}$, called *source* and *destination* tokens, respectively. This is equivalent to associating every directory with a multi-level lock [18] that can be in one of two modes. Each mode restricts executing some operations. Given a directory d, source token $\tau_{s(d)}$ disallows to move directories in to d, and destination token $\tau_{d(d)}$ disallows to move directory d itself. Neither token $\tau_{s(d)}$ nor $\tau_{d(d)}$ is incompatible with itself.

Assume that a client wishes to move a source directory d to a destination directory d'. Let the

$$
\begin{aligned}
\mathsf{State} &= (\mathsf{AWset(Dir)} \times \mathsf{AWset(File)}) \times \mathsf{map(Node} \times \mathsf{Dir)} \\
\sigma_{init} &= (\{(root, \mathsf{Unique-Tag})\}, \emptyset_{\mathsf{AWset}}, \emptyset) \\
Token &= \{\, \tau_e, \tau_{s(d)}, \tau_{d(d)} \mid e \in \mathsf{Node}, d \in \mathsf{Dir}\} \\
\bowtie &= \{(\tau_e, \tau_e), (\tau_{d(d)}, \tau_{s(d)}), (\tau_{s(d)}, \tau_{d(d)}) \mid e \in \mathsf{Node}, d \in \mathsf{Dir}\} \\
\mathcal{F}_{\mathsf{mkfile}(d,n,f)}((D,F,P)) &= (fd, \lambda(D',F',P').(D'.d.\mathsf{add}(n,f) \cup \mathsf{recursiveAdd}(\mathsf{ancestor}(d,P),D') \cup \\
&\quad \mathsf{update}(\mathsf{ancestor}(d,P),D',P'), F'.\mathsf{add}(f), P'[f \mapsto d]), \emptyset) \\
\mathcal{F}_{\mathsf{rmfile}(d,n,f)}((D,F,P)) &= (\text{"0"}, \lambda(D',F',P').(D'.d.\mathsf{remove}(n), F'.\mathsf{remove}(f), P'), \emptyset) \\
\mathcal{F}_{\mathsf{write}(d,n,f,c)}((D,F,P)) &= (nbytes, \lambda(D',F',P').(D'.d.\mathsf{add}(n,f) \cup \mathsf{recursiveAdd}(\mathsf{ancestor}(d,P),D') \cup \\
&\quad \mathsf{update}(\mathsf{ancestor}(d,P),D',P'), F'.\mathsf{add}(f) \cup F'.f.\mathsf{content}(c), P'), \emptyset) \\
\mathcal{F}_{\mathsf{mkdir}(d',n,d)}((D,F,P)) &= (\text{"0"}, \lambda(D',F',P').(\mathsf{addD}(d,D') \cup D'.d'.\mathsf{add}(n,d) \cup \\
&\quad \mathsf{recursiveAdd}(\mathsf{ancestor}(d',P),D') \cup \mathsf{update}(\mathsf{ancestor}(d',D),D',P'), F', P'[d \mapsto d']), \emptyset) \\
\mathcal{F}_{\mathsf{rmdir}(d',n,d)}((D,F,P)) &= (\text{"0"}, \lambda(D',F',P').(\mathsf{remD}(d,D'), F', P'), \emptyset) \\
\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}((D,F,P)) &= (\text{"0"}, \lambda(D',F',P').(D'.d'.\mathsf{add}(n,f) \cup \mathsf{recursiveAdd}(\mathsf{ancestor}(d',P),D') \cup \\
&\quad \mathsf{update}(\mathsf{ancestor}(d',D),D',P') \cup D'.d.\mathsf{remove}(f), F'.\mathsf{add}(f), P'[f \mapsto d']), \{\, \tau_f\}) \\
\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}((D,F,P)) &= (\text{"0"}, \lambda(D',F',P').(\mathsf{addD}(d,D') \cup D'.d''.\mathsf{add}(n,d) \cup \mathsf{recursiveAdd}(\mathsf{ancestor}(d'',D),D') \\
&\quad \cup \mathsf{update}(\mathsf{ancestor}(d'',P),D',P') \cup (A.d'.\mathsf{remove}(d), T), F', P'[d \mapsto d'']), \\
&\quad \{\, \tau_d, \tau_{s(d)}, \tau_{d(d)}\} \cup \mathsf{tokens}(d,d'')) \\
\mathsf{contents}() &= \{d \mid \exists i, (d,i) \in A \wedge (d,i) \notin T\} \\
\mathsf{addD}(d,(A,T)) &= (A \cup (d,i), T) \\
\mathsf{remD}(d,D) &= \text{if } (\exists e \in D.\mathsf{contents}(), (.,e) \in d \vee \exists f \in F, (.,f) \in d) \text{ then } (A,T) \\
&\quad \text{else } (A, T \cup (d,i)) \\
\mathsf{update}(S,D,P) &= (D.d'.\mathsf{add}(n',e) \mid e \in S \wedge e \notin D.\mathsf{contents}() \wedge d' = \mathsf{parent}(e,P) \wedge d'.\mathsf{query}(n') = e) \\
\mathsf{recursiveAdd}(S,(A,T)) &= T \setminus \{(e,i) \mid e \in S \wedge e \notin (A,T).\mathsf{contents}() \wedge (e,i) \in T\} \\
\mathsf{parent}(d,P) &= d' \mid P[d \mapsto d'] \\
\mathsf{ancestor}(e,P) &= \{e\} \cup \{d \mid d \downarrow_P^+ e\} \\
\mathsf{tokens}(d,d'') &= \{\tau_{d(e)} \mid e \in \mathsf{Dir} \wedge e \downarrow^+ d'' \wedge LCA(d,d'') \downarrow^+ e\}
\end{aligned}
$$

Precondition	Operation
$\nexists e \in \mathsf{Node}, (n,e) \in d \wedge root \downarrow^+ d$	$\mathcal{F}_{\mathsf{mkfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{rmfile}(d,n,f)}(D,F)$
$root \downarrow^+ f$	$\mathcal{F}_{\mathsf{write}(d,n,f,c)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d' \wedge d \downarrow f \wedge root \downarrow^+ f \wedge root \downarrow^+ d'$	$\mathcal{F}_{\mathsf{mvfile}(d,n,d',f)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d' \wedge root \downarrow^+ d'$	$\mathcal{F}_{\mathsf{mkdir}(d',n,d)}(D,F)$
$\nexists e \in \mathsf{Node}, n' : \mathsf{Name}, (n',e) \in d \wedge root \downarrow^+ d$	$\mathcal{F}_{\mathsf{rmdir}(d',n,d)}(D,F)$
$\nexists e \in \mathsf{Node}, (n,e) \in d'' \wedge root \downarrow^+ d \wedge d' \downarrow d \wedge root \downarrow^+ d'' \wedge d \not\downarrow^+ d''$	$\mathcal{F}_{\mathsf{mvdir}(d',n,d'',d)}(D,F)$

Figure 8.8: Mostly asynchronous and corrected file system (add-wins).

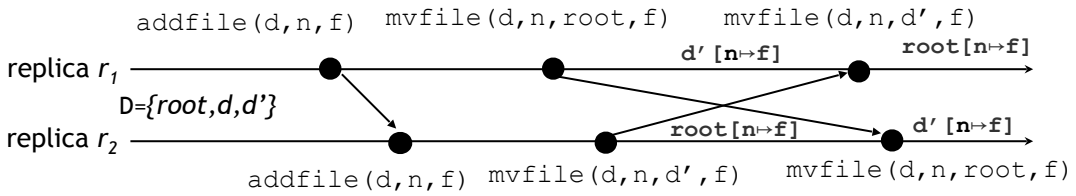


Figure 8.9: Counter-example: the parent relation anomaly.

set $A.d'$ contain all directories that are d''s ancestors up to the least common ancestor of d and d', noted $LCA(d,d')$. In order to move directory d under directory d', it is necessary to acquire d's source token, $\tau_{s(d)}$, d''s destination token, $\tau_{d(d')}$, and the destination tokens for all directories in set $A.d'$. For instance, in Figure 8.7, operation mvdir($root,n,d',d$) acquires tokens $\{\tau_{s(d)},\tau_{d(d')}\}$, and operation mvdir($root,n',d,d'$) acquires the tokens $\{\tau_{s(d')},\tau_{d(d)}\}$. Their token sets are not compatible, token $\tau_{s(d')}$ is not compatible with token $\tau_{d(d')}$, and token $\tau_{s(d)}$ is not compatible with $\tau_{d(d)}$, i.e., $\bowtie = \{(\tau_{s(d')},\tau_{d(d')}),(\tau_{s(d)},\tau_{d(d)})\}$. Therefore, the program execution of Figure 8.7 cannot occur.

For any pair of *move* operations, if their tokens are incompatible, only one of them can take effect, because token semantics requires that they exchange messages, which ensure that one of the operations is aware of the other. However, other move operations are causally independent, and hence can proceed in parallel.

We add the corresponding tokens to the move semantics, and perform the CISE stability analysis again.

This time, the tool generates a counter-example that indicates that two concurrent users might move the same node to different locations. Thus, the node would end up with two parent directories; violating the tree invariant. The counter-example is as follows: Initially, the file system has three directories: $D = \{root,d,d'\}$. Replica r_1 adds file f to directory d. Replica r_2 observes file f, and later moves f in to directory d'; concurrently, replica r_1 moves file f in to the root. After exchanging the updates, the file f will have two parents, as shown in Figure 8.9.

To avoid this issue of moving the same node concurrently, we create a new token type, move token. We assign an exclusive move token τ_e for each node e. A move operation must acquire the token τ_e. For instance, the *move* operation $\mathcal{F}_{(mvDir(d',n,d'',d))}$, acquires the mutually exclusive token τ_d before its execution.

Figure 8.8 illustrates the replicated file system semantics using the hybrid consistency model. It relies on CRDTs to ensure convergent state, and use the move tokens to ensure the tree invariant. Unlike the synchronous semantics, only move operations require tokens. The semantics successfully passes all three CISE analyses. The analyser proves that the consistency choices for different operations are sufficient to preserve the tree invariant, while provides an

acceptable behaviour similar to POSIX.

Chapter 9

Related Work

Contents

9.1 Related Work 88
9.1.1 Formal Reasoning about File Systems 88
9.1.1.1 First-Order Logic Reasoning 88
9.1.1.2 Separation Logic Reasoning 89
9.1.2 Conflict Resolution in File systems 90
9.2 Conclusion 91
9.3 Future Work 91

In this chapter, we discuss some related work, and conclude this part of thesis with a summary of findings of the application of CISE analysis in the design of a replicated file system, and identify areas for future work.

9.1 Related Work

9.1.1 Formal Reasoning about File Systems

There has been substantial work on formal specification of file systems in different specification languages, which are based on either first-order logic, or separation logic [87].

9.1.1.1 First-Order Logic Reasoning

A number of formalisations of file systems have been proposed using first-order logic [39, 40, 58, 75, 104]. Most of them focus only on primitive file I/O operations, such as reading and writing file content [58, 75]. Morgan and Sufrin [75] have proposed a formal specification for a UNIX file system using the Z notations [94], which is later proved correct [39]. The formal model represents each file system object by its path, i.e., a sequence of directories names, and defines operations to create files and update their contents. However, the specification model does not completely cover all POSIX behaviour, so that it does not consider the structure manipulation operations, such as move directories. In addition, users can access shared files without any permission controls. Similarly, Arkoudas et al. [58] have proved the correctness of read and write operations for a basic file system implementation using Athena, an interactive theorem prover. Given a simple file system implementation, Athena constructs 283 lemmas and theorems in order to verify the isolation of reading and writing files in a directory.

Huges [53] has specified a visual file system using the Z notations [94]. He focuses on modelling of a hierarchical file system, so that his model covers basic operations affecting the tree structure, including move and remove directories. However, his specification does not consider the no-loop property, it only takes transitive closure (i.e., reachability) as the main property of a tree structure. Inspired by Hughes's specification, Kriangsak et al. [60] have formalised and proved a tree-structured file system by using Event-B and Rodin platform [5]. Like our specification, their model is based on acyclic directory structure. A set of permissions are attached to an object, so that accesses to the object depends on the permissions allowed. The Rodin toolset generates 162 proof obligations to verify the specification model. Hesselink [104] has introduced an alternative approach to formulate the file structure using partial functions from paths to data.

Experiences with formal specification and verification of file systems show that first-order logic reasoning is adequate for high-level specification, and implementation of real file systems. However, the first-order logic does not scale well when reasoning about operation executions of a POSIX file system [77]. The POSIX English specification defines a set of preconditions for each operation, which must be satisfied before its execution. For instance, moving a source

directory into a destination directory takes effect, if the source directory is not an ancestor of the destination directory. Encoding such conditions using first-order logic entails many proof obligations and constraints that increase non-linearly with respect to the size of programs [77].

9.1.1.2 Separation Logic Reasoning

Recent work on file system verification relies on separation logic. Chen et al. [49] have introduced Crash Hoare Logic (CHL) for developing and verifying sequential and fault-tolerant file systems. The CHL logic checks whether a storage system implementation will recover to a state consistent with its specification after a failure. Using the analysis, the authors specified and verified FSCQ, a crash-safe user-space file system implemented in Haskell. The FSCQ's interface consists of a series of Hoare triples over high-level operations. The specification model of FSCQ relies on the separation logic to reason about operations at different level of abstractions including disk, files, directories, and logical disk. FSCQ uses a write-ahead log for failure recovery. The CHL analysis proved that the write-ahead log guarantees atomicity of updates by adding fault-conditions into the Hoare triples.

Ernst et al. [37] have presented a formal POSIX model, and verified that if a heap-based implementation of of Virtual File system Switch (VFS) meets the POSIX specification model. They take advantage of separation logic to map directories to pointer structures of VFS in order to reason about the implementation. The work introduces proof obligations by symbolic execution of operations, and use an interactive theorem prover for discharging the proof obligations. Their rules check that if any possible behaviour (output) of VFS operations is captured by the POSIX specification model.

Biri and Galmiche [26] have proposed a separation logic rule for trees and local reasoning over global paths. However, their simple tree model forbids structural's modifications, as neither new nodes can be created nor nodes can be moved, i.e., the tree structure is static.

Gardner et al. [42] have proposed a formal model of POSIX file system based on separation logic. The semantic of POSIX operations are captured with preconditions and postconditions in a Hoare-logic style. Some permissions are associated into each operation to control access to shared paths. Before applying an update, the necessary permissions must be obtained in order to ensure that effect of the update is propagated to entries whose path may overlap. However, the specification model does not support concurrent POSIX users.

Our co-design approach uses CISE logic, which is based on rely/guarantee to reason about a given file system semantics. Although, we focus on the tree structure of file system, the proof rules are not limited to the POSIX model, so that they can be applied to any file systems with different invariants. Existing specifications of file systems are mostly based on its sequential behaviour, and hence, using them to reason about concurrent operations is far from trivial. Concurrency is an essential part of a replicated file system. Concurrent updates may cause consistency issues that must be addressed. For instance, describing an operation, which creates a new directory in

some parent directory is trivial, but specifying how to deal with two concurrent operations, which add two different directory under the same under to the same parent directory is much harder. Our specification model co-designs the semantics of file system's operations and the consistency requirements. Using the CISE analysis, we were able to verify the file system semantics in polynomial time.

9.1.2 Conflict Resolution in File systems

Semantic-based concurrency control has been studied extensively in Database community. Several works have explored the semantics of applications (and data types), including commutativity and invariants, to design more scalable system and amortize locks [12, 103].

Clements et al. [31] have proposed a cache conflict-free implementation of POSIX file system on a shared-memory multiprocessor system. They explore the commutativity of POSIX operations to design a scalable file system implementation. They have presented an analyser, called COMMUTER, which checks the commutativity of POSIX operations. COMMUTER relies on symbolic executions for program testing. A symbolic model tests all permutations of operations, and computes necessary conditions under which those operations commute. Using the commutativity conditions, they modify the POSIX semantics. COMMUTER generates different test cases to verify the semantics in a real implementation. However, they focus only on scalability, not on the safety of executing commutative operations; they do not check that if the commutative operations maintain the tree invariant.

Balasubramaniam and Pierce [17] have proposed an optimistic files system replication model from a semantics perspective. Causally-dependent operations are ordered according into happen-before relation, while concurrent operations may be executed at any orders. Concurrent updates on the same directory are allowed if they do not conflict. For instance, concurrent users can add different files with different names to the same directory, but if one user modifies a file, and another deletes its parent directory, a conflict happens. The model requires users to manually resolve conflicts. This specification model was later formalised and proved by Ramsey and Csirmaz [84]. However, the operation-based model is limited i.e., the algebra model contains 51 different rules for few operations, including create, remove, and edit. It is not clear how one can extend the model to support more complex operations, such as move operations involving different directories. In addition, the model does not check the tree invariant; it is difficult to describe acyclic property by using their model.

Bjørner et al. [28] have proposed a replicated file system reconciler (DFS-R) that automatically resolve conflicts when they arise. They use model checkers to verify the conflict resolution strategies. Similar to our CISE-enabled tool, the analysis gives a counter-example for concurrent moves, meaning that concurrent move operations do not maintain directory hierarchies as tree-like structure. However, they do not address how to add synchronisation when the tree invariant is violated.

9.2 Conclusion

The inherent complexity of file systems demands automated techniques for understanding and reasoning about correctness in concurrent environments. In this work, we formalised and verified a replicated file system that supports most primitives POSIX commands. The specification of the file system focused on a tree structure and basic operations modifying the tree structure: create, delete, and move nodes that can be files or directories. In the specification model, we focused on the three main properties of a tree structure: (1) every node in a tree is reachable from the root (2) every node, except the root, has a single parent (3) there are no cycles in the tree structure.

Our main contribution is a case study of the application of the CISE analysis for designing an efficient file system semantics that provides a behaviour similar to POSIX at reasonable cost. The CISE analysis allowed us to concisely and precisely reason about a given file system semantics. We presented three different semantics of file system, each provides different levels of parallelism and anomalies. Using the co-design approach, we were able to remove synchronisation on most operations while retaining a semantics reasonably similar to POSIX. Applying the CISE analysis proved that the precondition of a move operation is not stable under another concurrent move operation: it follows that no file system can support an unsynchronised move without anomalies, such as loss or duplication.

Thanks to CISE's proof automation tool, the average running time to verify a given file system semantic was 1297 ms. The challenge of file system verification using the SAT solver was to translate reachability property because the SMT solver does not support any built-in transitive closure operator. We employed the tactics and strategies proposed in [67] and [36] to incorporate the reachability property in the context of the SMT solver.

9.3 Future Work

There are several avenues for future work from both verification and performance perspective. First, the file system verification based on CISE analysis requires that the database guarantees causal consistency. However, causal consistency does not sale week, as many file systems opt for weaker consistency models, such as eventual consistency. We plan to propose proof rules for weaker models where causality preservation is not mandatory for operation executions.

Second, we are going to implement the three file system semantics to compare their actual performance under real workloads. The plan is to integrate our findings from the CISE analysis into a highly-scalable geo-replicated file system. The challenge is to translate the tokens into an efficient concurrency control protocol. There are different ways to implement tokens in the file system hierarchy. A simple approach is to represent each token by a lock, so that each operation acquires the corresponding locks when called, and releases them when returning. However, it incurs a significant synchronisation cost. One optimisation is lock coarsening, where locks are acquired on entire directory when any node in the directory is accessed. Although a coarser lock

reduces the synchronisation cost, it also delays concurrent updates, which costs performance too. We are looking for dynamic and heuristic analysis that allow to measure and improve the token implementations.

Finally, the CISE analysis only verifies the correctness of the file system against concurrent executions. In the future, we plan to propose proof rules that allow developers to reason about the operation executions in the presence of replica and network failures. Thus, programmers would be able to prove that a file system specifications model handles properly any possible faults. This entails formalisation of failure models, as the specification of the file-system API captures its semantics under crashes.

Part IV

Safe Applications on the Cheap with Invariant Patterns

Chapter 10

Efficiently Implementable Patterns of Invariants

Contents

10.1 Classes of Invariant . . . 97
10.2 Generic Invariants on a Single Item (Gen1) . . . 98
10.2.1 Protocols and Mechanisms for Gen1-Invariants . . . 98
10.2.2 Total-Order . . . 99
10.3 EQ Invariants . . . 100
10.3.1 Protocols and Mechanisms for EQ-Invariants . . . 100
10.3.2 EQ Invariants and Convergence Resolutions . . . 101
10.4 PO Invariants . . . 101
10.4.1 Protocols and Mechanisms for PO-invariants . . . 102
10.5 Composite Invariant Patterns . . . 103
10.5.1 Composing Gen1-Invariants . . . 103
10.5.2 Composing Gen1-Invariant with EQ-Invariant . . . 104
10.5.3 Composing Gen1-Invariant with PO-Invariant . . . 104
10.5.4 Composing EQ-Invariants . . . 104
10.5.5 Composing EQ-Invariant with PO-Invariant . . . 105
10.5.6 Composing PO-Invariants . . . 105
10.6 Consistency Models . . . 106
10.6.1 Strict Serialisability (SSER) . . . 106
10.6.2 Serialisability (SER) . . . 106
10.6.3 Snapshot Isolation (SI) . . . 107
10.6.4 Causal Consistency (CC) . . . 108
10.6.5 Eventual Consistency (EC) . . . 108

10.6.6 Invariant Anomaly Comparison 108
10.7 Conclusion and Future work 109

In this chapter, we identify three main classes of common application invariants. We associate each class with useful programming patterns that help application developers to implement the invariant on a replicated database. Finally, we review some well-known consistency protocols, and compare them in terms of guarantees they provide for the same three classes of invariants.

10.1 Classes of Invariant

An application invariant specifies some safety properties that all states must satisfy. An invariant restricts the possible values observed by users, and the possible updates that a replica may perform.

The invariant is guaranteed in any sequential execution if and only if each individual operation has sufficient precondition. For instance, in a bank application with non-negative balance invariant, a withdraw form an account is legal only if the account has sufficient balance.

Concurrent execution of operations may violate it even if a sequential execution satisfies the invariant. For instance, if balance of an account be initially €2. Concurrent execution of two withdraw(2) operations would make the balance negative (€-2).

Programming an application is much harder if replicated databases do not support invariants, as application developers need to deal with invariant violations. The database may implement some consistency guarantees, i.e., without application-level intervention. Different consistency models provide different guarantees. For instance, the strongest consistency model, called *Strict Serialisability* [80] (SSER), preserves sequential invariants, but it requires to synchronise the critical path of every operation's execution across the whole database. More relaxed consistency models have potential for higher performance, but they make weaker guarantees and might violate some kinds of invariants.

One promising approach to understanding which invariants are enforced by the consistency model, is to leverage our static CISE analysis.

However, our experience shows that many applications share invariants of a similar flavour. For instance, the foreign key invariant is a common kind of invariant, required by many applications. This invariant constrains an object to a subset of values that match values of a different object. Understanding these common invariants could minimise the cost of developing and verifying a large variety of programs.

In this chapter, we present the following three classes of invariants: *Generic1(Gen1)*, *Equivalence(EQ)*, and *Partial-Order(PO)* invariants. We argue that many application invariants are combination of these classes. A Gen1 invariant specifies a constraint over the value of a single data item, ranging from simple properties (for instance, that a value is non-negative) to complex properties of whole data structures (for instance, acyclic graph). PO and EQ invariants relate the state of different data items. For instance, $x_i = x_j$, and $x_i \geq x_j$ represent invariants of EQ type, and of PO type, respectively.

We propose to associate some programming patterns to these classes of invariants where synchronisation is not necessary. For instance, in a bank application under a non-negative balance constraint, deposit operations are always safe.

We also identify patterns where synchronisation is necessary, but can be relaxed in a disciplined manner. If the application follows the pattern, this extends the database's guarantees to the associated class of invariants, often without synchronisation. Each of these invariant patterns matches to a certain consistency property: multi-operation transaction, i.e, atomicity, causal ordering, and total ordering, respectively. Atomicity is the "all-or-nothing" concept in ACID transactions, meaning that all replicas either see the effect of all the operations in the transaction as a unit, or none of them. Causal ordering ensures that operation executions respect Lamport's happened-before relation [64]. Total ordering serialises all operations. Atomicity, causal ordering and total ordering are orthogonal properties. Only total ordering requires full synchronisation (i.e., consensus). Given an invariant class, we identify its consistency requirement, and propose programming patterns, which maintains the invariant.

10.2 Generic Invariants on a Single Item (Gen1)

A Generic single-item invariant (Gen1) places constraints over the state of a single data item, i.e., it prevents some particular set of value from appearing in a database. For instance, the non-negative constraint for balance in the bank application, the capacity limit for course objects in the courseware application, and the stock limit for products in the auction application (Chapter 6) are examples of Gen1 invariants. A Gen1 invariant may be more complex, for instance, the SHA-2 hash of an object is equal to some constants, or that the invariant that some graph structure may not form a cycle.

10.2.1 Protocols and Mechanisms for Gen1-Invariants

In a sequential environment, the protocol to ensure Gen1 invariants is straightforward. We only need to ensure that the effect of each operation in isolation preserves the invariant. Thus, the verification of sequential execution reduces to verifying every single operation. An operation maintains the invariant if it has sufficient precondition to ensure that. If the initial state, before the operation, satisfies its precondition, then the final state, after applying the operation, must satisfy the invariant. For instance in the bank application with non-negative balance, a sufficient precondition for the withdraw operation to maintain this invariant is that the balance is greater than the amount debited,

$$\{\text{balance} \geq 0 \wedge \text{balance} \geq \text{amount} \geq 0\} \quad \text{withdraw(amount)} \quad \{\text{balance} \geq 0\}$$

To implement the Gen1 invariant in concurrent executions, we consider two cases: bounded and unbounded concurrency. In the bounded concurrency model, both the number of concurrent processes, and the number of updates that each process independently performs, are limited. To verify the Gen1 invariant in such environments, we define the precondition of each individual operation to take into account all possible concurrent updates. For instance, consider there are only two bank branches, and each one may withdraw up to €100 from the same account. The precondition that verifies that the balance in each branch is greater than €200 before performing a withdrawal, is enough. Generalising to a bank application with up to k concurrent withdraw operations, each withdrawing some bounded amount $amount_i$ from the same account, precondition of a withdraw operation must include the effect of all possible concurrent k withdraw operations,

$$balance \geq \sum_{i=1}^{k} amount_i$$

where k is bounded.

However, under unbounded number of concurrent withdraw operations, there is no sufficient precondition that can ensure non-negative balance.

Thus, Gen1 invariant requires to limit the concurrency. The general pattern for concurrency control is to totally order operations. For instance, we need to totally order withdraw operations in order to stop a withdrawal that would make the balance negative.

10.2.2 Total-Order

Two operations u and v are *totally-ordered* if and only if their effectors execute in the same order at all replicas. Total order of effectors ensures that all replicas that have observed the same set of (deterministic) operations have the same state.

There are different ways to order updates, ensuring that all replicas observe updates in the same order. One ways is for each replica to independently assign unique timestamps to its updates. The total order is given by the timestamps. This approach is known as *last-writer-wins (LWW)* (also called Thomas's write rule [54]). LWW guarantees Gen1 invariant, but updates may be arbitrarily lost. It may drop an update u when there is an update v concurrently executing with u, and with a larger timestamp. For example, consider a LWW register with a write(a) operation that sets value of the register to a. Two replicas r_1 and r_2 concurrently perform operations write(1), and write(2), where write(1) has timestamp t_1 and write(2) has timestamp t_2. If t_2 is higher than t_1, whatever the order of execution, only write(1) takes effect, i.e., the value of the register will be 1.

To avoid such anomalous behaviour, an agreement protocol, such as Paxos [47] or atomic broadcast [30] is needed to deliver and execute all updates in the same order, to all replicas.

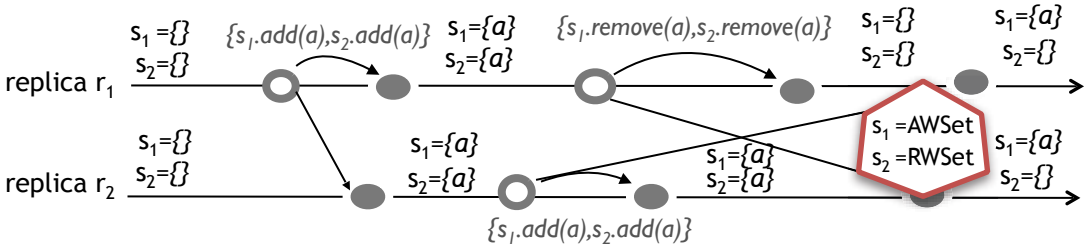


Figure 10.1: Atomicity and different concurrent set semantics

10.3 EQ Invariants

EQ invariants capture some equivalence relation between multiple data items. A simple example is a bi-directional relationship constraint, such that if Bob is a friend of Alice, then Alice must be a friend of Bob, i.e., either both relationships should be established, or neither should be,

$$\forall x, y, \quad x.friend(y) \Longleftrightarrow y.friend(x)$$

The friendship relation is an EQ constraint, which requires to update Alice's friends and Bob's friends together.

10.3.1 Protocols and Mechanisms for EQ-Invariants

An EQ invariant indicates that state of one object must be equivalent to that of some other object. In other words, every change to the former requires a change to the latter. Consider a scenario where Bob and Alice befriend each other. To maintain the EQ invariant, the operation that adds Bob to the set of Alice's friends must simultaneously add Alice to the set of Bob's friends. Then, the database must guarantee that every replica sees these two updates together. Thus, every user either observes both $Bob \in friends(Alice)$ and $Alice \in friends(Bob)$ or, both $Bob \notin friends(Alice)$ and $Alice \notin friends(Bob)$. The *atomicity* property ensures either effect of all updates inside a unit of execution ("transaction") are observed, or none.

The simple strategy to provide atomicity in a replicated database is to implement a mutual exclusion concurrency control policy. For example, if a replica wants to atomically update two sets s_i and s_j with EQ invariant $s_i = s_j$, it can acquire exclusive locks for each of sets s_i and s_j, update both sets, then release the lock. No other replicas will observe partial updates to sets s_i and s_j. However, the locking strategy incurs a significant synchronisation cost and limits concurrency.

The alternative and asynchronous solution to ensure atomicity is using highly-available atomic transactions [13]. Using transactions, we are able to group all updates affecting the EQ-dependent data items, and treat them as a single update. Atomicity guarantees that every replica either applies all effects of a transaction together or none.

10.3.2 EQ Invariants and Convergence Resolutions

However, in the absence of the total order, atomicity is not sufficient to preserve an EQ invariant in a replicated database, which leverages some convergence conflict resolution policies for handling concurrent updates on data items. The combination of different convergence heuristics used for EQ-dependent data items may violate the EQ invariant between them. For instance, consider the program execution illustrated in Figure 10.1. Initially sets s_1 and s_2 are empty, with an EQ invariant $s_1 = s_2$. Set s_1 follows an add-wins semantics, whereas set s_2 follows a remove-wins semantics in case of concurrent adds and removes on the same elements. Replica r_1 atomically adds element a to sets s_1 and s_2. Replica r_2 observes the operation and applies the same effect over its state. Now the state in both replicas is $a \in s_1$, and $a \in s_2$. Some time later, replica r_1 atomically removes a from these sets, concurrently, replica r_2 atomically adds the same element a to sets s_1 and s_2. After exchanging the replica's updates, a query in both replicas sees the element a in set s_1, but not in set s_2: $a \in s_1$ and $a \notin s_2$, violating the EQ invariant between s_1 and s_2.

The challenge is then to design a replication protocol that will converge to a same value that integrates all updates on a single data item, still maintains EQ invariants among different data items. To locally ensure the EQ invariant, the application must choose to provide a compatible convergence policy for the EQ-dependent data items. Returning to the above example, if the sets s_1 and s_2 both follow the same convergence policy, i.e., either add-wins semantics or remove-wins semantics, the convergent result will be correct with regard to the EQ invariant.

10.4 PO Invariants

Another interesting class of invariants imposes a partial order relation over the state of multiple data items. This type of invariant is called PO Invariant. Unlike the EQ-invariant, for PO-dependent objects, only a subset of the updates affects their dependency relation. For instance, in an auction application, there might be an invariant stating that the number of a product that a seller is auctioning is less than its stock. Adding a product to an auction may violate the invariant, whereas increasing the stock is always safe. Another example is the foreign key integrity rule in database, which constrains an object to a subset of values that match values of a different object. For instance, the foreign key invariant in the courseware application states that an enrolment relation refers to an existing course and a registered student:

$$(c,s) \in \mathsf{Enrollment} \implies c \in \mathsf{Course} \wedge s \in \mathsf{Student}$$

Enrolling students preserves the invariant, but removing students or courses may violate the invariant.

10.4.1 Protocols and Mechanisms for PO-invariants

Common instances of PO invariants include: numeric order ($\leq$), subset ($\subseteq$), and implication ($\Longrightarrow$). We use the general notation $LHS \leq RHS$ to represent a PO invariant, where LHS and RHS are the expressions on the left-hand side and right-hand side of the relation, respectively.

An implication rule ($x \implies y$) logically puts a partial order restriction over database state that relates object x (LHS) partially into object y (RHS), i.e. $x \leq y$. Different application invariants are defined based on the implication rule, such as security rules and foreign keys. For instance, the foreign key invariant in the courseware application is an implication rule, in which $LHS = (c,s) \in \mathsf{Enrollment}(c,s)$, and $RHS = (c \in \mathsf{Course} \wedge s \in \mathsf{Student})$.

One simple pattern to preserve a PO invariant in an asynchronous manner is: if effect of some operation u updates only either the RHS to RHS' where $RHS \leq RHS'$, or the LHS to LHS' where $LHS' \leq LHS$, then executing operation u always preserves the invariant. For example, in the courseware application, adding new courses (increasing RHS), say $c \in \mathsf{Course}$, or disenrolling a student from a course (decreasing LHS), say $(c,s) \notin \mathsf{Enrollment}(c,s)$, are safe with regard into the foreign key invariant.

However, updating the LHS to LHS' where $LHS \leq LHS'$, (respectively, updating the RHS to RHS' where $RHS' \leq RHS$) may violate the PO invariant, i.e., it is unsafe. For instance, enrolling a student s into a course c (increasing LHS), say $(c,s) \in \mathsf{Enrollment}(c,s)$, is unsafe, its effect might violate the PO invariant, e.g., student s enrolled into a non-existent course c.

To ensure the PO invariant in sequential execution, we need a sufficient precondition on every unsafe update. For instance, the precondition to enroll a student into a course is that the course exists.

However, operation's precondition may not be stable under the concurrent executions, resulting into the invariant violation [45]. For instance, a user wants to enroll a registered student s into an existing course c; the precondition is verified. Another user, concurrently, removes the course c. Now the precondition becomes false. If we were to continue the enrollment operation, student s is enrolled into a course that does not exist; violating the PO invariant.

The synchronous approach to guarantee the PO invariant after executing these unsafe operations is to totally order them. For instance, we can make every enrollment and removing operations mutually exclusive.

However, there is also a safe asynchronous approach, called the Demarcation Protocol [20]. The basic idea is to perform a safe operation before an unsafe one, in order to compensate in advance for any possible unsafe side effect. Thus, the execution of every unsafe operation u depends on a set of safe operations, called its *safe-dependent* operations, which must execute before the update, in order to maintain the PO invariant. The execution of the safe updates implies that the precondition of operation u is true, and hence the effect of operation is safe. For instance, the unsafe $\mathsf{enroll}(s,c)$ operation depends on $\mathsf{addCourse}(c)$ and $\mathsf{register}(s)$ operations. Before a replica applies the effector of the operation $\mathsf{enroll}(s,c)$, it must apply the effector of

addCourse(c) and register(s) operations, which guarantee student s is registered and course c exists.

Definition 10.1. Given a partial order $\preceq$, operation u can increase *LHS* (resp. decrease *RHS*), if and only if there is some operation v happened before the operation u that increases *RHS* (resp. decreases *LHS*), such that $\forall \sigma \in \mathsf{State}, P_u(\mathcal{F}_v^{eff}(\sigma))$.

To preserve the PO invariant, the replicated database needs to provide causal delivery ensuring that all safe operations that an unsafe update u depends upon, are delivered before the update u at all replicas. Returning into above example, adding a new course must be delivered before enrolling a student into the course.

10.5 Composite Invariant Patterns

The invariant patterns can be combined using conjunction ($\wedge$) and disjunction ($\vee$).

Implementing disjunction needs at least one of the sub-patterns is satisfied, but no order requirement among these sub-patterns. Conjunction restricts the program execution to preserve all sub-patterns, but no order is required among these sub-patterns. The remainder of this section will discuss the protocol for maintaining different combination of invariants in more detail.

10.5.1 Composing Gen1-Invariants

Let I_1 and I_2 be two invariants of Gen1 invariant class.

Conjunction Conjunction of a set of invariants is satisfiable if there is at least a program execution that preserves all invariants in the set simultaneously. A sequential execution maintains the conjunction invariant if every of its operations has sufficient precondition. For instance, let P_1, and P_2 be preconditions of an operation o required for maintaining invariants I_1, and I_2, respectively. Then the o's precondition to maintain the conjunction invariant is $P = P_1 \wedge P_2$. If the precondition P is true, applying the effect of the operation o is always safe.

However, concurrent executions may violate the sequential invariant. A replicated database needs to totally order an operation execution that violates at least one of the invariants I_1 and I_2, i.e., it is unsafe with regard to invariant I_1 or invariant I_2.

Disjunction To implement the disjunction pattern for Gen1 invariants, a replicated database needs to totally order an operation o only if applying the effect of the operation o may violate both invariants I_1 and I_2. Consider the counter x has a disjunction invariant $I = x \geq 0 \vee x \leq 3$. Although applying the effect of `inc()` is unsafe with regard to the sub-invariant $x \leq 3$, but it still maintains invariant I because its effect ensures the sub-invariant $x \geq 0$.

10.5.2 Composing Gen1-Invariant with EQ-Invariant

Let invariant I_1 be a Gen1 invariant on a data item x, and invariant I_2 relates the state of data item z into another data item y.

Conjunction The conjunction protocol implies that every operation updates atomically z and y, and if applying its effect is unsafe with regard to invariant I_1, then the operation must be also serialised.

Disjunction The asynchronous pattern to execute an operation o, while still maintaining the disjunction invariant is: if applying the effect of operation o is always safe regards to invariant I_1, or, if the effect of the operation atomically changes z and y, then operation o satisfies the disjunction invariant.

The synchronous pattern is to serialise the operation o for ensuring invariant I_1.

10.5.3 Composing Gen1-Invariant with PO-Invariant

Let invariant I_1 be a Gen1 invariant, and invariant I_2 be a PO invariant.

Conjunction The asynchronous pattern to maintain the conjunction invariant is: if applying an operation o is safe with regard to both invariant I_1 and I_2, then its effect always satisfies the conjunction, but if it may violate the invariant I_2, then we need to ensure causal delivery for operation o and all its safe-dependent operations.

However, if applying the effect of operation o may violate the invariant I_1, it must be serialised.

Disjunction The asynchronous pattern to maintain the disjunction invariant is: if the operation o is safe with regard to either invariant I_1 or invariant I_2, then its effect satisfies the disjunction. Otherwise, we only need to provide causal delivery, ensuring that operation o satisfies invariant I_2.

The synchronous pattern is to serialise the operation o for ensuring invariant I_1.

10.5.4 Composing EQ-Invariants

Let I_1, and I_2 be two EQ invariants.

Conjunction The pattern to maintain the conjunction between I_1 and I_2 is to ensure that the effect of each operation atomically changes all EQ-dependent data items. For instance, if I_1 is $x = y$, and I_2 is $w = z$, then every operation needs to atomically update x and y together, and atomically update w and z.

Disjunction Returning into the previous example, every operation needs to either atomically update x, and y, or, atomically update w, and z.

10.5.5 Composing EQ-Invariant with PO-Invariant

Let invariant I_1 be a PO invariant, and invariant I_2 is an EQ-Invariant.

Conjunction The pattern to maintain the conjunction invariant is: an operation o maintains the conjunction only if it atomically updates the EQ-dependent operations, and its effect is safe with regard to invariant I_1. However, if applying the effect of operation o may violate the invariant I_1, its execution needs causal delivery between the operation o, and all its safe-dependent operations.

Disjunction An operation o maintains the disjunction if it either atomically updates the EQ-dependent data items or it preserves the partial invariant I_1.

10.5.6 Composing PO-Invariants

Let I_1, and I_2 be two PO invariants.

Conjunction The pattern to preserve the conjunction is: an operation o maintains the conjunction invariant if it is safe with respect to both invariants I_1, and I_2, i.e., applying its effect is safe with regard to both PO invariants. However, if applying the effect of an operation o does not maintains any of them, causal delivery must be applied to the operation o and all its safe-dependent operations.

Disjunction The pattern to preserve the disjunction invariant is: an operation o maintains the disjunction invariant if it is safe with respect to at least one of the invariants I_1, and I_2. Otherwise, causal delivery is required for the operation, so that it can ensure at least one of the partial relations.

Composite patterns are useful from two points of view. First, it helps to express more compound invariants using the three invariant classes. For instance, disjunction pattern allows to support some forms of temporal integrity rules. Suppose in the bank example, the application might want to state, for example, an account x can have a negative balance on the condition that it gets some rights. This places a temporal constraint over the balance, i.e., balance needs to keep positive only when its account has no right. The above rule can be specified using a disjunction as follows: $right = true \vee balance \geq 0$, where $right = true$ represents the EQ invariant that asserts the account has the right to be negative, and the second assertion is a Gen1 invariant.

Moreover, composite patterns allow to decompose an invariant pattern into a number of sub-patterns and solve each one. Protocols to the sub-patterns are then combined to give a

solution to the original invariant. Consider an integrity rule $x + y \leq z + w$, where x, y, z, and w each represents different data items. One solution to implement this invariant is to decompose it into two sub-patterns $x \leq z$ and $y \leq w$ using conjunction rule, and ensuring that operation executions maintain each sub-pattern.

10.6 Consistency Models

So far, we showed that each of our invariant patterns matches to a certain consistency property: multi-operation transaction, i.e, atomicity, causal ordering, and total ordering, respectively. Different consistency models may provide different guarantees or or even no guarantee for the same three properties. In this section, we study some of the main consistency models, along with invariant anomalies that they may expose.

10.6.1 Strict Serialisability (SSER)

Strict serialisability (SSER) [80] is the strongest consistency model: it provides the intuition of a single server that executes all operations, even non-conflicting operations, in real-time order. For instance, Alice transfers €100 into Bob's account, and some time later Bob reads his account. A database providing SSER will guarantee that Bob sees €100 in his account. A program execution under SSER has the following three main features:

- Operations appear to occur in some sequential order.
- The total ordering of operations is consistent with the external order established by real time.
- A read operation sees either the effect of all operations done within a transaction, or none.

Each of these features is sufficient to maintain one of the three invariant classes, and hence, relaxing a feature may result into weakening the corresponding invariant. The first condition totally orders operations, which is required by some operations for ensuring Gen1 invariants. The external ordering implies causal delivery between operations required to maintain the PO invariant, and the third condition is the atomicity required by the EQ invariant. SSER also enforces generic invariants across data items.

10.6.2 Serialisability (SER)

Serialisability (SER) ensures that every execution of operations is equivalent to some serial execution. However, the serial order does not necessarily respect the external or causal ordering. For instance. Alice transfers €100 into Bob's account, and then €200 into Sara's account, some one else may observe Sara's €200, but not Bob's €100.

Consistency Criteria	Consistency Guarantees	Invariant Guarantees
Strict Serializability (SSER)	atomic, total order, external order	PO, EQ, Gen1
Serialisability (SER)	atomic, total order	PO*, EQ, Gen1
Snapshot Isolation (SI)	atomic, causal and total order (writes)	PO*, EQ, Gen1
Causal Consistency (CC)	atomic, causal order	PO, EQ
Eventual Consistency (EC)	-	-

PO indicates that the model maintains some PO invariants*

Table 10.1: Some consistency models and their invariant guarantees.

Violating the causal ordering may cause that operation executions under SER violate some kind of partial oder invariants. Consider an advertisement application with view() and like() operations. The view() operation increases the number of times that an ad is viewed, and the like() operation increases the number of people who liked the ad. The application requires that the number of times an ad that is viewed is less than the number of people who liked the ad. This invariant entails that like() operation must happen before view() operations. However, a serialisable execution may order the view() transaction before like() transaction, meaning that a client sees an ad before she likes the ad; violating the invariant.

10.6.3 Snapshot Isolation (SI)

Snapshot isolation (SI), introduced by Berenson et al. [23], is one of the most popular consistency models provided by commercial database systems, such as Oracle [88] and Microsoft SQL Server [74]. Under snapshot isolation, atomicity is divided into two properties: reads happen logically at the start of the transaction and writes happen logically at the end of the transaction. Read operations return a consistent snapshot of database that reflects the effect of all updates executed before the transaction starts. SI disallows the concurrent writes on the same data items.

SI does not ensure a serialisable behaviour, i.e., it allows concurrent reads on the same data items. It allows *write skew* anomaly. A write skew occurs when two concurrent users read the same data items, but modify disjoint data items. For instance, consider Alice's bank account and Bob's bank account are linked with each other, and the integrity rule over the linked accounts allows that one of the account balances to be negative, as long as that the total balance is never negative; this is a PO invariant. Assume a scenario where Alice and Bob each have €50 in their account. Under SI, if they both want to withdraw €60 concurrently from their account, they first take a consistent snapshot from their accounts. Alice reads her balance and Bobs's balance and concludes that they still have €100 available. In the same way, Bob also concludes that they have €100 available. Alice then withdraws €60 from her account. At the same time, Bob withdraws €60 from his account. This results in €-10 balance for each account, violating the invariant.

10.6.4 Causal Consistency (CC)

The strongest available and convergent consistency model is causal consistency (CC) [70, 72]. Causal consistency [11] implements causal delivery, i.e., guaranteeing that updates are executed in their happened-before order in a distributed system. Under CC, the effect of an operation is visible only after all operations, which happened before the operation are observed. A replica delays applying the effect of an update on a data item until all the update's dependencies are satisfied.

Maintaining causal delivery is important for applications with partial order invariants. A causally-consistent replicated database can implement the PO invariant without any need for synchronisation, ensuring the visibility order required by every unsafe update and all its safe-dependent operations inside a session. For instance, a CC protocol that always orders removing a data item after referring to the data item, ensures the foreign key invariant. Thus, programmers never have to deal with the situation where they can get the reference to a data item that does not exist.

However, causal consistency does not restrict the execution order of concurrent operations, and hence may break some integrity rules. For instance, Gen1 invariants require concurrent clients updating the same data item being ordered.

10.6.5 Eventual Consistency (EC)

Eventual consistency (EC) [100] provides a highly-available (and scalable) data storage based on asynchronous replication model. Under eventual consistency, an operation may execute at some origin replica without synchronising with other replicas. The origin replica propagates the update to other replicas in background. The effect of any update is eventually applied at all replicas, but possibly in different orders. EC does not restrict the order in which updates execute across replicas, even for those are causally related. Hence, data replicas are allowed to diverge.

Eventually-consistent data stores ensure that all replicas eventually converge to the same state. However, applications can be exposed to consistency anomalies that may arise from arbitrary update ordering.

10.6.6 Invariant Anomaly Comparison

Table 10.1 summarises all the consistency models that we reviewed in this section, and compare them in terms of the invariant preservation. While SSER preserves all the three sequential invariants, weaker consistency models than SSE may provide weaker guarantees or even no guarantee for the same three classes.

10.7 Conclusion and Future work

In this chapter, we have proposed three classes of invariants. We illustrated each of the three classes, the associated consistency property, and some programming patterns to implement the invariant. We also studied some cases of composite invariants. Finally, we reviewed several consistency models and consider possible invariants that they might violate.

One future research direction is to prove the correctness of our invariant patterns. As an initial step, we plan to apply the CISE analysis to verify the invariant patterns on a causally-consistent replicated database. One could also consider a database that implements our invariant patterns as a basis for exploring the consistency choice of various operations.

Bibliography

[1] https://github.com/Z3Prover/z3.

[2] Google Drive. `https://www.google.com/drive/`, 2015.

[3] Microsoft OneDrive. `https://onedrive.live.com/`, 2015.

[4] Daniel J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *IEEE Computer*, 45(2):37–42, February 2012.

[5] J.-R. Abrial. A system development process with event-b and the rodin platform. In *Proceedings of the Formal Engineering Methods 9th International Conference on Formal Methods and Software Engineering*, ICFEM'07, pages 1–3, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.d., MIT, Cambridge, MA, USA, March 1999.

[7] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995.

[8] Amazon. Supported operations in DynamoDB. `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/APISummary.html/`, 2015.

[9] Masoud Saeida Ardekani. *Ensuring Consistency in Partially Replicated Data Stores*. Ph.d., UPMC, Paris, France, September 2014.

[10] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems. In *32nd Symposium on Reliable Distributed Systems (SRDS)*, pages 163–172, Braga, Portugal, October 2013. IEEE Comp. Society.

[11] Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of highly-available eventually-consistent data stores. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 385–394, Donostia-San Sebastián, Spain, July 2015. ACM.

[12] B. R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: beyond commutativity. *Trans. on Database Systems*, 17(1):163–199, March 1992.

[13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. VLDB Endow.*, 7(3):181–192, November 2013.

[14] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *PVLDB*, 2015.

[15] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *ACM Queue*, 2014.

[16] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 198–212, New York, NY, USA, 1991. ACM.

[17] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Int. Conf. on Mobile Comp. and Netw. (MobiCom '98)*. ACM/IEEE, October 1998.

[18] Valter Balegas, Nuno Preguiça, Rodrigo Rodrigues, Sérgio Duarte, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Euro. Conf. on Comp. Sys. (EuroSys)*, pages 6:1–6:16, Bordeaux, France, April 2015.

[19] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Marc Shapiro, and Mahsa Najafzadeh. Extending eventually consistent cloud databases for enforcing numeric invariants. *CoRR*, abs/1503.09052, 2015.

[20] Daniel Barbará-Millá and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal, The Int. J. on Very Large Data Bases*, 3(3):325–353, July 1994.

[21] Basho Inc. Using strong consistency in Riak. `https://docs.basho.com/riak/latest/dev/advanced/strong-consistency/`, 2015.

[22] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 1–10, New York, New York, USA, 1995. ACM Press.

[23] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[24] Philip Bernstein, Vassos Radzilacos, and Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[25] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

[26] N. Biri and D. Galmiche. Models and separation logics for resource trees. *Journal of Logic and Computation*, 17(4):687–726, 2007.

[27] Ken Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *Trans. on Computer Systems*, 5(1):47–76, January 1987.

[28] Nikolaj Bjørner. Models and software model checking of a distributed file replication system. In *Formal Methods and Hybrid Real-Time Systems*, pages 1–23, 2007.

[29] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Footprint analysis: A shape analysis that discovers preconditions. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, pages 402–418, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication*, volume 5959 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 2010.

[31] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Symp. on Op. Sys. Principles (SOSP)*, pages 1–17, Farmington, PA, USA, 2013. ACM SIG on Op. Sys. (SIGOPS), Assoc. for Computing Machinery.

[32] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[33] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 251–264, Hollywood, CA, USA, October 2012. Usenix.

[34] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[35] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Symp. on Op. Sys. Principles (SOSP)*, volume 41 of *Operating Systems Review*, pages 205–220, Stevenson, Washington, USA, October 2007. Assoc. for Computing Machinery.

[36] Aboubakr Achraf El Ghazi and Mana Taghdiri. Analyzing alloy constraints using an smt solver: A case study. In *5th International Workshop on Automated Formal Methods (AFM)*, Edinburgh, United Kingdom, 2010.

[37] Gidon Ernst, Gerhard Schellhornand Dominik Haneberg, Jörg Pfähler, and Wolfgang Reif. *Verification of a Virtual Filesystem Switch*, pages 242–261. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[38] Alan Fekete. Allocating isolation levels to transactions. In *PODS*, 2005.

[39] L. Freitas, Zheng Fu, and J. Woocock. Posix file store in z/eves: an experiment in the verified software repository. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 3–14, July 2007.

[40] L. Freitas, Jim Woodcock, and A. Butterfield. Posix and the verification grand challenge: A roadmap. In *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pages 153–162, March 2008.

[41] Hector Garcia-Molina and Gio Wiederhold. Read-only transactions in a distributed database. *Trans. on Database Systems*, 7(2):209–234, June 1982.

[42] Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the posix file system. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 169–188. Springer Berlin Heidelberg, 2014.

[43] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[44] Daniel Gómez Ferro and Maysam Yabandeh. A critique of snapshot isolation. In *Euro. Conf. on Comp. Sys. (EuroSys)*, pages 155–168, Bern, Switzerland, April 2012. Assoc. for Computing Machinery.

[45] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *Symp. on Principles of Prog. Lang. (POPL)*, St. Petersburg, FL, USA, 2016.

[46] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Int. Conf. on the Mgt. of Data (SIGMOD)*, pages 173–182, Montréal, Canada, June 1996. ACM SIGMOD, ACM Press.

[47] Jim Gray and Leslie Lamport. Consensus on transaction commit. *Trans. on Database Systems*, 31(1):133–160, March 2006.

[48] Richard Guy, John S. Heidemann, Wai Mak, Gerald J. Popek, and Dieter Rothmeier. Implementation of the ficus replicated file system. In *In USENIX Conference Proceedings*, pages 63–71, 1990.

[49] Chen Haogang, Ziegler Daniel, Chajed Tej, Chlipala Adam, Kaashoek M. Frans, and Zeldovich Nickolai. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM.

[50] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Symp. on Principles and Practice of Parallel Prog. (PPoPP)*, pages 207–216, New York, NY, USA, 2008. ACM.

[51] Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

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

[53] J. Hughes. Specifying a visual file system in z. In *Formal Methods in HCI: III, IEE Colloquium on*, pages 3/1–3/3, Dec 1989.

[54] Paul R. Johnson and Robert H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, January 1976.

[55] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: A scalable approach to logging. *Proc. VLDB Endow.*, 3(1-2):681–692, September 2010.

[56] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*. North-Holland, 1983.

[57] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. on Comp. Sys. (TOCS)*, 10(5):3–25, February 1992.

[58] Viktor Kuncak Konstantine Arkoudas, Karen Zee and Martin Rinar. Verifying a file system implementation. In *Formal Methods and Software Engineering*, pages 373–390, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[59] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009.

[60] Damchoom Kriangsak, Butler Michael, and Abrial Jean-Raymond. Modelling and proof of a tree-structured file system in event-b and rodin. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM '08, pages 25–44, Berlin, Heidelberg, 2008. Springer-Verlag.

[61] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. Exploiting the commutativity lattice. In *Conf. on Prog. Lang. Design and Implementation*, pages 542–555, San Jose, California, USA, June 2011. Assoc. for Computing Machinery.

[62] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 233–243, New York, NY, USA, 2008. ACM.

[63] Puneet Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Usenix Tech. Conf.*, New Orleans, LA, USA, January 1995.

[64] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[65] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[66] L.B.Hustonand and Peter Honeyman. Disconnected operation for afs. In *In USENIX Conference Proceedings*, page 1?10, 1993.

[67] K. Rustan M. Leino. Automating induction with an smt solver. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 315–331, Berlin, Heidelberg, 2012. Springer-Verlag.

[68] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[69] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 265–278, Hollywood, CA, USA, October 2012.

[70] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Symp. on Op. Sys. Principles (SOSP)*, pages 401–416, Cascais, Portugal, October 2011. Assoc. for Computing Machinery.

[71] David Lomet. Simple, robust and highly concurrent B-Trees with node deletion. In *Proc. 20th Int. Conf. on Data Engineering (ICDE'O4)*, pages 18–28, Boston, MA, USA, April 2004. IEEE Computer Society, IEEE.

[72] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, Dept. of Comp. Sc., The U. of Texas at Austin, Austin, TX, USA, 2011.

[73] Microsoft. Consistency levels in DocumentDB. `https://azure.microsoft.com/en-us/documentation/articles/documentdb-consistency-levels/`, 2015.

[74] Microsoft Corporation. Transact-sql reference. `https://msdn.microsoft.com/en-us/library/ms173763.aspx/`, 2014.

[75] Carroll Morgan and Bernard Sufrin. Specification of the unix filing system. *Software Engineering, IEEE Transactions on*, SE-10(2):128–142, March 1984.

[76] Yannick Moy. *Verification, Model Checking, and Abstract Interpretation: 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008. Proceedings*, chapter Sufficient Preconditions for Modular Assertion Checking, pages 188–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[77] Gian Ntzik and Philippa Gardner. Reasoning about the posix file system: Local update and global pathnames. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 201–220, New York, NY, USA, 2015. ACM.

[78] Patrick E. O'Neil. The escrow transactional method. *Trans. on Database Systems*, 11(4):405–430, December 1986.

[79] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. *SIGOPS Oper. Syst. Rev.*, 19(5):15–24, December 1985.

[80] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[81] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Symp. on Op. Sys. Principles (SOSP)*, pages 288–301, Saint Malo, October 1997. ACM SIGOPS.

[82] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. Locus: A network transparent, high reliability distributed system. In *Symp. on Op. Sys. Principles (SOSP)*, pages 169–177. ACM, 1981.

[83] POSIX.1-2008. The open group base specifications issue 7.

[84] Norman Ramsey and Előd Csirmaz. An algebraic approach to file synchronization. Technical Report TR-05-01, Harvard University Dept. of Computer Science, Cambridge MA, USA, May 2001.

[85] David Ratner, Peter Reiher, and Gerald Popek. Roam: A scalable replication system for mobile computing. In *Int. W. on Database & Expert Systems Apps. (DEXA)*, pages 96–104, Los Alamitos, CA, USA, 1999. IEEE Comp. Society.

[86] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *Usenix Conf.* Usenix, June 1994.

[87] John 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, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[88] Richard Strohm. Oracle Database Concepts, 11g Release 1 (11.1), January 2011.

[89] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1311–1326, New York, NY, USA, 2015. ACM.

[90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. on Computers*, 39(4):447–459, April 1990.

[91] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag.

[92] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *PLDI*, 2015.

[93] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pages 385–400, Cascais, Portugal, October 2011. Assoc. for Computing Machinery.

[94] J. M. Spivey. The z notation: A reference manual. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, 1998. Prentice-Hall.

[95] Vinh Tao, Marc Shapiro, and Vianney Rancurel. Merging semantics for conflict updates in geo-distributed file systems. In *ACM Int. Systems and Storage Conf. (Systor)*, pages 10.1–10.12, Haifa, Israel, May 2015.

[96] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Int. Conf. on Para. and Dist. Info. Sys. (PDIS)*, pages 140–149, Austin, Texas, USA, September 1994.

[97] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[98] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Symp. on Op. Sys. Principles (SOSP)*, pages 172–182, Copper Mountain, CO, USA, December 1995. ACM SIGOPS, ACM Press.

[99] Werner Vogels. File system usage in windows nt 4.0. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 93–109, New York, NY, USA, 1999. ACM.

[100] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008.

[101] A.-I.A. Wang, P. Reiher, R. Bagrodia, and G.H. Kuenning. Understanding the behavior of the conflict-rate metric in optimistic peer replication. In *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*, pages 757–761, Sept 2002.

[102] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. on Computers*, 37(12):1488–1505, December 1988.

[103] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. on Computers*, 37(12):1488–1505, December 1988.

[104] M.I. Lali Wim H. Hesselink. Formalizing a hierarchical file system. *Formal Aspects of Computing*, 24(1):27–44, 2010.

[105] Haifeng Yu and A. Vahdat. Building replicated internet services using tact: a toolkit for tunable availability and consistency tradeoffs. In *Advanced Issues of E-Commerce and Web-Based Information Systems, 2000. WECWIS 2000. Second International Workshop on*, pages 75–84, 2000.