When Sequential Code Meets Replicated Data

Programming Language Support to Simplify the Development of Correct Replicated Data Types

Kevin De Porre

Dissertation submitted in fulfillment of the
requirement for the degree of Doctor of Sciences

December 19, 2022

Promotor:
Prof. Dr. Elisa González Boix, Vrije Universiteit Brussel

Jury:
Prof. Dr. Bart de Boer, Vrije Universiteit Brussel, Belgium (chair)
Prof. Dr. Bas Ketsman, Vrije Universiteit Brussel, Belgium (secretary)
Prof. Dr. Nikolaos Deligiannis, Vrije Universiteit Brussel, Belgium
Prof. Dr. Annette Bieniusa, Technische Universität Kaiserslautern, Germany
Dr. Martin Kleppmann, Technische Universität München, Germany

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel : +32 2 629 33 44
crazycopy@vub.be
www.crazycopy.be

ISBN 9789464443431
NUR 989
THEME UMA

Abstract

Many applications today run atop a geo-distributed system, that is, a system that replicates data over several machines (called replicas) located at strategic positions around the globe. Such systems typically ensure good performance and high availability by allowing replicas to be updated independently. However, replicas may execute concurrent updates in different orders, which can lead to conflicts. Programmers must foresee and handle these conflicts, which is extremely difficult, even for experts.

To avoid manual conflict resolution, researchers proposed replicated variants for sequential data types, called Replicated Data Types (RDTs). RDTs resemble common data structures but internally implement mechanisms to detect and solve conflicts. A data type may have several RDTs exhibiting different semantics depending on how conflicts are solved.

In this dissertation, we argue that real-world applications require custom RDTs that are tailored to the needs of the application. However, existing RDTs cannot be customized as they exhibit hardcoded conflict resolution semantics. This leaves programmers no choice but to build their own RDTs or modify existing RDTs using ad-hoc conflict detection and resolution mechanisms. This is known to be error-prone and results in brittle systems. Moreover, these new or modified RDTs are seldom verified due to the complexity of software verification, and thus bugs are likely to go unnoticed.

To address the aforementioned issues, this dissertation explores programming language abstractions and techniques to support the design, implementation, and verification of geo-distributed systems using RDTs. This led to a solution that is twofold. First, we propose a general approach for replicating existing sequential data types instead of building dedicated RDTs for every use case. Our approach lets programmers augment sequential data types with a *declarative specification* of the desired

conflict resolution semantics expressed through application invariants. We statically analyze the data type to detect all conflicts and find solutions that adhere to the desired semantics. At runtime, our novel replication protocol efficiently serializes operations such that replicas converge to the same state and maintain application-specific invariants with minimal coordination. To validate our approach, we successfully built an extensive portfolio of RDTs and several real-world applications that exhibit performance similar to state-of-the-art approaches.

Second, we devise VeriFx, a novel high-level programming language to implement and verify ad-hoc RDTs. Programmers implement custom RDTs in VeriFx and *automatically* get a proof of correctness or a counterexample if the implementation is wrong. Verified RDTs can be transpiled to mainstream languages in order to deploy them in an existing system. To demonstrate its effectiveness, we used VeriFx to implement and verify a portfolio comprising more than 40 RDTs, distilled from the literature on Conflict-free Replicated Data Types (CRDTs) and Operational Transformation (OT) and from industrial databases.

Our results show two important insights. First, it is possible to build efficient RDTs with application-specific concurrency semantics, without having to manually handle conflicts. Second, the implementation and verification of RDTs can be unified in a high-level language such that software engineers without deep knowledge of verification can nevertheless implement RDTs and verify them automatically.

Samenvatting

Hedendaagse applicaties draaien veelal bovenop een geo-gedistribueerd systeem, zijnde een systeem dat gegevens repliceert over meerdere machines (replica's genaamd) die zich op strategische posities over de hele wereld bevinden. Dergelijke systemen zorgen doorgaans voor goede prestaties en hoge beschikbaarheid doordat replica's onafhankelijk kunnen worden aangepast. Replica's kunnen echter gelijktijdige updates in verschillende volgordes uitvoeren, wat tot conflicten kan leiden. Programmeurs moeten deze conflicten voorzien en oplossen, hetgeen zeer ingewikkeld is.

Om te vermijden dat programmeurs conflicten handmatig moeten oplossen, stelden onderzoekers gerepliceerde varianten voor sequentiële datatypes voor, genaamd Replicated Data Types (RDT's). RDT's lijken op traditionele datastructuren maar implementeren intern mechanismen om conflicten op te sporen en op te lossen. RDT's kunnen verschillende semantiek hebben afhankelijk hoe conflicten worden opgelost.

In dit proefschrift argumenteren we dat geavanceerde toepassingen op maat gemaakte RDT's vereisen die zijn afgestemd op de behoeften van de toepassing. Bestaande RDT's kunnen echter niet worden aangepast omdat conflictoplossing hardgecodeerd is. Dit dwingt programmeurs om hun eigen RDT's te bouwen of bestaande RDT's aan te passen met behulp van ad-hoc mechanismen voor conflictdetectie en -oplossing. Het is welbekend dat ad-hoc mechanismen foutgevoelig zijn en resulteren in fragiele systemen. Bovendien worden deze nieuwe of gewijzigde RDT's zelden geverifieerd vanwege de complexiteit van software verificatie, waardoor bugs waarschijnlijk onopgemerkt blijven.

Om de aangehaalde problemen aan te pakken, onderzoeken we in deze dissertatie programmeertaalabstracties en technieken voor de ontwikkeling van correcte RDT's. Dit leidde tot een tweeledige oplossing. Ten eerste

stellen we een algemene aanpak voor om bestaande sequentiële datatypes te repliceren zodat men geen gespecialiseerde RDT's dienen te bouwen voor elke use case. Met onze aanpak kunnen programmeurs sequentiële datatypes uitbreiden met een *declaratieve specificatie* van de gewenste semantiek voor conflictoplossing, uitgedrukt door applicatie specifieke invarianten. We analyseren deze datatypes statisch om alle conflicten op te sporen en oplossingen te vinden die aansluiten bij de gewenste semantiek. Tijdens de programma uitvoering serialiseert ons nieuwe replicatieprotocol bewerkingen op een efficiënte manier zodat replica's naar dezelfde staat convergeren en toepassing specifieke invarianten behouden met minimale coördinatie. Om onze aanpak te valideren, hebben we een uitgebreid portfolio aan RDT's en verschillende applicaties gebouwd die gelijkaardige prestaties als state-of-the-art oplossingen vertonen.

Ten tweede stellen we VeriFx voor, een high-level programmeertaal om ad-hoc RDT's te implementeren en te verifiëren. Programmeurs implementeren nieuwe RDT's in VeriFx en krijgen *automatisch* een bewijs dat hun implementatie correct is of een tegenvoorbeeld indien dit niet het geval is. Geverifieerde RDT's kunnen worden vertaald naar mainstream talen om ze in een bestaand systeem te integreren. Om de effectiviteit van onze aanpak te demonstreren hebben we VeriFx gebruikt om een portfolio van meer dan 40 RDT's te implementeren en te verifiëren. Dit portfolio bevat welgekende RDT's uit de literatuur over Conflict-free Replicated Data Types (CRDT's) en Operational Transformation (OT) alsook uit industriële databanken.

Onze resultaten leiden tot twee belangrijke inzichten. Ten eerste is het mogelijk om efficiënte RDT's te bouwen met toepassing specifieke semantiek, zonder handmatig conflicten op te moeten lossen. Ten tweede kan de implementatie en verificatie van RDT's worden verenigd in een high-level taal zodat software-ingenieurs zonder diepgaande kennis van verificatie toch RDT's kunnen implementeren en automatisch verifiëren.

Acknowledgements

Four years ago, I started this adventure, armed with an idea. Little did I know about the rollercoaster I had just stepped in, which would bring me to so many places to meet amazing people and occasionally would bring me down to get back up shortly after. Luckily, the rollercoaster arrived safely and well at its destination, resulting in this dissertation. Therefore, I want to thank the jury members, Bart de Boer, Bas Ketsman, Nikos Deligiannis, Annette Bieniusa, and Martin Kleppmann, for taking the time to read this dissertation and for their valuable feedback.

Let me now take the time to thank my colleagues, friends, and family who supported me over the years and gave me the strength to continue even in difficult times. I would not be writing this today without the unconditional support of my academic mommy, Elisa Gonzalez Boix. I am very grateful for the freedom she gave me to explore my own path and steer me at the right moments in time.

During my first conference in Dresden in 2019, I met Carla Ferreira, another person that would become special to me as we would go on to collaborate during the rest of my PhD. I now had two academic mothers (an official one and an unofficial one) that perfectly managed co-parenting.

I would also like to thank the fellow DISCOers and ex-DISCOers (Scull, Carmen, Isaac, Matteo, Jim, Carlos, Aäron, Dina) and the fellow SOFTies for the fruitful presentations, discussions, and Friday drinks! Talking about the latter, I should not forget to acknowledge Jonas De Bleser for those memorable Fridays.

There is also a group of nerds called the "zware nerds" (Tony, Djorre, Wito Mojito, and Mette1337) who deserve a special thank you. We met during our studies at the VUB and became close friends.

Finally, I want to thank my mother and father for giving me all these opportunities and always encouraging me. I also want to thank my little

brother, Mateo, which was always there to remind me to stop working and play with him instead ;p Bedankt broertje, voor alle avonden die wij samen al spelend doorbrengen! A special thanks also goes to my Opa, Mami, Papy, Meme, and Tonton for always believing in me. And of course, I kept the most important person as last. Ellen, my girlfriend, my everything, who has given me infinite support and love this entire time. A new chapter is unfolding for us, and I can't wait to discover it!

Contents

Acronyms

ADT Algebraic Data Type.

AST Abstract Syntax Tree.

CAL Combinatory Array Logic.

CmRDT Commutative Replicated Data Type.

CRDT Conflict-free Replicated Data Type.

CvRDT Convergent Replicated Data Type.

DAG Directed Acyclic Graph.

DC Data Center.

DSL Domain-Specific Language.

ECRO Explicitly Consistent Replicated Object.

FOL First-Order Logic.

IPA Invariant-Preserving Applications.

IT Inclusive Transformation.

IVL Intermediate Verification Language.

LAN Local Area Networks.

LoC Lines of Code.

LUB Least Upper Bound.

MRDT Mergeable Replicated Data Type.

OAC Observable Atomic Consistency.

OOP Object-Oriented Programming.

OT Operational Transformation.

PoR Partial Order-Restrictions.

RDT Replicated Data Type.

SEC Strong Eventual Consistency.

SECRO Strong Eventually Consistent Replicated Object.

SMT Satisfiability Modulo Theories.

UI User Interface.

VC Verification Condition.

VM Virtual Machine.

List of Figures

List of Tables

Listings

List of Algorithms

Chapter 1

Introduction

In the early days of computers, programmers exclusively wrote sequential programs that ran on a single machine. Such programs consist of code representing instructions that are executed sequentially one after the other. With the invention of Local Area Networks (LAN), machines could be interconnected within some physical area (e.g. a building) and so the first distributed systems saw the light of day.

A distributed system essentially is a collection of independent machines that work together to accomplish a common goal but appears as a single coherent system to its users [TV07]. For example, email - one of the most successful distributed systems - consists of millions of mail servers and clients that collaborate to exchange electronic messages over the internet.

Over the past decades, we witnessed an unprecedented evolution in hardware technologies giving rise to countless distributed systems. Such systems no longer constitute a niche but are truly mainstream. The applications that run atop them are diverse and span numerous application domains. Examples include applications like Uber and Lyft, which coordinate vehicles, bicycles, and steps to provide a form of transportation-as-a-service, or contact tracing applications like Coronalert that warn users about high-risk contacts based on the location and time of the users.

These hardware advances, especially improved network connectivity, together with the ever-growing urge to be connected, modified the users' expectations which now want applications to work anywhere at any time. However, software technology - i.e. the programming languages and tools that are used to build these applications - has not witnessed a similar evo-

lution. As a result, many distributed applications are written in languages that were designed for sequential programming (e.g. C, Python, etc.) and feature programming paradigms that are not adapted to distributed programming[1].

The lack of evolution in software technology is problematic because distributed programming is intrinsically different from programming local applications that run on a single machine. The main difference is the fact that the independent entities - commonly referred to as nodes - that make up the distributed system can only interact by exchanging messages over a network. As explained by Waldo et al. [Wal+97] this introduces numerous problems that must be addressed by the programmer. For instance, communication between nodes is slow and often unreliable (i.e. messages may arrive out-of-order or even be dropped). In addition, individual nodes or communication links may fail but these *partial failures* should not take down the entire system. Furthermore, nodes may interact concurrently with shared resources or data but care must be taken to ensure data consistency and integrity. This is especially difficult in this context because distributed systems lack a global clock and the clocks of individual machines are unreliable as they are subject to clock drift and skew [TV07]. Hence, the order of events (especially causal relations between them) cannot be determined from their physical time.

In this thesis, we study programming languages, models, and tools for the development and verification of distributed applications. In particular, we focus on the challenges of data replication and consistency.

1.1 Geo-Replicated Systems

Distributed systems commonly replicate (i.e. copy) data to several nodes. Each node is said to hold a "replica". A replica is an abstract notion of a copy. In practice, a replica may be a server in a client-server architecture, a node of a peer-to-peer system, a client of a blockchain, etc.

Replication is crucial to ensure high availability, good scalability, and fault tolerance. It improves fault tolerance because clients can access data from several replicas when a node or communication link fails. Replication also improves scalability as the system can load balance its workload over

[1] An exception is Erlang which was designed for fault-tolerant distributed programming and has been used to build major distributed applications like WhatsApp.

the available replicas. Moreover, replication reduces user-observed latencies by placing copies geographically closer to the clients, a technique that is known as geo-replication. Sometimes, replicas are even placed on the clients to ensure offline availability.

Although replication addresses common issues of distributed systems, it increases the system's overall complexity and raises potential consistency problems. Users are under the illusion of a single coherent system and thus expect to read the latest information but this is not always the case. How often did one modify their profile picture on Facebook only to find out it still uses the old picture for some time? The reason behind this inconsistency is that the update modifies a single replica. Depending on which replica users access they may read outdated information (e.g. their old profile picture). However, the update will eventually be propagated to all replicas such that they become consistent again.

Keeping replicas consistent is a difficult task because distributed systems provide very few guarantees; individual nodes may fail, communication is unreliable, there is no global clock to order events, etc. The main difficulty consists in maintaining the system's consistency guarantees while tolerating network partitions. For example, how should the system handle updates if some replicas are not reachable? The limits of consistency and availability under network partitions are formulated by the CAP theorem [Bre00].

1.1.1 The CAP Theorem

We first explain the different properties (C, A, and P) of this theorem. Assume a model in which distributed components read and write to conceptually shared memory. A distributed system is strongly consistent (C) if every read observes the value of the latest write. It is highly available (A) if components can always execute read and write operations and get a meaningful response. This definition of availability does not allow operations to timeout. Finally, the system is partition tolerant (P) if it is resilient to network partitions; i.e. the system maintains its consistency and availability guarantees under network partitions.

The CAP theorem [Bre00; GL02] states that replicated data in a distributed system cannot be strongly consistent (C), highly available (A), and partition tolerant (P). Instead, the system can achieve only two of these three properties (i.e. AC, AP, or CP). For example, social media

applications like Facebook allow users to post status updates and comment on pictures even when they are offline. Such applications favor availability over consistency and thus are AP. On the other hand, banking systems may keep accounts strongly consistent to avoid overdrafts and thus favor consistency over availability (CP).

Although, in theory, distributed systems can be available and consistent (AC), network partitions are bound to occur in practice. For example, mobile applications like Uber and Waze frequently face disconnections when users are driving through tunnels or poorly connected areas. Such network partitions are inevitable since users are on the move and have only intermittent connectivity. Therefore, every distributed system must be partition tolerant. This essentially leaves programmers with a trade-off between availability and consistency when network partitions occur [Bre12]. This decision is made for every piece of shared data in a distributed system, i.e. some data may be shared in a consistent manner (CP) while another data item is shared in an available manner (AP).

It is important to note that highly available systems tolerate *temporary* inconsistencies but do not have to completely give up on consistency. For example, network partitions in a blockchain may cause accidental forks but eventually, everyone agrees on the longest fork.

1.1.2 The Quest for High Availability and Low Latency

For decades CP was the preferred choice for distributed systems as strong consistency was ought to be a requirement for any application. However, strong consistency has a profound impact on the system's performance and availability because replicas need to agree on all updates.

With the rise in popularity of the Internet and increased access to it, distributed systems started experiencing higher workloads, making CP very costly and resulting in poor availability. Nowadays, it is common for distributed systems to face millions of requests coming from users all over the globe connected through a variety of devices (laptops, smartphones, tablets, etc.). For such systems, high availability, low latency, and good scalability are often more important than strong consistency.

For example, a former software engineer at Amazon explained that every 100ms increase in page load time reduces Amazon's sales by 1% [Lin] which today would account for a $4.8 billion drop in annual revenues [mac]!

For the reasons mentioned above, modern distributed systems tend to favor availability (AP) over consistency (CP). To this end, they relax the consistency guarantees which improves availability and reduces latencies but may lead to temporary inconsistencies between the replicas which appear as anomalies to the users. For example, a highly available web shop may face a situation where two or more users concurrently purchase the last piece of a certain item. After detecting this problem the system will ship the item to one of the users and inform the others that something went wrong and that the item is no longer in stock. To apologize for the inconvenience the shop may offer a discount code to the user. In contrast to what early business models believed, we notice that, over the years, users got used to these kinds of anomalies.

The key to building large-scale distributed systems consists of striking the right balance between availability and consistency for the application at hand. In essence, the system should maximize availability and only choose for strong consistency if the cost to compensate for anomalies exceeds the advantages of high availability [BG13]. For example, web shops like Amazon benefit from a highly available shopping cart because it generates additional sales (thanks to improved user experience) which largely exceeds the cost of compensation (e.g. occasional discount codes). However, the payment system may need to be strongly consistent to ensure that users have enough money to pay for their cart, that discount codes are redeemed only once, etc.

Since the introduction of the CAP theorem in 2000, the landscape of distributed systems significantly changed, and as argued by Kleppmann [Kle15], we should no longer classify distributed systems as fully AP or CP. Nowadays, modern distributed systems often provide mixed consistency guarantees and do not adhere to the C, A, and P properties as defined in the CAP theorem.

1.1.3 Problem Statement

We explained that distributed systems strive for maximal availability which can lead to anomalies (often called conflicts). Unfortunately, conflicts considerably increase the system's complexity as programmers must implement mechanisms to detect and solve them. For decades, programmers have implemented ad-hoc conflict resolution strategies but these are error-prone and result in brittle systems [ASB15; KB17; Sha+11b].

Around 2010, researchers started developing Replicated Data Types (RDTs) to free programmers from manual conflict resolution. RDTs expose an interface akin to a sequential data type but internally embed conflict resolution mechanisms to ensure that all replicas eventually converge to equivalent states (a property known as state convergence). Over the years, researchers devised RDTs for common data structures such as counters, sets, maps, graphs, etc.

Today, researchers are still actively developing new RDTs and improving existing designs. The workflow for the development of RDTs currently consists of three phases: *design*, *verification*, and *implementation*. In the first phase, experts design the RDT such that all conflicts are detected and solved. In the second phase, experts formally verify the design to ensure that it upholds the required consistency guarantees (e.g. state convergence). In the third and final phase, application developers pick up these RDT designs and implement them in their system.

Lately, RDTs are gaining a lot of traction as they are being integrated in industrial databases (Riak, Redis, etc.) and commercial products. However, some problems remain that limit their wider adoption for the development of geo-distributed systems. We identify three main problems:

Non-customizable semantics. The literature provides RDTs with hardcoded concurrency semantics for common data structures, but real-world applications must tailor the semantics to the application's needs. For example, concurrent bookings in a flight reservation system may lead to two passengers reserving the same seat. The system may solve this conflict by assigning the seat to the customer with the most expensive ticket or by favoring customers that are part of their loyalty program, or any other sensible conflict resolution policy. Currently, this requires building a new RDT *from scratch* for this specific use case and thus exposes programmers to conflicts.

Limited support for application invariants. Most RDTs focus on state convergence but do not support application-specific invariants. The inability to maintain invariants comes from the fact that the RDTs have hardcoded conflict resolution strategies and maintaining application-specific invariants requires rethinking those strategies completely.

Some recent approaches augment RDTs with invariants that are described in a separate specification [Li+12; Bal+15; LPR18; Bal+18; SKJ15; Got+16; Kak+18]. The specification is analyzed to detect invariant-breaking operations and the system selectively strengthens the consistency requirements of those operations in order not to break invariants. However, these approaches are conservative and may impose too much coordination, affecting the system's performance and availability. Moreover, programmers have to write separate specifications, often in logic, which is error-prone and hampers software evolution because the specifications must evolve along with the implementation.

Complexity of verification. We identify major threats to the correctness of RDTs in each phase (design, verification, and implementation) of the development. During the design phase, experts may miss subtle corner cases. This has happened even to the most experienced RDT designers [Kle22]. Hopefully, such flaws are caught during the verification phase. However, in the verification phase, RDTs are mostly verified using paper proofs which are subject to reasoning flaws. Recently, researchers started mechanically verifying RDT designs using interactive theorem provers [Gom+17; ZBP14] but this requires additional expertise in formal methods and verification and is extremely time-consuming [LM10]. Finally, programmers are likely to make mistakes during the implementation phase because they do not fully grasp the subtleties underlying RDT designs. Since implementations are rarely verified, any bug in the implementation is likely to go unnoticed. Moreover, real-world applications require custom RDTs, but programmers currently do not have the tools and techniques to design, implement, and verify their own RDTs.

A recent technical report [Kle22] elaborates on the difficulty of designing correct RDT algorithms and the inability of experienced software engineers to find subtle bugs in these algorithms. This confirms our personal experience, therefore, we argue that the current programming techniques and verification tools for the development of RDTs are insufficient.

1.2 Research Vision

To improve the programmability and verifiability of RDTs better programming support and tooling are needed. We believe that every approach must adhere to three fundamental principles:

Don't Design for Replication, Replicate your Design. Designing dedicated RDTs for each data structure and every possible concurrency semantics does not scale. Our vision is that developers should be able to replicate existing data types and declaratively define the concurrency semantics depending on the application's needs.

Correct Replicated Data Types Out-of-the-Box. RDTs should be correct out-of-the-box. Currently, RDT designers are responsible for ensuring correctness, but this requires reasoning about all possible conflicts. Moreover, programmers often adapt existing RDT designs to fit their applications but these changes may render the resulting implementation incorrect. While humans may miss subtle corner cases, machines are better at verifying all cases. We envision specialized verification tools that enable programmers to *automatically* verify high-level RDT implementations.

Programming Language Support. Solutions that aid the development and verification of RDTs must be integrated into a suitable programming abstraction that adheres to the software development principles of code reuse, modularity, etc. For example, programmers may implement common RDT logic in an abstract class and concretize it with application-specific invariants. We envision approaches for the development of RDTs to be integrated into high-level programming languages such that programmers can design, implement, and verify RDTs *within the same* language.

Besides adhering to the above principles, approaches for the development of RDTs must be fast and scale to the needs of modern distributed systems. These performance considerations form an important aspect that drives the design of programming languages and abstractions for RDTs.

1.3 Approach and Contributions

This dissertation starts from the observation that RDTs are difficult to design and implement correctly. We explore novel approaches to design, implement, and verify RDTs. *We foresee a tension between efficiency and simplicity*; many RDTs have specialized implementations that are very efficient but hard to understand, whereas generalized approaches are simpler but cannot achieve the performance of specialized implementations. Thus, approaches for implementing and verifying RDTs must strike a balance between simplicity and efficiency. Our goal is to devise principled solutions that are simple enough to be used by regular software engineers and performant enough for large-scale geo-distributed applications.

To achieve our goal, we explore a novel approach that turns sequential data types into correct RDTs. Our approach combines user-defined specifications with a novel replication algorithm that governs the execution of operations on weakly consistent replicated state to guarantee convergence and maintain application invariants. Moreover, we develop a novel programming language that automatically verifies the correctness properties of RDT implementations, thereby simplifying verification.

Our journey toward a simple and efficient approach to implement and verify RDTs resulted in three concrete contributions which we present in this dissertation:

The ECRO family of RDTs. We devise ECROs, a new family of RDTs that are derived by extending sequential data types with a distributed specification describing the desired concurrency semantics and the application's invariants. Specifications are statically analyzed to derive information about conflicts. This information is then used at runtime by our novel replication protocol to serialize operations in a way that guarantees state convergence and maintains application invariants with minimal coordination between replicas.

Synthesizing RDT specifications. We propose EFx, a programming language with a novel contract system for RDTs. Contracts extend operations with preconditions and invariants. The language automatically synthesizes distributed specifications from the data type's implementation and its contracts. EFx then combines the synthesized specifications with the ECRO approach to derive application-

specific RDTs. Thus, EFx is a high-level language for developing application-specific RDTs using the ECRO approach.

Automated verification of RDTs. We propose VeriFx, a programming language for the implementation and automated verification of RDTs. Programmers implement RDTs in VeriFx which then automatically verifies the necessary correctness properties. VeriFx features built-in correctness properties for well-known RDT families (e.g. CRDTs [Sha+11b]) but programmers can also define custom correctness properties using VeriFx's novel proof construct. For each proof, VeriFx derives the necessary proof obligations, which are encoded into first-order logic and discharged automatically by leveraging SMT solving. If a property does not hold, VeriFx returns a high-level counterexample.

1.4 Supporting Publications

The ideas we realized throughout this dissertation led to several publications. We summarize the main publications supporting our contributions:

- **Putting Order in Strong Eventual Consistency** [De +19b]
  Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, Elisa Gonzalez Boix
  In: Pereira, J., Ricci, L. (eds) *Distributed Applications and Interoperable Systems.* DAIS 2019. Lecture Notes in Computer Science(), vol 11534. Springer, Cham.

  This paper discusses our initial approach to turning sequential data types into RDTs by means of a novel programming abstraction called SECRO. At runtime, replicas search for an ordering of the operations that maintains the application's invariants. The protocol guarantees that all replicas find the same ordering and thus converge to the same state. This work laid the foundations for the ECRO family of RDTs we developed later.

- **CScript: A distributed programming language for building mixed-consistency applications** [De +20]

Kevin De Porre, Florian Myter, Christophe Scholliers, Elisa Gonzalez Boix
Journal of Parallel and Distributed Computing, vol. 144, pp. 109-123 (2020).

This article is a journal extension of our conference paper about SECROs [De +19b] and focuses on CScript, a domain-specific language that extends JavaScript with built-in support for data replication. CScript features consistent and available replicated objects and regulates the interactions between those objects to avoid subtle inconsistencies when mixing consistency models.

- **ECROs: Building Global Scale Systems from Sequential Code** [De +21]
  Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix
  Proceedings of the ACM on Programming Languages, vol 5, OOPSLA, Article 107 (October 2021).

This work addresses the shortcomings of SECROs by introducing the ECRO approach which features a slightly different programming model and an improved replication protocol. Developers extend sequential data types with a distributed specification that is statically analyzed to detect conflicts and unravel their cause ahead of time. This information is then used at runtime to serialize concurrent operations safely and efficiently. Thus, our approach derives correct RDTs from sequential data types without changes to the data type implementation and with minimal coordination. We implement our approach in Scala and develop an extensive portfolio of RDTs.

- **A Contract-Based Approach for Designing Highly Available Replicated Data Types**
  Kevin De Porre, Carla Ferreira, Elisa Gonzalez Boix
  In preparation

This paper is a journal extension of our work on ECROs. It introduces EFx, a novel programming language with a contract sys-

tem that simplifies the development of RDTs using the ECRO approach. Programmers write concurrency contracts that associate high-level preconditions and invariants to the operations of sequential data types. EFx then derives correct ECROs by synthesizing a distributed specification from the data types and their contracts.

- **VeriFx: Correct Replicated Data Types for the Masses** [DFGed]
  Kevin De Porre, Carla Ferreira, Elisa Gonzalez Boix
  Submitted to ACM Transactions on Programming Languages and Systems (TOPLAS).

  This paper proposes VeriFx, a high-level programming language with *automated* proof capabilities. VeriFx lets programmers implement RDTs atop functional collections and express correctness properties that are verified automatically. Verified RDTs can be transpiled to mainstream languages. VeriFx provides libraries that implement the execution model and define the correctness properties of well-known RDT families. We used those libraries to verify 37 CRDTs and reproduce a study on the correctness of Operational Transformation functions.

In addition to the publications highlighted above, we also published two workshop articles that helped shape the ideas behind this dissertation [DG19; De +19a].

1.5 Dissertation Roadmap

This dissertation seeks to facilitate the implementation and verification of RDTs by means of suitable programming abstractions. In doing so, we must strike a good balance between efficiency and simplicity as argued before. This trade-off is reflected in the structure of this thesis which starts by proposing a simple but inefficient approach and gradually improves on it until a good balance is found. We briefly summarize the remaining chapters that constitute this dissertation:

Chapter 2: State of the Art in Geo-Replicated Systems starts by describing several data consistency models that are important to this dissertation. Then, it reviews state-of-the-art techniques for designing and implementing RDTs, thereby, making a distinction between traditional RDTs that guarantee only state convergence, and invariant-preserving RDTs that also consider application-specific invariants. The chapter finishes by reviewing general verification languages and techniques for the verification of RDTs.

Chapter 3: From Sequential to Replicated Data Types proposes a general replication protocol that can execute arbitrary operations without coordination while still guaranteeing state convergence and maintaining application invariants. This protocol is integrated in a language abstraction, called SECRO, which extends sequential data types with application-specific invariants in order to get an RDT. The resulting approach is simple as programmers only need to add the application's invariants to existing data types. Unfortunately, the underlying replication protocol is not efficient as it requires replicas to search for a correct serialization among all possible permutations of the operations.

Chapter 4: Efficient Replicated Data Types from Sequential Code proposes a novel programming abstraction, called ECROs, that extend sequential data types with a distributed specification describing the operations' preconditions and postconditions and the application's invariants. We then statically analyze the specification to derive information about non-commutative and invariant-breaking operations and find solutions to these conflicts beforehand. Using this information, our improved replication protocol can avoid the expensive search phase as it now has all the information it needs to serialize the operations efficiently (as opposed to searching for a valid serialization among all permutations). While this approach yields excellent performance, programmers need to provide first-order logic specifications that are disconnected from the actual data type implementation.

Chapter 5: A High-Level Programming Language for Efficient RDTs proposes EFx, a minimalist object-oriented programming language whose core consists of a contract system for the development of RDTs. EFx's contract system allows programmers to define precon-

ditions and invariants atop sequential data types. EFx automatically synthesizes correct ECRO specifications from sequential data types and their contracts. This considerably simplifies the development of ECROs since programmers no longer have to write tedious first-order logic specifications. Furthermore, EFx improves correctness because the synthesized specifications avoid mismatches between the specification and the implementation.

Chapter 6: Automated Verification of Replicated Data Types proposes VeriFx, a high-level programming language for the implementation and automated verification of RDTs. Programmers can implement RDTs atop functional collections and define the necessary correctness properties using VeriFx's novel proof construct. VeriFx was designed with automated verification in mind such that every language feature has an efficient encoding in first-order logic. As a result, any VeriFx program can be transpiled to first-order logic, and user-defined proofs can be discharged automatically using traditional SMT solving.

Chapter 7: Conclusion concludes this dissertation. To this end, we revisit the problem statement and provide an overview of our contributions. Finally, we identify avenues for future research.

Chapter 2

State of the Art in Geo-Replicated Systems

The previous chapter briefly described the complexity of building highly available and efficient distributed systems. We argued that better programming support is needed to implement these systems correctly and identified three key principles that form the basis of our approach.

In this chapter, we discuss the state of the art that is needed to understand our contributions. We start by reviewing the different families of consistency models in Section 2.1. Then, we describe state-of-the-art abstractions for replicated data in weakly consistent systems in Section 2.2. Finally, we turn our attention to verification techniques for distributed systems and discuss how existing approaches verify convergence properties and data integrity invariants for weakly consistent programs.

2.1 Consistency Models

Distributed systems replicate data to improve availability, scalability, and fault tolerance. Users can access data from different sources, which improves fault tolerance in the presence of partial failures. By placing replicas at strategic places across the globe, distributed systems can drastically reduce the access times observed by users. To improve scalability, incoming requests can be spread over the available replicas and replicas may be allocated dynamically depending on the load experienced by the system.

Although users are oblivious to the fact that data is replicated, replication raises potential consistency problems that may affect them. For example, a user may update their address in the system but later read their old address from a replica that has not yet received the update. The values that can possibly be read by users are described by consistency models. For example, the aforementioned behaviour cannot occur in a system whose consistency model guarantees "read your writes".

In what follows we analyze three common families of consistency models. First, Section 2.1.1 describes strong consistency models which historically were the models of predilection in databases. However, recent advances in hardware technology reshaped the landscape of distributed systems which now often face thousands or even millions of client requests per second, coming from users all over the world. For such systems, high availability and low latency are often more important than strong consistency. Section 2.1.2 introduces weak consistency; a family of models that fit these requirements as they are available under network partitions and guarantee low latency. Finally, Section 2.1.3 introduces hybrid consistency. This family of consistency models combines strong consistency and weak consistency and is especially useful to maintain application-level invariants without fully giving up on availability.

2.1.1 Strong Consistency

Strong consistency is a family of consistency models that provide users with a consistent view on the data. Thus, users have the same view on the shared data, independent of the replica they read from. This requires writes to be made visible to everyone or to no one, but nothing in between.

Linearizability [HW90] is a well-known strong consistency model and corresponds to the "C" of consistency in the CAP theorem [Bai] (cf. Section 1.1.1). A distributed system is linearizable if operations appear to be executed atomically and in an order that is consistent with their real-time ordering [Jep]. Once a write completes, all subsequent reads will observe the write (unless overwritten by a later write). To this end, writes are synchronous and require synchronization between the replicas; i.e. replicas communicate to decide whether to accept or reject writes, and writes return only when this decision has been made. However, network partitions may hamper synchronization and lead to writes being aborted. Thus, a linearizable system favors consistency over availability under network par-

titions because availability requires every request (be it a read or a write) to get a meaningful response (i.e. not a timeout).

Besides linearizability, other strong consistency models exist such as serializability [BHG87] which requires the effects of concurrent transactions in a (distributed) database to be equivalent to a serial execution of those transactions. Independent of the model, guaranteeing strong consistency comes at a cost. Network communication is slow and protocols for strong consistency (e.g. consensus algorithms) typically require several round trips. As a result, strong consistency models are not suited for latency-critical applications or massively concurrent applications because the time that is needed to synchronize replicas is high and affects the amount of concurrency that can be reached.

2.1.2 Weak Consistency

Weak consistency is a family of consistency models that tolerate temporary inconsistencies in order to guarantee high availability, even under network partitions. By relaxing the consistency guarantees, weak consistency models drastically improve scalability and reduce user-observed latencies. For example, clients can write to the nearest available replica without having to wait for the other replicas to be updated.

Although early distributed systems were mostly strongly consistent, some experimented with weak consistency. In 1982, the CAP theorem was not yet formulated when researchers designed Grapevine [Bir+82], a distributed system for message delivery, resource location, authentication, and access control whose primary use case consisted of an improved mail service. The mail service had to be available even if some Grapevine servers failed, which resulted in Grapevine being the first widely used weakly consistent distributed system.

Similarly, Bayou [Ter+95], a distributed storage system for mobile computing environments, leveraged weak consistency to cope with the unstable network connections of portable machines. Interestingly, Bayou provided novel programming abstractions to detect and solve application-specific consistency issues that arise from concurrent updates.

Nowadays, weak consistency has been adopted by many distributed systems to improve user experience through offline availability, reduced latencies, etc. Under weak consistency, replicas immediately apply updates locally and *asynchronously* propagate the updates to the other replicas.

However, concurrent updates may conflict and cause replicas to diverge. At some point, this conflict needs to be addressed in order to reconcile the replicas. When and how this happens depends on the consistency model at hand. We review common weak consistency models below:

Eventual Consistency. Eventual consistency is a form of weak consistency that guarantees that in the absence of new writes, eventually, all reads will return the value of the latest write [Vog09]. By tolerating temporary inconsistencies, eventual consistency is able to move conflict resolution off the critical path and reconcile replicas later in the background. Eventual consistency is often criticized for being vague [VV16] as it does not specify when replicas converge or how to order operations. For example, a system that ignores all updates and returns 42 on every read is considered to be eventually consistent even though it is completely useless.

Strong Eventual Consistency. Strong Eventual Consistency (SEC) strengthens eventual consistency with an additional requirement, called *strong convergence*, which requires replicas that received the same updates (possibly in different orders) to be in the same state [Sha+11b]. Thus, SEC precisely specifies when replicas converge.

SEC is a considerably stronger model than eventual consistency because the strong convergence property implies that replicas converge to equivalent states as soon as they observed the same updates. Hence, replicas do not require synchronization in order to converge.

Since SEC is a weak consistency model, replicas can issue updates concurrently. Concurrent updates have no predefined ordering and replicas may execute them in different orders. Thus, concurrent updates must commute in order to guarantee strong convergence.

Session Guarantees for Weakly Consistent Data. The aforementioned weak consistency models focus exclusively on convergence but users may still face counterintuitive behaviour due to inconsistencies between replicas. As mentioned earlier, users may write to one replica and read outdated information from another replica that has not yet observed the write. This is confusing since users regard the system as a single entity and are oblivious to the fact that data is replicated.

To capture the users' expectations, Terry et al. [Ter+94] propose four client-centric consistency models, called *session guarantees*. A session corresponds to the actions (i.e. reads and writes) performed by a user of the application. The session guarantees aim at providing meaningful consistency guarantees from the viewpoint of a user within their session such that the illusion of a single centralized server is respected[1]. The four session guarantees are:

Read Your Writes. After a write, all subsequent reads within the same session must observe the write. Thus, the application must ensure that reads reflect previous writes that occurred within the same session unless they were overwritten by later writes.

Monotonic Reads. Reads do not go back in time within a session. More precisely, when a read observes a set of writes, all subsequent reads within that session will observe a superset of those writes.

Writes Follow Reads. All replicas execute writes after the reads on which they depend. A write W_1 may depend on a set of writes W that were observed by a previous read R_1. As a result, write W_1 is dependent on all the writes in W due to read R_1; thus, all replicas should execute the writes in W before executing W_1.

Monotonic Writes. Writes are executed only after all previous writes of the same session were executed. This guarantee is only concerned with prior session writes whereas in Writes Follow Reads writes may depend on reads that observed writes from other sessions.

Interestingly, all session guarantees, except Read Your Writes, can be achieved with high availability [Bai+13]. The problem with Read Your Writes is that users can always write to one replica and read from another replica that has not yet observed the write. Still, applications do not need to completely give up on availability to guarantee Read Your Writes. It suffices for clients to stick to the same replica in order to observe prior session writes; such a system is said to be *sticky available* [Bai+13]. Many systems are already sticky available. For example, users of peer-to-peer systems read and write to a local replica which trivially guarantees Read

[1]The session guarantees were originally presented for client-server architectures but can be generalized to any architecture; replicas do not have to be servers but could also be peers in a peer-to-peer system.

Your Writes and Monotonic Reads since any write made by a peer will always be reflected by its local replica.

Causal and Causal$^+$ Consistency. Causal consistency [Aha+95] requires replicas to respect the order of causally-related operations. If an operation o_1 happened before an operation o_2 then all replicas should execute them in that order. As an example, causal consistency is used by chat applications to ensure that all replicas receive the messages and their replies in order; concretely, if Bob replies to Alice's message, then all users should receive Alice's message before receiving Bob's reply. Note that causal consistency does not prescribe an ordering for concurrent operations. Thus, users of a causally consistent chat application may receive concurrent messages in different orders.

Causal consistency corresponds to the combination of the four session guarantees outlined before [BSW04]. Although most session guarantees are highly available, Read Your Writes is only sticky available. As a result, causal consistency is also sticky available [Bai+13].

Causal consistency alone is not enough to ensure convergence since replicas are free to execute concurrent operations in any order. To guarantee convergence, a causally consistent system must also ensure that replicas deterministically solve conflicts between concurrent operations. Such systems are said to guarantee causal$^+$ consistency.

Other Weak Consistency Models. The weak consistency models described above are by no means exhaustive. A plethora of weak consistency models exist [VV16] but we focused this discussion on the models that are important for this dissertation.

2.1.3 Hybrid Consistency

The consistency models reviewed so far can be categorized as strongly consistent or weakly consistent and lay at opposite sides of the consistency spectrum. However, some applications benefit from a hybrid model in which some writes are weakly consistent while others are strongly consistent. For example, banking applications may require balances to remain positive. Some operations, such as deposits and transfers, are safe as they cannot break this invariant. Safe operations can be weakly consistent in

order to be available and fast. However, some operations, such as withdrawals, are unsafe because concurrent invocations may render balances negative. To avoid overdrafts, withdrawals must be strongly consistent. Naturally, hybrid consistency models aim to minimize the number of operations that are strongly consistent because they are slow and unavailable under network partitions [Ter+13].

We now present a number of hybrid consistency models which allow safe operations to execute under weak consistency and require unsafe operations to execute under strong consistency. Hybrid consistency models are often used to ensure state convergence by coordinating non-commutative operations, and to uphold application-specific invariants by coordinating unsafe operations.

RedBlue Consistency. RedBlue consistency [Li+12] partitions operations into strongly consistent (red) operations and weakly consistent (blue) operations based on a static analysis of the data type. Operations that do not commute or break invariants are unsafe and thus labeled red, the remaining operations are safe and thus labeled blue. Blue operations are fast because they execute locally. Red operations are slow because they require coordination between the replicas.

In a RedBlue consistent system replicas may execute operations according to any serialization that respects causality and totally orders the red operations. In other words, replicas extend the partial order defined by causality and agree on an ordering of concurrent red operations. Replicas are free to execute concurrent blue operations in any order since those commute and do not break invariants. Thus, RedBlue consistency ensures state convergence and preserves application-specific invariants.

Explicit Consistency. Explicit consistency [Bal+15] is concerned with maintaining application-specific invariants. A distributed system guarantees explicit consistency if all causal serializations of the operations also maintain the invariants. Explicit consistency does not prescribe an ordering for safe (i.e. invariant-preserving) concurrent operations. It is up to the programmer to implement some form of conflict handling to ensure convergence if concurrent operations do not commute (e.g. by relying on conflict-free replicated data types, cf. Section 2.2.1.1).

Partial Order-Restrictions Consistency. Partial Order-Restrictions (PoR) consistency [LPR18] takes as input a set of restrictions over pairs of operations and coordinates only those operations at runtime. Thus, replicas are free to execute any causal serialization of the operations as long as they agree on the order of restricted operations.

PoR consistency aims to be a general hybrid consistency model that allows for fine-grained control over which operations are coordinated. Having fine-grained control over coordination is important to minimize the amount of coordination and thus ensure high availability and fast response times. However, finding a minimal set of restrictions is nontrivial and error-prone; imposing too few restrictions may lead to runtime anomalies such as divergence or broken invariants, whereas too many restrictions considerably degrade performance. Therefore, the authors propose an algorithm for determining a minimal set of restrictions. However, they did not implement the algorithm but instead manually identified restrictions for the presented use case.

Many hybrid consistency models can be implemented on top of PoR. For example, one can implement RedBlue consistency on top of PoR by introducing a restriction for every pair of red operations.

Observable Atomic Consistency. Observable Atomic Consistency (OAC) [ZH20] distinguishes between convergent operations and totally-ordered operations. Convergent operations are expected to commute, therefore, they are not coordinated and are fast. Totally-ordered operations are coordinated and are not allowed to execute concurrently with any other operation (not even convergent operations). It is up to the application programmer to decide which operations are convergent and which operations must be totally-ordered. If some replica observes a convergent operation p before a totally-ordered operation u (denoted $p \prec u$) then all replicas must execute p before u, and vice-versa, if $u \prec p$ at some replica then all replicas must execute them in that order.

Although OAC resembles RedBlue consistency (convergent operations are roughly equivalent to blue operations, and totally-ordered operations are roughly equivalent to red operations), OAC is stronger because it also imposes an ordering between convergent and totally-ordered operations whereas RedBlue only requires red operations to be totally ordered. Thus,

under OAC, only the convergent operations between two totally-ordered operations are allowed to execute in any order.

2.2 Programming Abstractions for Replication

When dealing with replicated data, strong consistency has long been the preferred consistency model. Early distributed databases were strongly consistent and even today most databases support strong consistency (e.g. Google Cloud Spanner, MongoDB, Riak, etc.). Protocols that implement strong consistency are *general* (i.e. data type independent) since they are concerned only with the order of operations or transactions but not with the semantics of those. For example, transactional databases often guarantee serializability which requires concurrent executions to be equivalent to a serial execution of those transactions in order to avoid interleaving anomalies. Thus, databases implement general mechanisms for strong consistency which frees programmers from consistency issues.

While strong consistency is conceptually simple and can be achieved by general and well-understood replication protocols, guaranteeing some form of weak consistency (e.g. eventual consistency) is more difficult. The added complexity comes from the fact that replicas may execute updates concurrently (and in different orders) which can lead to conflicts. Such conflicts are application specific and solving them is best done by application programmers as they have the necessary domain knowledge. Recall our flight reservation example from Section 1.1.3, where concurrent bookings may lead to two or more passengers reserving the same seat. This conflict can be solved in different ways, for instance by assigning the seat to the customer with the most expensive ticket, by favoring customers that are part of the company's loyalty program, or any other sensible conflict resolution policy. This example demonstrates that conflict resolution is application dependent; hence, there are no general replication protocols for weak consistency. For this reason, conflict detection and resolution is often left to application developers.

Bayou [Ter+95], a weakly consistent storage system, allowed programmers to provide additional dependency checks and merge procedures for every write to the database. Dependency checks are a mechanism for detecting application-specific conflicts and merge procedures are used to solve the detected conflicts.

Despite early systems such as Bayou, highly available databases were only popularized more recently by Dynamo [DeC+07], Amazon's highly available distributed key-value store. Dynamo was designed such that writes never fail, thereby, moving conflict resolution to the reads. Programmers can let the store handle conflicts but, as noted by the authors, the store can apply only simple (application independent) conflict resolution policies such as last writer wins. Therefore, Dynamo also allows programmers to manually solve conflicts by merging conflicting states.

2.2.1 Replicated Data Types

Since conflict resolution cannot be hidden behind weakly consistent distributed databases, programmers need to foresee all conflicts that may arise from concurrent updates and implement appropriate conflict resolution strategies. Many systems implement ad-hoc approaches to detect and solve conflicts but this is error-prone and results in brittle systems [ASB15; KB17; Sha+11b].

To avoid the pitfalls of ad-hoc conflict resolution, researchers proposed new programming abstractions, called RDTs, that serve as basic building blocks for the development of highly available distributed systems. RDTs are reminiscent of sequential data types but abstract the underlying conflict resolution algorithm. RDTs thus hide conflict resolution behind the data type; this is possible since the implementation is aware of the data type's semantics. Lately, NoSQL databases - such as Riak KV, Redis, Akka Distributed Data, and others - are adding support for RDTs.

RDTs can be categorized into two families: traditional RDTs and invariant-preserving RDTs. Traditional RDTs focus exclusively on state convergence, whereas invariant-preserving RDTs - sometimes called rich RDTs - are also concerned with maintaining application-level invariants.

We review traditional RDTs in Section 2.2.1.1 and invariant-preserving RDTs in Section 2.2.1.2. Then, we provide an overview of these state-of-the-art RDTs and whether and how they guarantee convergence and maintain application invariants in Section 2.2.1.3.

2.2.1.1 Traditional Replicated Data Types

More than three decades ago, **Operational Transformation (OT)** was proposed as a technique to achieve state convergence in the face of con-

current operations on replicated data. Several control algorithms for OT were proposed - dOPT [EG89], aDOPTed [RNG96], GOTO [Sun+98], etc. - to keep replicated data consistent based on data type-specific Inclusive Transformation (IT) functions. IT functions modify incoming operations against previously executed concurrent operations such that the modified operation preserves the intended effect. From now on, we will refer to these IT functions as "transformation functions".

Much work focused on designing transformation functions for collaborative text editing [EG89; Imi+03; RNG96; Sun+98; SCF97], but it has been shown that all of them (even some with mechanized proofs) are wrong [Imi+03; LL04; Ost+06; Ran+13].

Although most efforts focused on designing transformation functions for collaborative text editing, the OT approach can also be used to design other data types such as replicated registers and stacks. In fact, the resulting data types can be seen as RDTs *avant la lettre* since the transformation functions hide conflict resolution behind the data type.

The failure to devise correct transformation functions led researchers to abandon OT in favor of more principled approaches. Shapiro et al. [Sha+11b] proposed **Conflict-free Replicated Data Types (CRDTs)**, a family of RDTs that are carefully designed around mathematical properties that ensure conflict freeness. By design, concurrent updates cannot conflict and thus CRDTs trivially guarantee SEC.

CRDTs can be divided into two groups: state-based and operation-based CRDTs. State-based CRDTs require the state to form a join semilattice and update methods to result in monotonically non-decreasing states (i.e. updates can only make the state go up in the lattice but not down). Replicas periodically propagate their entire state which is then merged into the other replicas by computing the Least Upper Bound (LUB) of their states. It has been shown that replicas converge if the merge function is associative, commutative, and idempotent [Sha+11b].

Operation-based CRDTs execute operations locally and asynchronously propagate them to the other replicas (often times relying on a communication mechanism that delivers messages in causal order). Still, concurrent operations arrive in an arbitrary order that may be different at all replicas. Therefore, operation-based CRDTs require concurrent operations to commute in order to guarantee SEC.

Although the mathematical rules are well established, designing new CRDTs that obey those rules is difficult, even for experts [Kle22]. Therefore, several composition techniques have been proposed but none allow arbitrary compositions for all CRDTs. Lasp [MV15] enables programmers to combine CRDTs using well-known operations from functional programming, however, every CRDT must be defined in terms of transformations over existing CRDTs. It is not clear if all CRDTs can be obtained this way and how the transformations affect performance. JSON CRDTs [KB17] let programmers define new CRDTs by nesting lists and maps that support insertions, deletions, and assignments. However, those collections and their interfaces are restricted and cannot define arbitrary CRDTs. For example, one cannot define a counter CRDT by using their list CRDT to count increments and decrements because 1) programmers cannot append at the end of the list, and 2) the length of the list cannot be queried. Weidner et al. [WMM20] leverage the mathematical properties of semidirect products to combine CRDTs. However, composing CRDTs using semidirect products is complex because programmers need to define arbitration orders between the CRDTs' operations, they also need to transform concurrent operations, and sometimes they even have to modify the original CRDTs.

Burckhardt et al. [Bur+12] proposed **cloud types**, a collection of RDTs that guarantee eventual consistency. In contrast to the original CRDT model, programmers can define custom data schemas for their applications by composing cloud integers, cloud arrays, etc. However, the proposed cloud types have predefined merge semantics. If the application requires different semantics, programmers are bound to implement new cloud types and the accompanying merge procedures, which means they are back into ad-hoc conflict resolution.

To free programmers from having to manually engineer merge functions, **Mergeable Replicated Data Types (MRDTs)** leverage invertible relational specifications defined by the programmer in order to derive correct merge functions for inductive data types. Such specifications consist of an abstraction function that transforms the data type to relations over sets, and a concretization function that maps set relations back into the data type. Replicas can then be merged automatically by transforming their states to relations over sets, then merging those relations using a

pre-defined merge function for sets, and finally transforming the merged set relations back into the data type.

Programmers of MRDTs can, however, not tweak the conflict resolution semantics as they are hardcoded by the underlying merge semantics of sets. In addition, programmers need to translate high-level data types to low-level set relations which is nontrivial.

2.2.1.2 Invariant-Preserving Replicated Data Types

While traditional RDTs guarantee some model of weak consistency, they do not consider application-level invariants. Operations may maintain invariants locally but break them when executed concurrently.

Several approaches have been proposed to extend existing RDTs with mechanisms to maintain application-specific invariants. Sieve [Li+14] lets programmers extend database schemas with annotations describing the desired conflict resolution semantics and application-specific invariants. Sieve analyzes database schemas (using a combination of static analysis and runtime checks) to detect potential invariant violations and label operations as red or blue and thus guarantee RedBlue consistency. Internally, Sieve leverages CRDTs to ensure convergence.

Indigo [Bal+15] lets programmers annotate classes with first-order logic invariants in Java and requires operations to be annotated with their effects. The specification is then statically analyzed to detect unsafe operations, i.e. operations that violate invariants when executed concurrently. To cope with unsafe operations, programmers can choose between two strategies: invariant repair and violation avoidance. The former allows unsafe operations to execute concurrently but requires the conflict resolution code to be adapted to repair broken invariants. The latter strategy coordinates unsafe operations such that they cannot execute concurrently and hence cannot break invariants. In both cases, Indigo guarantees explicit consistency.

Invariant-Preserving Applications (IPA) [Bal+18] is a static analysis tool that builds on the invariant repair strategy of Indigo. Like Indigo, programmers write specifications for RDTs, which are then statically analyzed by the tool to detect unsafe operations. However, IPA can also suggest source code modifications to make operations invariant preserving. To this end, it returns an updated specification of the application comprising the necessary modifications. It is the programmer's responsi-

bility to modify the application according to this new specification. Note that the proposed modifications affect only concurrent operations and thus do not change the semantics of sequential operations. For example, in a courseware system a student may enroll for some course while concurrently someone deletes the student's account. In this case, the system may end up in a state where the student is enrolled in the course but no longer exists in the system. This breaks referential integrity, a common database invariant. IPA can suggest a modification of the enroll operation that first creates an account for the student if the student no longer has one and only then enrolls the student in the course.

Quelea [SKJ15] is a declarative programming model that features contracts for expressing application-specific consistency properties. Contracts are first-order logic expressions that describe the set of legal executions by means of primitive consistency relations. Quelea automatically maps these contracts to an appropriate consistency level (eventual consistency, causal consistency, or strong consistency).

Q9 [Kak+18] provides a library of RDTs and uses symbolic execution to detect invariant violations in weakly consistent applications built atop those RDTs. To avoid these violations, Q9 selectively strengthens the consistency guarantees of specific operations to find the weakest model under which the violations disappear. However, Q9's symbolic execution engine considers only a bounded number of concurrent operations.

Hamsaz [HL19] and Hampa [LHL20] statically analyze specifications of sequential objects provided by the programmer in order to derive coordination protocols that guarantee state convergence and preserve invariants. The derived protocols coordinate all non-commutative operations in order to guarantee state convergence, and coordinate all unsafe method calls in order to preserve application invariants. Hampa also provides additional recency guarantees.

2.2.1.3 Overview

Table 2.1 provides an overview of state-of-the-art RDTs and how they guarantee state convergence and preserve application invariants. The traditional RDTs (OT, CRDTs, cloud types, MRDTs) exploit different approaches to guarantee state converge but do not support application-specific invariants. Indigo, IPA, Sieve, and Q9 leverage CRDTs to ensure state convergence and extend them with mechanisms to maintain

	Model	**State Convergence**	**Application Invariants**
OT	SEC	By transforming concurrent operations	Not supported
CRDTs	SEC	By design	Not supported
Cloud Types	EC	Using user-defined merge procedures	Not supported
MRDTs	SEC	Using derived three-way merge procedures	Not supported
Indigo	Hybrid	By relying on CRDTs	Coordinates unsafe ops or repairs broken invariants
IPA	SEC	By relying on CRDTs	Suggests modifications of the operations
Sieve	Hybrid	By relying on CRDTs	Coordinates unsafe operations
Q9	Hybrid	By relying on CRDTs	Picks weakest consistency model that upholds contract/invariants
Quelea	Hybrid	Guaranteed by the underlying database (Cassandra)	Picks weakest consistency model that upholds contract/invariants
Hamsaz & Hampa	Hybrid	By coordinating non-commutative ops	Coordinates unsafe operations

Table 2.1: Overview of state-of-the-art RDTs.

application-specific invariants. However, those approaches do not help build the CRDTs in the first place. Thus, they do not simplify the development of custom RDTs. Hamsaz and Hampa support the development of custom RDTs by synthesizing the RDT from a specification of the sequential object and its invariants. However, these specifications are expressed using low-level formalisms (often first-order logic) which makes them difficult to write and error-prone. This is problematic since correctness of the synthesized RDT depends on the input specification. Similarly, Quelea allows programmers to associate contracts describing application-specific invariants to the objects of an eventually consistent data store, but requires contracts to be written in a separate contract language that uses low-level consistency relations such as visibility and session order.

As identified in our research vision (Section 1.2), we believe that RDT solutions must 1) replicate existing data types with application-specific logic instead of designing custom RDTs for each use case, 2) be correct out-of-the-box, and 3) be integrated in a suitable programming language

abstraction. However, none of the aforementioned approaches adhere to all three principles. OT, CRDTs, cloud types, and MRDTs infringe the first principle because they require dedicated implementations for each data type. The remaining approaches infringe the second and third principles because correctness depends on a separate specification that is written in some low-level formalism. For example, Indigo and IPA require programmers to annotate Java code with first-order logic postconditions and invariants written as plain strings. Similarly, Sieve, Quelea, Hamsaz, and Hampa require formal specifications written in a separate language. Q9 is the only approach that analyzes high-level RDTs instead of low-level specifications and thus does not infringe the third principle.

2.3 Distributed Systems Verification

We explained that programmers of distributed systems leverage RDTs to avoid ad-hoc conflict resolution which is hard and error-prone [ASB15; KB17; Sha+11b]. Instead, RDTs hide these conflict resolution mechanisms behind the data type. Thus, RDTs are exclusively designed by experts and are then picked up by application programmers who implement those designs and sometimes even modify them to fit their application.

As identified in the problem statement (Section 1.1.3), the current workflow for the development of RDTs is not bulletproof. Experts may for instance miss subtle corner cases, or programmers may wrongly implement RDT specifications as they do not understand the subtleties underlying those designs. At runtime, such errors may lead to unintended behavior (e.g. divergence), called bugs. To detect bugs it is common practice among programmers to write tests that check that the system and its components behave as expected. While tests are relatively easy to write, they do not guarantee to uncover all bugs as they may for instance miss problematic inputs. In fact, bugs in RDTs manifest because the RDT designer or implementer missed a certain corner case, often a special combination of sequential and concurrent operations. However, if they did not consider that corner case during the design or implementation, they are unlikely to consider it during testing, and thus will not detect the bug.

To ensure that programs are bug-free, they can be formally verified; that is, they can be proven to guarantee certain properties. For example, programmers may write pen-and-paper proofs to manually verify that

an RDT converges or upholds a certain invariant. Although most computer scientists are somewhat familiar with paper proofs, such proofs are still subject to reasoning flaws that render them obsolete. To avoid reasoning flaws, programmers can mechanically verify their programs and algorithms. For example, programmers can formalize the RDT in a verification language and prove the necessary consistency properties. The resulting proofs are then checked by the underlying language to ensure that they do not contain (mathematical) reasoning flaws. For this reason, mechanical proofs are more convincing, but they require significant programmer intervention which is time-consuming and reserved to verification experts [LM10; OHe18].

In the remainder of this section, we review state-of-the-art verification approaches for RDTs. Before we delve into verification of RDTs, we review general-purpose verification languages, which can also be used to verify RDTs. Then, we turn our attention to the verification of RDTs and invariants for distributed systems in Sections 2.3.2 and 2.3.3, respectively.

2.3.1 Verification Languages

Verification tools, languages, and techniques can be classified into three categories: interactive, auto-active, and automated [LM10]. Interactive verification includes proof assistants like Coq and Isabelle/HOL in which programmers define theorems and prove them manually using proof tactics. Although some automation tactics exist, proving complex theorems requires considerable manual proof efforts since every step of the proof has to be written explicitly and seemingly trivial properties have to be proven explicitly in turn. A recent example is Liquid Haskell [Vaz+14] in which programmers specify correctness properties using refinement types and write proofs using plain Haskell functions. Proofs can be assisted or in some cases even fully discharged by the underlying Satisfiability Modulo Theories (SMT) solver. However, advanced proofs require significant manual proof efforts because programmers need to manually prove the parts where the SMT solver fails. For example, mainstream SMT solvers do not apply inductive reasoning, a common proof technique. Therefore, programmers need to manually write inductive proofs in Liquid Haskell.

Auto-active verification aims to automate the verification process such that programmers do not have to manually write proofs, but requires additional information from the programmer. For example, languages like

Dafny [Lei10] and Spec# [BLS05] verify programs for runtime errors and user-defined invariants but require programmers to annotate the code with preconditions, postconditions, loop invariants, etc. These annotations essentially form a specification of the code which the verification language uses to verify the program by leveraging the automated verification capabilities of Intermediate Verification Languages (IVLs).

Finally, mainstream *automated* verification techniques consist of IVLs and SMT solvers. IVLs like Boogie [Bar+06] and Why3 [FP13] automate the proof task by generating Verification Conditions (VCs) from the source code and discharging them using one or more SMT solvers. However, fully automated verification of programs written in IVLs may fail due to the way how they are encoded in SMT solvers; e.g. verifying properties about recursive functions will fail due to the need for inductive reasoning. Similarly, verification of programs written directly in SMT may also fail due to the use of undecidable theories. Instead of using IVLs or SMT solvers directly, programmers use auto-active verification languages, which, internally, translate the program to IVLs to verify the VCs.

The aforementioned verification languages are general such that programmers can use them to verify arbitrary properties of any program. As a result, these languages require significant programmer intervention because they cannot automate the verification of arbitrary language features. For example, imperative programs may mutate variables in a loop to compute the sum of an array of integers. One may use Dafny to verify that the loop does not index the array out of bounds, but this requires them to specify an appropriate loop invariant. Otherwise, Dafny is not able to handle the loop. Even automated verification tools such as SMT solvers may fail to verify certain programs as not all logic theories supported by those solvers are decidable.

2.3.2 Verifying Correctness of Replicated Data Types

We now review existing approaches that verify the correctness properties of RDTs. In particular, Section 2.3.2.1 focuses on verifying convergence properties for OT, and Section 2.3.2.2 focuses on verifying SEC for well-known RDTs such as CRDTs.

2.3.2.1 Verifying Convergence for Operational Transformation

We explained that OT transforms incoming operations against previously executed concurrent operations. Operations are functions from state to state: $Op : \Sigma \to \Sigma$ and are transformed using a type-specific transformation function $T : Op \times Op \to Op$. Thus, $T(o_1, o_2)$ denotes the transformation of o_1 against a previously executed concurrent operation o_2.

Ressel et al. [RNG96] proved that replicas eventually converge if the IT function satisfies two properties. The first property, TP_1, requires concurrent operations o_i and o_j to commute after transforming them:

$$\forall o_i, o_j \in Op, \forall s \in \Sigma : T(o_j, o_i)(o_i(s)) = T(o_i, o_j)(o_j(s))$$

The second property, TP_2, requires that the transformation of incoming operations o_k does not depend on the order in which previously observed operations o_i and o_j are transformed:

$$\forall o_i, o_j, o_k \in Op, \forall s \in \Sigma : T(T(o_k, o_i), T(o_j, o_i)) = T(T(o_k, o_j), T(o_i, o_j))$$

Unfortunately, the original transformation functions proposed by Ellis and Gibbs [EG89] do not satisfy the aforementioned properties [Sun+98; RNG96; SCF98]. Over the years, several transformation functions were proposed [RNG96; Sun+98; SCF97] to address the problems of prior functions. Imine et al. [Imi+03] used SPIKE [BR95], an automated theorem prover, to verify TP_1 and TP_2 for all these transformation functions, and found counterexamples for each of them, except for the transformation functions of Suleiman et al. [SCF97]. They then proposed a simplified set of transformation functions but Li and Li [LL04] later found a manual counterexample that shows that the transformation functions of Imine et al. [Imi+03] and Suleiman et al. [SCF97] do not satisfy TP_2. In turn, Li and Li [LL04] proposed their own transformation functions but Oster et al. [Ost+06] found a counterexample with the help of SPIKE.

2.3.2.2 Verifying SEC for Replicated Data Types

With the widespread of strong eventually consistent RDTs, such as CRDTs and MRDTs, several verification techniques have been proposed. Some are used in paper proofs, while others are mechanised and fall into our earlier classification of interactive, auto-active, and automated verification techniques. We discuss several mechanised RDT verification approaches

that fall into the categories of interactive and automated techniques, but we are not aware of any auto-active verification techniques for RDTs.

Formal techniques for paper proofs. [Bur+14] proposes a formal framework to specify and manually verify RDTs using replication-aware simulations in paper proofs. [Att+16] uses a variation on this framework to provide precise specifications of replicated lists - which form the basis of collaborative text editing - and prove the correctness of an existing text editing protocol on paper. [LF21] proposes a new correctness criterion for CRDTs that extends SEC with functional correctness and enables manual verification of CRDT implementations and client programs. Their focus is mainly on functional correctness and they provide paper proofs. [JR18] introduces a notion of validity for RDTs and manually proves validity for some CRDTs. Although these formalisms are useful to reason about RDTs, we believe that RDTs should be verified mechanically to avoid subtle reasoning flaws from creeping into the proofs.

Interactive verification techniques. [Gom+17] and [ZBP14] propose formal frameworks in the Isabelle/HOL theorem prover to interactively verify SEC for CRDT implementations. [Liu+20] extends Liquid Haskell [Vaz+14] with typeclass refinements which are used to prove SEC for several CRDT implementations. While simple proofs can be discharged automatically by the underlying SMT solver, advanced CRDTs often cannot be verified automatically. This requires programmers to manually write proofs using Liquid Haskell's theorem proving facilities. [Nie+22] developed libraries to implement and verify operation-based CRDTs in separation logic and used them to verify a number of operation-based CRDTs. Their approach requires programmers to write Coq specifications atop the provided libraries and manually prove correctness.

Automated verification techniques. [Wan+19] proposes replication-aware linearizability, a criterion that enables sequential reasoning to verify the correctness of CRDT implementations. The authors manually encoded the CRDTs in the Boogie IVL in order to get automated correctness proofs. Those encodings are, however, non-trivial and differ from real-world CRDT implementations. [NJ19] developed a proof rule that is parametrized by the consistency model and automatically checks conver-

gence for CRDTs. The framework operates on a first-order logic specification of the CRDT but due to imprecisions the framework may reject correct CRDTs.

2.3.3 Verifying Invariants

Researchers are also actively devising formal techniques to reason about data integrity invariants in weakly consistent distributed systems and verify them. Again, we classify mechanised verification techniques as interactive, auto-active, or automated. While some of these techniques are auto-active or automated, we are not aware of interactive techniques to verify invariants in weakly consistent distributed systems.

Auto-active verification of invariants. Repliss [ZBP20] is an auto-active verification tool to verify highly available programs written in their Domain-Specific Language (DSL). Programmers write additional specifications defining functional properties using invariants that refer to the state and to the history of operations. Using symbolic execution, these properties are verified per operation, by assuming that the invariants hold in the initial state and then proving that they also hold after applying the operation. However, if the provided invariants are too weak the tool may fail to verify the properties for the post-state, in which case the programmer must provide stronger invariants. The authors used Repliss to verify invariants such as referential integrity for a weakly consistent chat application built atop CRDTs.

Automated verification of invariants. Invariant confluence [Bai+14] is a correctness criterion for coordination avoidance. In essence, invariant confluent operations maintain application invariants even when executed concurrently. Thus, invariant confluent operations are safe and do not require coordination. In follow-up work, [WH18] devise a decision procedure for invariant confluence that can be checked automatically by their interactive system written in Python. Programmers can write specifications of objects and their invariants in a DSL in Python and interact with the decision procedure to check invariant confluence. Internally, the decision procedure compiles the object and its invariants into a formula whose satisfiability is checked using the Z3 [MB08] SMT solver. Soteria [NPS20]

automatically verifies program invariants for state-based replicated objects based on the invariant confluence criterion. Soteria is written in the Boogie [Bar+06] IVL and requires programmers to provide a specification of the object written in Boogie together with some additional domain-specific annotations. Similarly, CISE [Got+16] proposes an automated proof rule to check if a given consistency model maintains a particular data integrity invariant. To this end, programmers write low-level SMT specifications for the data type operations and invariants. The tool then generates the necessary proof obligations and checks them using Z3. The aforementioned approaches focus on data integrity invariants and assume that the system guarantees convergence.

2.3.4 Overview

We reviewed techniques to verify RDTs and invariants in weakly consistent distributed systems. Some formal techniques are used in paper proofs while others are mechanised. Although formalisms for paper proofs are useful to reason about RDTs, they are prone to subtle reasoning flaws. Therefore, we argue that verification techniques should be mechanised.

Tool	**Category**	**SEC**	**Inv**	**Input**
[Gom+17]	interactive	✓		Implementation and proofs in Isabelle/HOL
[ZBP14]	interactive	✓		Specifications and proofs in Isabelle/HOL
[Liu+20]	interactive	✓		Refinement types and proofs in Liquid Haskell
[Nie+22]	interactive	✓		Specifications and proofs in Coq
Repliss [ZBP20]	auto-active		✓	Specifications written in DSL
RA linearizability [Wan+19]	automatic	✓		Specifications written in Boogie
[NJ19]	automatic	✓		Specifications in first-order logic
[WH18]	automatic		✓	Specifications written in Python DSL
Soteria [NPS20]	automatic		✓	Specifications written in Boogie
CISE [Got+16]	automatic		✓	Specifications written in SMT

Table 2.2: Overview of mechanised verification techniques for RDTs. Thicks indicate if the technique has been applied to verify SEC or application-specific invariants.

Table 2.2 provides an overview of the mechanised verification approaches discussed in this section. The discussed interactive tools are integrated in proof assistants and were used to manually verify SEC for a number of CRDTs. However, writing interactive proofs requires advanced knowledge of proof assistants and is very time-consuming. We discussed two automated approaches to verify SEC for CRDTs but they operate on low-level specifications that are disconnected from actual implementations. Similarly, auto-active and automated approaches have been proposed for verifying application-specific invariants in weakly consistent distributed systems but all of them operate on disconnected specifications.

We conclude that verification of RDTs is gaining a lot of traction, but there is still no approach that can derive correctness proofs automatically for high-level RDT implementations. All approaches require manual proof efforts or operate on a disconnected specification. In order for verification approaches to be effective, we argue that they should be automated and integrated into high-level programming languages such that programmers can verify RDT implementations directly without having to go through an additional verification step in a separate language.

2.4 Conclusion

This chapter reviewed the state of the art in distributed systems and started with an overview of data consistency models and their guarantees. We then introduced RDTs and categorized them into traditional RDTs that only guarantee state convergence, and invariant-preserving or rich RDTs that also maintain application invariants. Afterward, we reviewed formal verification techniques and how they are applied to distributed systems, in particular to RDTs and program invariants.

Although distributed systems and formal verification are often regarded as disconnected, they are closely related. In fact, formal verification should be applied during the development of distributed systems to verify key properties such as convergence, invariant preservation, etc. However, these two fields often operate on different levels of abstraction and require different skills, which leads to thinking they are disconnected. For example, distributed systems are implemented in mainstream languages (C++, Java, Erlang, Go, etc.) whereas formal verification often consists of abstract specifications in specialized verification languages (Coq,

Isabelle, etc.) that do not resemble traditional implementations. The difference in abstraction level blurs the connection between those fields.

In this dissertation, we unify both fields by enabling programmers to build distributed systems and formally verify them, without requiring expertise in verification. We first present principled approaches for the development of custom RDTs that support application invariants and are correct out-of-the-box. Afterward, we integrate formal verification into a high-level programming language such that the implementation and verification of RDTs both operate on the level of executable implementations which feels familiar to programmers.

Chapter 3

From Sequential to Replicated Data Types

As explained in Section 1.1, distributed systems replicate data under weak consistency to ensure high availability and low latencies, and to improve the system's overall scalability and fault tolerance. However, programming under weak consistency is difficult due to conflicts that arise from concurrent updates. To avoid manual conflict resolution, programmers rely on RDTs when building collaborative applications.

Unfortunately, the literature proposes only a limited portfolio of RDTs exhibiting hardcoded conflict resolution semantics. When that portfolio falls short, programmers can resort to two solutions. One is to fall back to manual conflict resolution which requires programmers to completely rethink their data structures. This cannot be reasonably expected from programmers as even experts make mistakes when designing basic RDTs such as maps [Kle22]. Alternatively, programmers can try to compose existing RDTs, but, as explained in Section 2.2.1.1, current techniques [MV15; KB17; WMM20] do not support arbitrary compositions.

In this chapter, we develop a new and radically different programming abstraction that allows programmers to extend sequential data types with application-specific information in order to derive custom RDTs. Our approach is called Strong Eventually Consistent Replicated Object (SECRO) and results in a novel family of RDTs that guarantee SEC by computing a conflict-free serialization of the operations. Programmers can tailor the semantics of SECROs by restricting the set of valid serializations through

concurrent pre and postconditions defined over the data type's operations. Our approach embraces the idea that conflict resolution depends on the semantics of the application [Ter+95].

The remainder of this chapter is organised as follows. First, we discuss several possibilities to guarantee state convergence without coordination between replicas, in Section 3.1. Second, we define SECROs in Section 3.2, use it to build a collaborative text editor, and discuss its replication protocol in depth. Third, we evaluate our approach with regard to memory consumption and latency of operations, in Section 3.3. Finally, we compare our approach to related work and conclude.

3.1 State Convergence Without Coordination

As explained in Section 2.1.2, SEC requires replicas that received the same operations to be in equivalent states. This implies that replicas solve conflicts independently and converge without coordination.

There are two ways to achieve convergence without coordination. The first consists of solving the conflicts that arise when concurrent operations are executed in different orders by the replicas. By solving these conflicts, concurrent operations are made commutative and the RDT effectively converges independently of the order in which concurrent operations are executed. This approach corresponds to what most RDT families do (cloud types, CRDTs, etc.) but puts additional burden on the RDT designers because they must identify all possible conflicts and tweak the data type implementation such that replicas deterministically solve them.

The second approach consists of imposing a total order on concurrent operations such that all replicas execute concurrent operations in the same order. As a result, concurrent operations do not need to commute. When rollbacks are allowed, replicas can reorder already executed operations to achieve this total order without coordination. In contrast to the first approach, it is possible to devise a *general* replication protocol that serializes concurrent operations according to a total order and thus works on arbitrary data types. In fact, IceCube [Ker+01] leverages this idea to build a general-purpose reconciliation engine for diverged replicas (cf. Section 3.4 for a more detailed discussion).

The SECRO approach presented in this chapter also explores the second option; its replication protocol ensures that all replicas execute oper-

ations according to a total order that 1) respects causality between operations, and 2) respects application-specific invariants defined by the programmer. Central to SECROs is the idea of employing application-specific information to restrict the set of valid serializations to only those that exhibit the desired concurrency semantics. The resulting approach is general and *automatically* derives custom RDTs from sequential data types.

In the next section, we define the SECRO data type, demonstrate its use through the implementation of a collaborative text editor, and present the underlying replication protocol that serializes operations.

3.2 Strong Eventually Consistent Replicated Objects (SECROs)

A SECRO is an object representing a data type that can be replicated to a group of devices. Like objects in Object-Oriented Programming (OOP), SECROs contain state in the form of fields, and behaviour in the form of methods. These methods form the SECRO's public interface and can be categorized into *accessors* (i.e. query methods) and *mutators* (i.e. methods that update the internal state).

Mutators can have associated *state validators* which define the data type's behaviour in the event of concurrent operations. State validators restrict the set of valid operation serializations and come in two forms:

Preconditions. A predicate that must be true before its associated operation can be executed. If the predicate is false, the operation is aborted and this specific serialization of the operations is invalidated.

Postconditions. A predicate that must hold after its associated operation and *all* concurrent operations executed. Postconditions verify the state that results from a group of concurrent, potentially conflicting, operations. If the postcondition of an operation is false it invalidates the entire serialization which will then be discarded.

SECRO replicas execute operations locally and propagate them asynchronously. When receiving an operation, replicas compute the set of all operations that are directly or transitively concurrent to the received operation (because those may conflict) and search for a valid serialization

of those operations. A serialization is valid if it respects the causality of operations[1] and passes the state validators described above.

3.2.1 Use Case: A Collaborative Text Editor

We now demonstrate how to build RDTs using SECROs by means of a real-world example. All code snippets are in CScript [De +20], our JavaScript extension for distributed programming that features built-in support for data replication and embodies our implementation of SECROs.

Consider the case of a collaborative text editor that organizes text documents as a *balanced* tree of characters in order to ensure logarithmic time insertions and deletions. Documents are replicated and support insertions and deletions of characters at specification positions in the text. To ensure good user experience and offline availability, the editor should be eventually consistent. However, building such a text editor is challenging because most tree RDTs are not balanced or require coordination to rebalance (e.g. [Néd+13]). Moreover, it does not seem possible to efficiently implement a balanced tree RDT by composing existing RDTs such as lists and maps. Instead, we will use SECROs to turn an existing AVL tree data type into a replicated text editor.

Listing 3.1 shows the structure of the replicated `Document` which extends the `SECRO` class (Line 1). Internally, the `Document` class has one field, called `_tree`, which holds a balanced tree of characters (Lines 2 to 4). We did not implement our own tree data structure but instead use an existing AVL tree data structure provided by the Closure library[2] which handles insertions and deletions in the tree as well as rebalancing the tree for us.

The `Document` SECRO defines three accessors (`containsId`, `generateId` and `indexOf`) and two mutators (`insertAfter` and `delete`). `containsId` returns a boolean that indicates the presence or absence of a certain identifier in the document tree. `generateId` uses a boundary allocation strategy [Néd+13] to compute stable identifiers based on the reference identifiers. This ensures that the identifiers reflect the position in the document and do not change throughout the lifetime of the document (see Appendix A for a detailed explanation). `indexOf` returns the index of

[1] Since concurrency is not a transitive relation some operations in the set may exhibit causal relations which must be respected.

[2] `https://developers.google.com/closure/library/`

Listing 3.1: Structure of the text editor.

```
1  class Document extends SECRO {
2    constructor(tree = new AvlTree((c1,c2) => c1.id-c2.id)){
3      this._tree = tree;
4    }
5    @accessor
6    containsId(id) {
7      const dummyChar = {char: '', id: id};
8      return this._tree.contains(dummyChar);
9    }
10   @accessor
11   generateId(prev) { /* see Appendix A */ }
12   @accessor
13   indexOf(char) {
14     return this._tree.indexOf(char);
15   }
16   // operations to manipulate the tree
17   insertAfter(id, char) { /* see Listing 3.2 */ }
18   delete(id) { /* see Listing 3.3 */ }
19   // State validators
20   pre insertAfter(doc, args) { /* see Listing 3.2 */ }
21   post insertAfter(ogDoc, doc, args, newChar) {
22     /* see Listing 3.2 */
23   }
24   post delete(ogDoc, doc, args, res) {/* see Listing 3.3 */}
25 }
```

a character in the document tree[3]. Note that the accessors are side-effect free and are annotated with `@accessor`, otherwise, CScript treats them as mutators. The `insertAfter` mutator inserts a new character `char` after an existing character identified by `id` (called the reference character). The `delete` mutator deletes the character identified by the given `id`.

Listing 3.2 shows the implementation of the `insertAfter` mutator. The mutator generates a new stable identifier for the character based on the identifier of the reference character (Line 2). The character is then tagged with this new identifier (Line 3). Finally, the tagged character is inserted in the tree and returned from the mutator (Lines 4 and 5).

The `insertAfter` operation has two state validators:

A precondition (Lines 7 to 10) that checks that either the reference character is `null` (in order to prepend a character to the document) or it exists in the document tree (Line 9). This precondition ensures that if a concurrent `delete` removes the reference character, the

[3] All characters in the tree are unique because they are tagged with unique identifiers.

Listing 3.2: Inserting a character in a tree-based text document.

```
1  insertAfter(id, char) {
2    const newId   = this.generateId(id),
3          newChar = new Character(char, newId);
4    this._tree.add(newChar);
5    return newChar;
6  }
7  pre insertAfter(doc, args) {
8    const [id, char] = args;
9    return id === null || doc.containsId(id);
10 }
11 post insertAfter(originalDoc, newDoc, args, newChar) {
12   const [id, char] = args,
13         refChar = {char: "dummy", id: id};
14     if (id === null) // we prepended `newChar` to the document
15       return doc._tree.contains(newChar);
16     else // `newChar` must occur after `refChar`
17       // if `refChar` was deleted, `indexOf` will return -1
18       return doc.indexOf(refChar) < doc.indexOf(newChar);
19 }
```

replica will first insert this character before deleting the reference character. Preconditions have the same name as their associated mutator and take as parameters the object's current state followed by an array containing the arguments of the call. In this case, `args` is an array containing two values: the `id` and `char` that are passed to the call of `insertAfter` (Line 8).

A postcondition (Lines 11 to 19) that checks that the newly added character occurs at the correct position in the resulting tree, i.e. after the reference character that is identified by `id`. According to this postcondition, any interleaving of concurrent character insertions is valid, e.g. two users may concurrently write "foo" and "bar" resulting in one of: "foobar","fboaor", etc. If the programmer only wants to allow the interleavings "foobar" and "barfoo" the SECRO must operate on the granularity of words instead of characters. Note that postconditions in CScript take 4 arguments: 1) the object's state before the call to the mutator, 2) the state that results from applying the mutator, 3) an array with the arguments of the call, and 4) the mutator's return value (`newChar` in this case).

Listing 3.3: Deleting a character from a tree-based text document.

```
1 delete(id) {
2   return this._tree.remove(id);
3 }
4 post delete(originalDoc, doc, args, res) {
5   const [id] = args;
6   return !doc.containsId(id);
7 }
```

Listing 3.3 shows the implementation of the `delete` mutator and its associated postcondition. Lines 1 to 3 delete a character by removing it from the underlying AVL tree using the character's unique identifier. The postcondition ensures that the character no longer occurs in the tree (Lines 4 to 7).

3.2.2 Replication Protocol

As exemplified by the collaborative text editor, SECROs extend sequential data types with state validators in order to get an RDT that guarantees SEC and maintains application-specific invariants. To provide these guarantees SECROs implement a dedicated optimistic replication protocol.

Every replica runs the SECRO protocol and propagates update operations asynchronously to the other replicas. In contrast to CRDTs, the operations of a SECRO do not necessarily commute. Therefore, the replication protocol totally orders the operations at all replicas *without* requiring coordination between them. This order may not violate any of the operations' pre or postconditions.

Intuitively, replicas maintain their *initial state* and a sequence of operations called the *operation history*. Each time a replica receives an operation, it is added to the replica's history. This may require reordering parts of the history which boils down to finding an ordering of the operations that fulfils two requirements. First, the order must respect the causality of operations. Second, applying all the operations in the given order may not violate any of the state validators. An ordering that adheres to these requirements is called a *valid execution*. As soon as a valid execution is found each replica resets its state to the initial one and executes the operations in order. Note that replicas must deterministically reorder the

history such that replicas that received the same operations find the same valid execution. How this is done will be explained later.

The existence of a valid execution cannot be guaranteed. For example, pre and postconditions may contradict. It is the programmer's responsibility to provide correct pre and postconditions.

The replication protocol provides the following guarantees:

1. Eventually, all replicas converge towards the same valid execution (i.e. eventual consistency).

2. Replicas that received the same updates have identical operation histories (i.e. strong convergence).

3. Replicas eventually perform the operations of a valid execution if one exists, or issue an error if none exists.

The operation histories of replicas may grow unboundedly as they perform operations. In order to alleviate this issue replicas periodically *commit* their state. To this end, replicas maintain a *version number*. Whenever a replica commits, it clears its operation history and increments its version number. The replication protocol then notifies all other replicas of this commit, which adopt the committed state and also empty their operation history. From that point on, all incoming operations that apply to a previous version number will be ignored. As we will explain later, commits commute with all operations (including other commits) and thus do not require coordination between the replicas which ensures high availability. However, commits may cause concurrent operations to be dropped to keep the history bounded.

3.2.2.1 Algorithm

We now detail the algorithm behind the SECRO replication protocol. The algorithm makes the following assumptions:

- Operations are propagated using causal order broadcasting, i.e. a communication medium that delivers messages in an order that is consistent with the happened-before relation [Lam78]. Still, concurrent operations may arrive in arbitrary orders at the replicas.

- Nodes in the network may contain any number of replicas.

- Nodes maintain vector clocks to timestamp the operations of a replica.
- Nodes generate globally unique identifiers.
- Reading the state of a replica happens side-effect free and mutators solely affect the replica's state (i.e. the side effects are confined to the replica itself).
- Eventually all messages are delivered, i.e. reliable communication: no message loss nor duplication (e.g., TCP/IP).
- There are no byzantine failures, i.e. no malicious nodes.

A replica r is a tuple $r = (v_i, s_0, s_i, h, id_c)$ consisting of the replica's version number v_i, its initial state s_0, its current state s_i, its operation history h, and the id of the latest commit operation id_c. A mutator m is represented as a tuple $m = (o, p, a)$ consisting of the update operation o, precondition p, and postcondition a. The operation is a function that takes the current state, executes the operation, and returns the updated state. Similarly, the precondition and postcondition are functions that take the current state and return a boolean that indicates whether or not the precondition, respectively, the postcondition holds. We denote that a mutation m_1 happened before m_2 using $m_1 \prec m_2$. Similarly, we denote that two mutations happened concurrently using $m_1 \parallel m_2$. Both relations are based on the clocks carried by the mutators [Jua+16].

We now discuss in detail the three kinds of operations that are possible on replicas: reading, mutating, and committing state.

1. **Reading Replicas** Reading the value of a replica (v_i, s_0, s_i, h, id_c) simply returns its latest local state s_i.
2. **Mutating Replicas** When a mutator $m = (o, p, a)$ is applied to a replica a *mutate* message is broadcast to all replicas[4]. Such a message is an extension of the mutator (o, p, a, c, id) which additionally contains a logical timestamp c and a globally unique identifier id.

As explained before, operations on SECROs do not need to commute. Operations are timestamped with logical clocks which exhibit a partial

[4]In practice, we do not send the o, p, and a functions but only the name of the mutator and the arguments.

order defined by causality. All replicas run Algorithm 1 which ensures that they execute operations according to some total order that extends the partial order and respects the operations' state validators.

Algorithm 1 starts when a replica receives a *mutate* message. The algorithm consists of two parts. First, it adds the *mutate* message to the operation history, sorts the history according to the $>>$ total order, and then, generates all serializations that respect causality (see Lines 1 and 2). We say that $m_1 = (o_1, p_1, a_1, c_1, id_1) >> m_2 = (o_2, p_2, a_2, c_2, id_2)$ iff $c_1 \succ c_2 \vee (c_1 \parallel c_2 \wedge id_1 > id_2)$. The generated serializations are all the permutations of h' that respect the causal relations between the operations. Since replicas use the same deterministic algorithm to compute causal permutations and start from the same sorted operation history, the resulting algorithm is deterministic. This ensures that all replicas generate the same sequence of permutations.

Second, the algorithm searches for the first *valid* permutation. In other words, for each operation within such a permutation the algorithm checks that the preconditions (Lines 9 to 12) and postconditions (Lines 13 to 15) hold. Remember that postconditions are checked only after all concurrent operations executed since they happened independently (e.g. during a network partition) and may thus conflict. For this reason, Line 8 computes the transitive closure of concurrent operations[5] for every operation in the linear extension. This transitive closure corresponds to the set of operations that could affect each other and thus may introduce conflicts.

Even though concurrency is not transitive, we consider operations that are not directly concurrent. To understand why this is important, consider a replica r_1 that executes operation o_1 followed by o_2 ($o_1 \prec o_2$) while concurrently replica r_2 executes operation o_3 ($o_3 \parallel o_1 \wedge o_3 \parallel o_2$). Since o_3 may affect both o_1 and o_2, the algorithm needs to take into account all three operations. This corresponds to the transitive closure $\{o_1, o_2, o_3\}$.

Finally, the algorithm returns the replica's updated state as soon as a valid execution is found, otherwise, it throws an exception.

3. Committing Replicas A commit operation clears the replica's operation history h, increments the replica's version and updates the initial state s_0 with the replica's current state s_i. Every commit has

[5]The transitive closure of a mutate message m with respect to an operation history h is denoted $TC(m, h)$ and is the set of all operations that are directly or transitively concurrent with m. A formal definition is provided in Appendix B.

Algorithm 1 Handling *mutate* messages.

arguments: A *mutate* message m = (o, p, a, c, id), a replica = (v_i, s_0, s_i, h, id_c)

```
1: h' = h ∪ {m}
2: for ops ∈ CausalPermutations(sort_>>(h')) do
3:     len = |ops|
4:     s'_i = s_0
5:     pre = 0
6:     post = 0
7:     for m ∈ ops do
8:         concurrentClosure = TransitiveClosure(m, h') ∪ {m}
9:         for (o, p, a, c, id) ∈ concurrentClosure do
10:            if p(s'_i) then
11:                pre += 1
12:                s'_i = o(s'_i)
13:        for (o, p, a, c, id) ∈ concurrentClosure do
14:            if a(s'_i) then
15:                post += 1
16:        ops = ops \ concurrentClosure
17:    if pre == len ∧ post == len then
18:        return (v_i, s_0, s'_i, h', id_c)
19: throw NoSolutionException
```

a unique identifier in order to deterministically break ties in case of concurrent commits. By periodically committing state, replicas can avoid unbounded growth of their operation history, but operations concurrent with the commit will be discarded.

When a replica is committed a *commit* message is broadcast to all replicas (including the committed one). This message is a quadruple $(s_i, v_i, clock, id)$ containing the committed state, the replica's version number, the current logical clock time, and a unique id.

Algorithm 2 Handling *commit* messages.

arguments: A *commit* message = $(s_c, v_c, clock, id)$, a replica = (v_i, s_0, s_i, h, id_c)

```
1: if v_c = v_i then
2:     return (v_i + 1, s_c, s_c, ∅, id)
3: if v_c = v_i − 1 ∧ id < id_c then
4:     return (v_i, s_c, s_c, ∅, id)
```

To ensure that replicas converge in the face of concurrent commits we design commit operations to commute. As a result, commit does not compromise availability. Algorithm 2 dictates how replicas handle *commit* messages. The algorithm distinguishes between two cases. The first case considers committing the current state (Line 1). The replica's version is then incremented, its initial and current state are set to the committed state, the operation history is cleared, and the id of the last performed commit is updated. The second case considers committing the previous state (Line 3). This means that the commit operation applies to the previous version v_{i-1}. Thus, the received commit message is concurrent with the last performed commit (i.e. the one that caused the replica to update its version from v_{i-1} to v_i). To ensure convergence, the commit with the smallest ID wins. This ensures that replicas deterministically break ties such that concurrent commits commute. Note that the algorithm does not need to tackle the case of committing an older state since it cannot happen under the assumption of causal order broadcasting.

3.2.2.2 Time Complexity

We now elaborate on the worst-case time complexity of the SECRO replication protocol. Consider a replica with a history of size n. In the worst case, all operations in the history are concurrent. As a result, there are no causal relations between the operations and there are $n!$ permutations to consider. For every permutation, the SECRO protocol tests it validity. In the worst case, only the last permutation is valid. Testing for validity implies that for every operation its precondition is tested, the operation is executed, and the postcondition is tested. Since the preconditions, operations, and postconditions are data-type specific there time complexity may vary, hence, we introduce additional variables p, o, and a that denote the worst-case time complexity of the preconditions, operations, and postconditions respectively. The total worst-case time complexity thus becomes: $O(n! * (p + o + a))$.

For example, if the preconditions and postconditions are constant-time operations (i.e. $O(1)$) but the operations' time complexity is linear to the length of the history, the time complexity becomes $O(n! * n)$.

3.2.2.3 Consistency Guarantees

We previously explained the SECRO replication protocol in detail. We now elaborate on the consistency guarantees provided by SECROs.

The replication protocol behind SECROs ensures that replicas that received the same operations find and execute the same serialization and thus converge to the same state. For this reason, SECROs guarantee SEC.

Although SECROs were originally designed to guarantee SEC, they provide the stronger guarantee of causal$^+$ consistency which is the combination of convergence and the four session guarantees (cf. Section 2.1.2).

If clients are sticky, SECROs guarantee the four session guarantees: Read Your Writes, Monotonic Reads, Writes Follow Reads, and Monotonic Writes. Read Your Writes and Monotonic Reads follow from the fact that clients are sticky and the protocol's operation history grows monotonically; i.e. once an operation is observed it will be in the history forever and thus will be observed by all future reads.

Writes Follow Reads and Monotonic Writes follow from the fact that replicas propagate operations (i.e. writes) in causal order and execute a causal serialization of the operations. If a write w is issued after a set of writes W were observed (by a read), then all writes $w' \in W$ causally precede w and thus all replicas will execute a serialization of the operations that respects these causal relations.

3.3 Performance Evaluation

We now conduct a performance evaluation of our approach. To this end, we compare SECROs to JSON CRDTs [KB17], the closest related work for designing custom RDTs without manual conflict resolution. JSON CRDTs enable programmers to build custom CRDTs by nesting lists and maps in a JSON-like data format. Our experiments quantify the memory usage and latency of operations for collaborative text editing applications that use these approaches. To ensure a fair comparison, we implemented both approaches in JavaScript. The JSON CRDT implementation of the text editor represents text documents as a linked list of characters. The SECRO implementation features two variants: one that uses a list of characters and one that uses a balanced tree of characters.

3.3.1 Methodology

The experiments reported in this chapter were performed on a cluster consisting of 10 worker nodes which are interconnected through a 10 Gbit twinax connection. Each worker node has an Intel Xeon E3-1240 processor at 3.50 GHz and 32 GB of RAM. Depending on the experiment, the benchmark is either run on a single worker node or on all ten nodes. We specify this for each benchmark.

To get statistically sound results we repeat each benchmark at least 30 times. Each benchmark starts with a number of warmup iterations to account for virtual machine warmup. Furthermore, we disable NodeJS' just-in-time compiler optimisations to obtain more stable execution times.

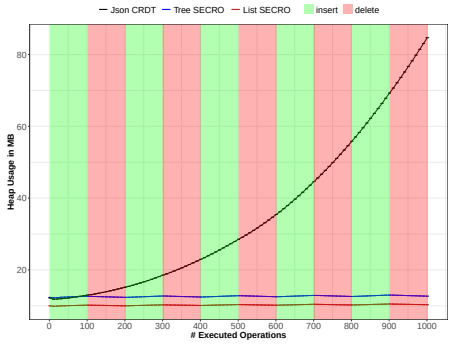
We perform statistical analysis over our measurements as follows. First, we discard samples that are affected by garbage collection (only for the latency benchmark). Then, for each measurement including at least 30 samples we compute the average value and the corresponding 95% confidence interval.

3.3.2 Memory Consumption

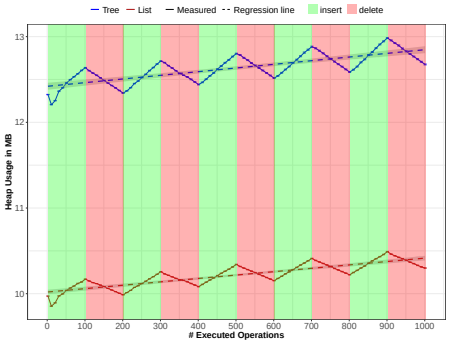
To compare the memory usage of the SECRO and JSON CRDT text editors, we perform an experiment in which 1000 operations are executed on a single replica of the text editors. We continuously alternate between 100 character insertions followed by the deletion of those 100 characters to showcase the effect of deletions on memory usage. We force garbage collection after each operation[6] and measure the heap usage. The resulting measurements are shown in Fig. 3.1. Green and red columns indicate character insertions and deletions respectively.

Figure 3.1a confirms our expectation that the SECRO implementations are more memory efficient than the JSON CRDT implementation. The memory usage of the JSON CRDT text editor grows unbounded because it cannot delete characters but merely marks them as deleted. This is a common CRDT-technique that is known as tombstones. Conversely, SECROs support true deletions by reorganising concurrent operations in a non-conflicting order. Hence, all 100 inserted characters are deleted by the following 100 deletions. This results in lower memory usage.

[6]Forcing garbage collection is needed to get the real-time memory usage. Otherwise, the memory usage keeps growing until garbage collection is triggered.



(a) Comparison between the memory usage of the SECRO and JSON CRDT text editors.



(b) Comparison between the list and tree implementations of the SECRO text editor.

Figure 3.1: Memory usage of the collaborative text editors. Error bars represent the 95% confidence interval for the average taken from 30 samples. The experiments are performed on a single worker node of the cluster.

Figure 3.1b compares the memory usage of the list and tree-based implementations using SECROs. We conclude that the tree-based implementation consumes more memory than the list implementation. The reason is that nodes of a tree maintain pointers to their children, whereas nodes of a singly linked list only maintain a single pointer to the next node. Interestingly, we observe a staircase pattern. This pattern indicates that memory usage grows when characters are inserted (green columns) and shrinks when characters are deleted (red columns). Overall, memory usage increases linearly with the number of executed operations, even though we delete the inserted characters and commit the replica after each operation. Hence, SECROs cause a small memory overhead for each executed operation. This linear increase is shown by the dashed regression lines.

3.3.3 Latency of Operations

We now measure the latency of character insertions at the end of a text document. Although this is not a realistic editing pattern, it showcases the worst-case performance of SECROs. From Fig. 3.2a we notice that both the list-based and tree-based SECRO implementations exhibit quadratic performance, whereas the JSON CRDT list-based implementation exhibits linear performance. The reason for this is that reordering the SE-

CRO's history (cf. Algorithm 1) induces at least a linear overhead on top of the operations themselves. Since inserting in a linked list is also a linear operation, the overall performance of the text editor's insert operation becomes quadratic.

Figure 3.2a also shows that the SECRO implementation that uses a linked list is faster than its tree-based counterpart. To determine the cause of this counterintuitive observation, we measure different parts that make up the total execution time:

Execution time of operations. Time spent in append operations.

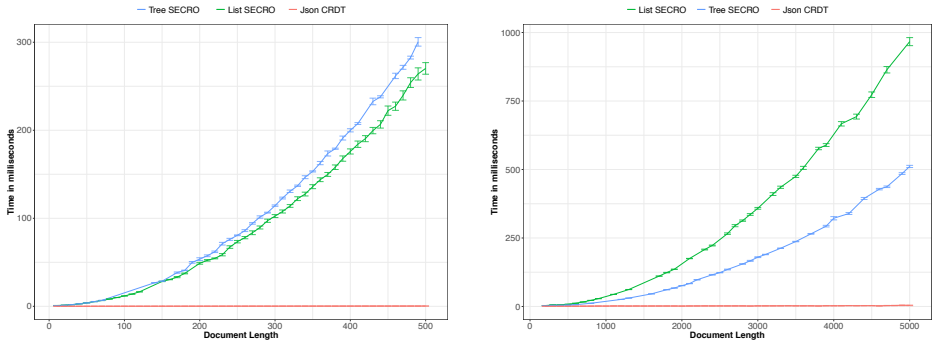
Execution time of preconditions. Time spent in preconditions.

Execution time of postconditions. Time spent in postconditions.

Copy time. Since JavaScript objects are mutable, replicas take a copy of their state before tentatively executing a serialization of the operations. The time spent copying the state is the copy time.

Figures 3.3a and 3.3b depict the detailed execution time for the list and tree implementations respectively. The results show that the total execution time is dominated by the copy time. We observe that the tree implementation spends more time copying the document than the list implementation. The reason being that copying a tree entails a higher overhead than copying a linked list as more pointers need to be copied. Furthermore, the tree implementation spends considerably less time executing operations, preconditions and postconditions, than the list implementation. This results from the fact that the balanced tree provides logarithmic time operations.

Unfortunately, the time overhead incurred by copying the document kills the speedup we gain from organising the document as a tree. This is because each insertion inserts only a single character but requires the entire document to be copied. To validate this hypothesis, we re-execute the benchmark shown in Fig. 3.2a but insert 100 characters per operation. Figure 3.2b shows the resulting latencies. As expected, the tree implementation now outperforms the list implementation. This means that the speedup obtained from 100 logarithmic insertions exceeds the overhead of copying the tree. In practice, this means that single character manipulations are too fine-grained. Manipulating entire words, sentences or even paragraphs is more beneficial for performance.



(a) Latency of an operation that appends a single character to a document.

(b) Latency of an operation that appends 100 characters to a document.

Figure 3.2: Latency of character insertions in the collaborative text editors. Replicas are never committed. Error bars represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection are discarded.

From the latency benchmarks discussed in this section, we conclude that 1) the SECRO algorithm introduces at least a linear overhead because the history needs to be reordered, and 2) copying the state (i.e. the entire document) induces a considerable additional overhead. To address the reordering overhead, replicas can periodically commit their state such that the history remains small. The effect of commit on the latency of insert operations is analyzed in Section 3.3.4. Regarding the copying overhead, we believe that it is not inherent to SECROs, but rather a consequence of its implementation on top of a mutable language such as JavaScript. In an immutable language, there would be no need to copy the state.

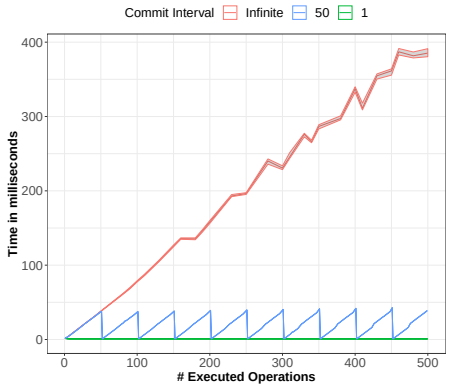
3.3.4 Effect of Commit on the Latency of Operations

We now conduct two benchmarks that analyze the effect of commit on the latency of operations. The first benchmark measures the latency of constant time operations on SECROs in order to analyze how commit reduces the reordering overhead. The second benchmark illustrates the effect of commit on the latency of insertions in the collaborative text editor for different commit intervals.

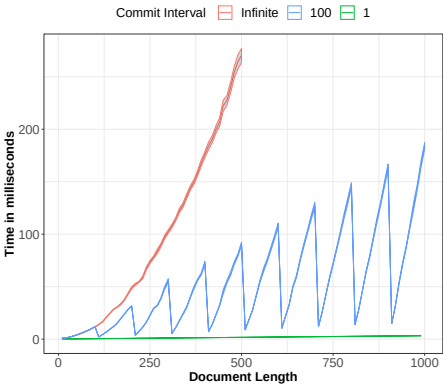
(a) List implementation

(b) Tree implementation.

Figure 3.3: Detailed latency to append characters to the SECRO text editor. The replica is never committed. The plotted latency is the average taken from a minimum of 30 samples. Samples affected by garbage collection are discarded.



(a) Execution time of a constant time operation in function of the number of executed operations.



(b) Time to append a character to the text document using the list implementation of the SECRO text editor.

Figure 3.4: Execution time of SECROs for different commit intervals, performed on a single worker node of the cluster. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.

Benchmark 1. To quantify the performance overhead of SECROs we measure the latency of 500 *constant* time operations, for different commit intervals. We cannot use the text editor for this purpose because its operations are not constant. Moreover, the constant time operation needs to execute long enough to allow for accurate latency measurements. Therefore, we designed the constant time operation to compute 10 000 tangents. We did not associate any pre- or postcondition to this operation. As a result, any serialization of the operations is valid and this experiment reflects the best-case performance of SECROs.

Figure 3.4a depicts the latency of the aforementioned constant time operation. If we do not commit the replica (red curve), the operation's latency increases linearly with the number of operations. Hence, SECROs induce a linear overhead. This results from the fact that the replica's operation history grows with every operation. Each operation requires the replica to reorganise the history. To this end, the replica generates linear extensions of the history until a valid ordering of the operations is found (see Algorithm 1 in Section 3.2.2.1). Since we defined no preconditions or postconditions, every order is valid. The replica thus generates exactly

one linear extension and validates it. To validate the ordering, the replica executes each operation. Therefore, the operation's latency is linear to the size of the operation history.

As mentioned previously, commit implies a trade-off between concurrency and performance. Small commit intervals lead to better performance but less concurrency, whereas large commit intervals support more concurrent operations at the cost of performance. Figure 3.4a illustrates this trade-off. For a commit interval of 50 (blue curve), we observe a sawtooth pattern. The operation's latency increases until the replica is committed, whereafter it falls back to its initial latency. This is because *commit* clears the operation history. When choosing a commit interval of 1 (green curve), the replica is committed after every operation. Hence, the history contains a single operation and does not need to be reorganised. This results in a constant latency.

Benchmark 2. We now analyse the latency of insert operations on the collaborative text editor. Figure 3.4b shows the time it takes to append a character to a text document in function of the document's length, for various commit intervals. If we do not commit the replica (red curve), append exhibits quadratic latency. This is because the SECRO induces a linear overhead and append is a linear operation. For a commit interval of 100 (blue curve) we again observe a sawtooth pattern. In contrast to Fig. 3.4a the peaks increase linearly with the size of the document, since append is a linear operation. If we choose a commit interval of 1 (green curve) we effectively get a latencies that are linear to the size of the document. This results from the fact that we do not need to reorganise the replica's history. Hence, we execute a single append operation.

Conclusion. Based on the results above, we draw two conclusions. First, SECROs induce a linear overhead on the latency of operations. Second, commit is a pragmatic solution to keep the performance of SECROs within reasonable bounds. Determining the optimal rate at which replicas should be committed depends on the application at hand. If operations are slow the history is best kept small through regular commits, whereas, if operations are fast replicas can commit less frequently to accommodate more concurrency.

3.4 Notes on Related Work

We now review influential ideas related to our approach. Bayou [Ter+95] was the first system to use application-level semantics for conflict detection and resolution by means of user-provided dependency checks and merge procedures. SECROs, however, do not require manual conflict detection or resolution. Instead, programmers specify application-specific invariants and the underlying replication protocol serializes operations accordingly.

IceCube [Ker+01] is a general-purpose engine that reconciles diverged replicas using semantic information provided by programmers in the form of constraints. Constraints can be static or dynamic and are defined over serializations of concurrent operations. Expressing appropriate constraints is nontrivial as it requires reasoning about the order of operations. In contrast, SECROs detect conflicts using data-oriented invariants which are easier to express. Like SECROs, IceCube's dynamic constraints may lead to rollbacks which represent a performance hit.

As explained in Section 2.2.1.1, traditional RDTs rely on commutative operations or user-defined merge procedures to solve conflicts (e.g. cloud types, CRDTs, MRDTs, etc.). To free programmers from manual conflict resolution, some approaches [MV15; KB17; WMM20] let programmers build custom RDTs by composing existing ones. Lasp [MV15] lets programmers define new CRDTs by applying functional transformations over existing CRDTs. It is not clear if all CRDTs can be obtained this way. JSON CRDTs [KB17] let programmers build new CRDTs using a JSON-like data format that allows arbitrary nestings of lists and maps. Unfortunately, the lists and maps expose only a limited API (only insertions, deletions, and assignments). [WMM20] leverage semidirect products to combine CRDTs. However, this requires expertise from the programmers because they need to define arbitration orders for concurrent operations, transform concurrent operations, etc. While the aforementioned approaches start from existing CRDTs and compose them in a conflict-free way, the SECRO approach is different in that it starts from a sequential data type and totally orders operations instead of designing them specially to be conflict-free.

3.5 Conclusion

In this chapter we proposed SECROs, a novel approach to design RDTs by extending sequential data types with state validators. The replication protocol behind SECROs totally orders operations such that causality is respected and application-specific invariants defined by the operations' state validators are maintained. Since the protocol is deterministic, replicas that observed the same operations compute the same serialization and thus converge without coordination between the replicas.

SECROs are a first step to broaden the scope of RDTs to data types with non-commutative operations. This adheres to our first principle of replicating existing designs instead of designing for replication. Furthermore, the underlying replication protocol guarantees convergence out-of-the-box and is fully integrated in CScript, which respectively adheres to our second and third principles outlined in Section 1.2.

Our evaluation showcases the perpetual tension between performance and expressiveness. SECROs improve the expressiveness of RDTs by allowing arbitrary data types to be replicated and thus lifting the traditional commutativity requirement, but need to give in on performance.

In the next chapter, we describe an improved approach to build RDTs from sequential data types. The approach still leverages application-specific semantics described by programmers, but analyzes them beforehand to reduce the algorithm's runtime overhead.

Chapter 4

Efficient Replicated Data Types from Sequential Code

We previously proposed SECROs, a programming abstraction that enables programmers to build RDTs by extending sequential data types with invariants. Although SECROs guarantee SEC and uphold application invariants, they do not scale well because replicas have to search for a valid serialization of the operations at runtime. To keep the performance acceptable, replicas need to periodically commit their state but finding a good commit rate is challenging.

We now explore an alternative approach that keeps the essence of SECROs, namely, application-specific invariants, but statically analyzes the data type and its invariants to make informed decisions at runtime that ensure good performance. We revisit the replication protocol to build a serialization guided by the analysis results instead of searching for a serialization among all permutations.

This chapter introduces the Explicitly Consistent Replicated Object (ECRO) programming model, which lets programmers define RDTs by extending sequential data types with a *distributed specification* describing the desired semantics through application-specific invariants, akin to SECRO's state validators. The specification is statically analyzed to derive information about conflicting operations. At runtime, replicas use this information to build a Directed Acyclic Graph (DAG) of operations that represents a partial ordering of the operations. The DAG is carefully constructed such that all topological orders of the DAG (i.e. all serializa-

tions of the partial order) converge to equivalent states and maintain the application's invariants.

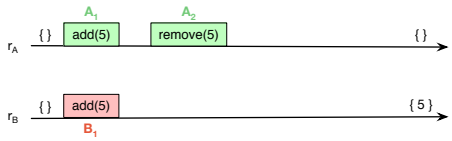
We evaluate the applicability of our approach in two ways. First, in Section 4.5, we conduct a qualitative evaluation to assess if ECROs simplify the development of RDTs compared to state-of-the-art approaches. Second, in Section 4.6, we conduct a performance evaluation to assess the feasibility of the static analysis phase and measure the latency and scalability of the replication protocol.

4.1 The Need for Static Analysis

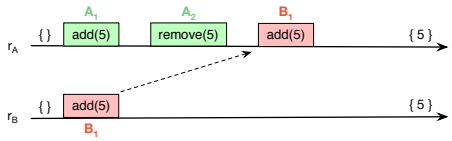
As mentioned before, SECROs provide a simple way for programmers to replicate existing data types, but reordering operations at runtime is inefficient and limits the applicability of the approach. This inefficiency comes from the fact that, at *runtime*, replicas potentially have to generate and test all causal permutations of the operations. This is needed because replicas have no information about the operations, hence, all serializations have to be tested one by one until a valid one is found. Generating all serializations is subject to the problem of combinatorial explosion.

To make this approach practical, we revisit SECRO's replication protocol such that replicas independently build a serialization of the operations instead of searching for a serialization in a huge search space. Since replicas may receive (concurrent) operations in different orders they may construct different serializations, however, those serializations should be equivalent (i.e. lead to equivalent states) *and* maintain all invariants.

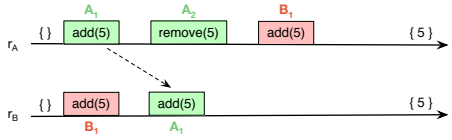
Section 3.1 described two possibilities to achieve convergence without coordination. The first possibility consists in solving the conflicts that occur from concurrent operations, while the second consists in totally ordering concurrent operations across replicas. The replication protocol behind ECROs adheres to the second option but only totally orders concurrent operations that are not commutative (whereas SECROs totally order all concurrent operations). Since all replicas execute non-commutative concurrent operations in the same order, they are guaranteed to converge to the same state. Thus, replicas may execute operations according to different serializations, but those serializations are equivalent. Equivalent serializations exhibit a total order for non-commutative operations but commutative operations may be executed in any order. This requires



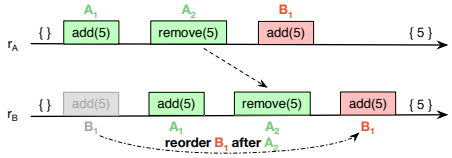
(a) Concurrent operations issued by r_A and r_B.



(b) Operation B_1 is propagated to replica r_A.



(c) Operation A_1 is propagated to replica r_B.



(d) Operation A_2 is propagated to replica r_B which reorders operation B_1 to ensure add-wins semantics.

Figure 4.1: Reordering operations in a replicated Add-Wins Set ECRO.

replicas to execute non-commutative operations in causal order and impose some artificial but deterministic order on concurrent operations if they do not commute.

Additionally, operation serializations must maintain the application's invariants. Oftentimes, concurrent operations break invariants because they are executed in a conflicting order (e.g. placing a bid on an auction after it was closed concurrently). Such conflicts can be solved by reordering the operations. If no safe ordering exists, ECROs coordinate the problematic operations to ensure that they do not run concurrently. In contrast, SECROs would fail to find a valid serialization for those problematic operations and thus raise a runtime exception.

To build an improved replication protocol for ECROs, that efficiently serializes operations and maintains application-specific invariants, the protocol needs additional information about the operations. Concretely, the protocol needs to know which operations commute, which operations break invariants, under which conditions operations commute or conflict, etc. A key insight of our work is that this information can be *automatically* derived by statically analyzing the data type and its invariants. At runtime, replicas use the information derived by the static analysis to serialize operations according to the application's invariants.

Consider for example a replicated add-wins set and how information about this data type can be used to serialize operations. Clearly, `add(x)` and `remove(y)` operations on a set do not commute if $x = y$ and thus replicas need to execute these operations in the same order to ensure convergence. Moreover, add-wins semantics requires adds to win over concurrent deletes of the same element, which can be achieved by executing `add(x)` operations after concurrent `delete(y)` operations when $x = y$. This information can be statically derived from the set's implementation and its add-win invariant, and be used at runtime to serialize the operations. Figure 4.1 depicts two replicas of an add-wins set that initially is empty $\{\}$. In Fig. 4.1a, replica r_A adds 5 to the set (operation A_1) and later removes it from the set (operation A_2). Concurrently, replica r_B also adds 5 to the set (operation B_1). Operation B_1 is concurrent with operations A_1 and A_2. Then in Fig. 4.1b, operation B_1 is propagated to replica r_A which adds 5 to the set. Similarly, in Fig. 4.1c, operation A_1 is propagated to replica r_B which adds 5 to the set. This operation leaves the state unchanged since 5 was already in the set. When operation A_2 is propagated to replica r_B (cf. Fig. 4.1d), r_B cannot immediately apply A_2 since removing 5 from the set would violate the desired add-wins semantics. Instead, based on the information from the analysis, r_B could *reorder* the concurrent `add(5)` (operation B_1) such that it is executed after `remove(5)` (operation A_2) and thus guarantees add-wins semantics. This is exactly what the ECRO replication protocol does.

Note how, at runtime, replica r_B knew from the static analysis that it had to reorder B_1 after A_2 in order to ensure add-wins semantics, whereas, SECROs would have to search through all causal serializations ($B_1; A_1; A_2$, $A_1; B_1; A_2$, and $A_1; A_2; B_1$) in order to find the valid serialization ($A_1; A_2; B_1$).

4.2 Building Geo-Distributed Applications, the ECRO Way

We now present ECROs, our improved approach to programming RDTs from sequential data types. The approach is implemented in Scala. We provide an overview of the approach, demonstrate its use by implementing two set RDTs and an auction system RDT, and discuss how it differs from state-of-the-art approaches.

4.2.1 Overview

Figure 4.2 depicts a high-level overview of the ECRO approach. Programmers build RDTs by extending *sequential* data types with a *distributed specification* defining the data type's semantics through invariants over replicated state. Together, the sequential data type and its distributed specification form an ECRO. The state of an ECRO is replicated across machines, each of which is said to hold a *replica*. Programmers interact with ECROs by calling *methods* on a replica. Method calls are propagated between replicas using a broadcasting mechanism that guarantees eventual and causal delivery. Under these assumptions, ECRO's replication protocol guarantees safety and strong convergence. *Safety* is the property that the replicated state respects the application's invariants. *Strong convergence* [Sha+11b] is the property that correct replicas that processed the same calls (possibly in a different order) are in equivalent states.

Key to guaranteeing the above properties is our analysis tool, called Ordana, that statically analyzes distributed specifications to detect conflicting operations and find solutions beforehand. At runtime, the ECRO replicas use the information inferred by Ordana to serialize calls efficiently (i.e. with minimal coordination) while upholding safety and strong convergence. To this end, every replica keeps a tentative serialization of the calls which may be affected by concurrent calls. When a prefix of calls is causally stable at a replica, the replica knows that every other replica has observed those operations. Thus, no more concurrent operations can arrive and the prefix is *committed*. Similarly to SECROs, commit updates

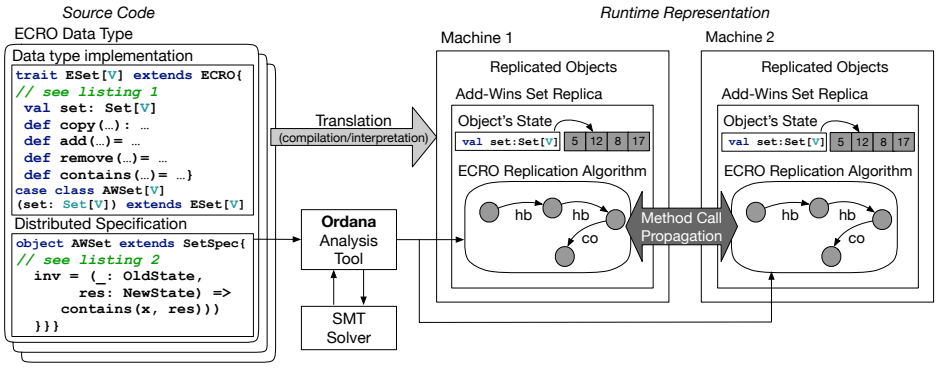


Figure 4.2: Overview of ECROs.

the replica's internal state to the state that results from applying the prefix of operations, after which those operations are garbage collected.

4.2.2 Building Replicated Sets

We now illustrate the ECRO approach by implementing a set RDT which lies at the core of many geo-distributed applications. A set RDT differs from a sequential set as multiple users may add and remove the same element concurrently. When these updates have been received by all replicas, the element must be present in all replicas (add-wins semantics) or absent from all replicas (remove-wins semantics). Hence, a sequential data type may have several replicated counterparts, each exhibiting different semantics when facing concurrent operations.

Like SECROs, ECROs let programmers turn any sequential data type into an RDT by defining the desired semantics in the data type's distributed specification. The specification describes the operations that modify the internal state by means of three components: a precondition, a postcondition, and an invariant. Each component (detailed later in Section 4.3.1) is a function that is parametrized by the data type's state(s) and returns a first-order logic predicate used by Ordana to analyze the operations.

Listing 4.2 shows part of the implementation of the add-wins `AWSet` and remove-wins `RWSet` ECROs in Scala. The syntax is simplified for presentation purposes. The complete implementation of the sets is provided in Appendix C.1. Both sets extend the `ESet` trait[1] (shown in Listing 4.1) which wraps Scala's built-in immutable set and offers the typical set op-

[1]Scala traits are similar to Java interfaces with default implementations. Moreover, classes can extend multiple traits in Scala.

Listing 4.1: Sequential set implementation.

```
1 trait ESet[V] extends ECRO {
2   val set: Set[V]
3   def copy(set: Set[V]): ESet[V]
4   def add(x: V) = copy(set + x)
5   def remove(x: V) = copy(set - x)
6   def contains(x: V) = set.contains(x)
7 }
```

Listing 4.2: Add-Wins and Remove-Wins Set ECROs.

```
1  case class AWSet[V](set: Set[V]) extends ESet[V]
2  case class RWSet[V](set: Set[V]) extends ESet[V]
3  object AWSet {
4    val x: Identifier = ..; val y: Identifier = .. // Appendix C.1
5    val contains: Relation = ... // see Appendix C.1
6    postcondition of add {
7      (old: OldState, res: NewState) =>
8        contains(x, res) /\
9        // old -> res copies relations from `old` to `res`
10       contains.copyExcept(old -> res, elem === x)
11   }
12   postcondition of remove {
13     (old: OldState, res: NewState) =>
14       not (contains(x, res)) /\
15       contains.copyExcept(old -> res, elem === x)
16    }
17   invariant on add {
18     (_: OldState, res: NewState) => contains(x, res))
19 } }
20 object RWSet {
21   // contains & postconditions same as AWSet
22   invariant on remove {
23     (_: OldState, res: NewState) =>
24       not (contains(x, res))))
25 } }
```

erations. The sets' distributed specifications use an embedded Scala DSL that we built for programming with first-order logic (cf. Appendix C).

By convention, the specification is defined in the class' companion object[2]. The postconditions for `add(x)` and `remove(x)` state that after adding/removing `x`, the element is present/absent from the resulting state `res`, and that all other elements are unchanged (lines 11 and 16). The `AWSet` contains an invariant on the `add(x)` operation to force element `x` to be present in the resulting state `res` and thus guarantees add-wins semantics (lines 17-19). Similarly, the `RWSet` contains an invariant on the `remove(x)` operation to force element `x` to be absent from the resulting state and thus guarantees remove-wins semantics (lines 22-25).

This example demonstrates the flexibility of ECROs: to switch between add-wins and remove-wins semantics, *only* the invariant defined in the distributed specification was changed (one line of code). In contrast, traditional RDT solutions like CRDTs require two different data type im-

[2]A class' companion object is an object that is defined in the same file and has the same name as the class. It can be used to group static variables and methods.

plementations each engineered to yield the desired semantics (as we will further discuss in Section 4.5.2.1). Other works, like RedBlue consistency, do not require changes to the data type but would synchronize all `add` and `remove` operations due to the possibility of an add-remove conflict. In contrast, Ordana finds a solution to add-remove conflicts that does not require coordination (as shown in Fig. 4.1).

4.2.3 Building a Geo-Distributed Auction System

We now show how to use the ECRO approach to build a custom RDT for which no ready-made RDT design exists. To this end, we develop a geo-distributed auction system akin to the RUBiS system [EJ09], where users first register, whereafter they can open auctions, bid on auctions, and close auctions. Moreover, users can sell a number of items for a fixed price and buy such items. RUBiS requires usernames to be unique, bids to be linked to one existing user, items to have a non-negative stock, etc.

In an attempt to develop RUBiS, one may compose a set RDT of users with a map RDT from auction IDs to auctions where an auction consists of a set RDT of bids and an enable-once flag RDT indicating whether the auction is open or closed. However, traditional RDTs (CRDTs, Cloud Types, etc.) require programmers to manually uphold application invariants. For example, each bid must be linked to an existing user. This is a common invariant, known as *referential integrity*. However, if a user places a bid on an auction and concurrently the user is deleted, some replica may first delete the user and then place the bid, which violates referential integrity because the user no longer exists. It is not clear how to ensure this invariant with traditional RDTs because it requires an atomic update across the set RDT of users and the auction's set RDT of bids, but this is not supported by traditional RDTs.

We now discuss how to implement a replicated RUBiS system starting from a sequential implementation, using ECROs. We focus the discussion on the auction operations. The complete implementation of the auction operations is provided in Appendix C.2. The sequential implementation keeps a set of users and a map from auction IDs to auctions containing a set of bids, a status (open or closed), and a winner. When a user bids on an auction, it is added to the set of bids. Bids may only be placed on open auctions. Listing 4.3 shows the distributed specification of the `placeBid` and `closeAuction` methods. The precondition of `placeBid`

Listing 4.3: Distributed specification of an auction system.

```
1  case class Rubis(users: Set[User], auctions: Map[AID, Auction])
       extends ECRO {
2    def placeBid(auctionId:AID,userId:User,price:Int): Rubis = ...
3    def closeAuction(auctionId: AID): Rubis = ...
4  }
5  object Rubis {
6    // see Appendix C.2 for the definition of the relations
7    val auction: Relation = ...
8    val user: Relation = ...; val bid: Relation = ...
9    precondition of placeBid {
10     (state: CurrentState) =>
11       auction(auctionId, Open, state) /\
12       user(userId, state) /\ (price >> 0)
13   }
14   postcondition of placeBid {
15     (old: OldState, res: NewState) =>
16       old + bid(auctionId, userId, price, res) /\
17       bid.copy(old -> res)
18   }
19   postcondition of closeAuction {
20     (old: OldState, res: NewState) =>
21       old + auction(auctionId, Closed, res) /\
22       not (auction(auctionId, Open, res)) /\
23       auction.copyExcept(old -> res, id === auctionId)
24   } }
```

(line 9 to 13) requires the auction to be open, the user to exist, and the bid to be bigger than zero. The postcondition of `placeBid` (line 14 to 18) extends the state with the new bid and copies all bids from the old state to the new state. The postcondition of `closeAuction` (line 19 to 24) puts the auction's status on closed, states that it can no longer be open, and copies all other auctions from the old state to the new one.

By statically analyzing the distributed specification, Ordana detects operations that violate application invariants. For example, concurrent `placeBid` and `closeAuction` calls may lead to a bid being placed on a closed auction if `closeAuction` is executed before `placeBid`, which violates the precondition of `placeBid`. ECROs solve this conflict by imposing an order on the operations (cf. Section 4.5.1). In contrast, invariant-preserving RDTs such as Redblue, PoR, Hamsaz, etc. (cf. Section 2.2.1.2) coordinate all calls to these operations because of this potential conflict. Clearly, they are too conservative since only concurrent calls to `placeBid` and `closeAuction` that modify the *same* auction are conflicting, yet no calls to `placeBid` and `closeAuction` are allowed to run concurrently.

4.2.4 Coping with Different Classes of Conflicts

ECROs start from the observation that many conflicts are due to a "bad" ordering of *concurrent* operations, and solve those conflicts by reordering the operations, rather than coordinating them. We briefly categorize four types of conflicts and explain how ECROs cope with them.

The first category of conflicts arises when replicas concurrently execute non-commutative operations, which are then exchanged and applied in different orders at different replicas, yielding diverged states. To ensure state convergence, ECROs deterministically order concurrent non-commutative operations at all replicas. In contrast, existing approaches [Li+12; LPR18; SKJ15; Kak+18; HL19; LHL20] coordinate these operations unnecessarily.

The second category of conflicts arises when some operation leads to a state transition, which renders concurrent operations unavailable in the new state (e.g. `closeAuction` closes an auction and may render concurrent `placeBid` operations unavailable). ECROs solve those conflicts by safely reordering unavailable operations before transitioning to the new state (e.g. reorder `placeBid` before `closeAuction`), whereas existing approaches coordinate those operations.

The third category of conflicts involves numeric invariants. For example, a banking application may implement the account balance as a non-negative counter (aka a bounded counter). However, concurrent withdrawals may overdraw the account, and reordering the withdrawals does not solve this problem. For such conflicts, ECROs coordinate the problematic operations, and so does related work.

The last category of conflicts is due to replicas executing mutually exclusive operations concurrently. For example, if usernames must be unique, then the `registerUser(username)` operation must take a lock on `username` to avoid that someone else registers the same username concurrently. These conflicts break invariants and thus require coordination between the operations. Like most approaches, ECROs coordinate mutually exclusive operations. Some applications may however allow temporary invariant violations to avoid coordination and instead repair the invariant after the facts [GPS16].

4.3 Deriving Safe Serializations from Distributed Specifications

We previously explained that ECRO replicas compute a serialization of the method calls that respects application invariants and guarantees strong convergence. Key to efficiently computing such serializations is a static analysis phase that answers four questions:

1. Which sequential method calls commute?
2. Which concurrent method calls commute?
3. When are concurrent method calls safe or unsafe?
4. If two concurrent method calls are unsafe, does a safe ordering of the calls exist? If yes, which order?

To answer these questions, we developed Ordana: a static analysis tool that implements three analyses on distributed specifications. The first is a dependency analysis based on [HL19] that detects dependencies between sequential method calls[3]. The second is a commutativity analysis based on [Bal+15] that detects commutativity of concurrent calls. The dependency analysis and commutativity analysis are combined to detect commutativity of sequential calls. The third is a novel safety analysis that detects conflicts and finds solutions by reordering calls locally.

Before detailing the analyses, we define the components of an ECRO's distributed specification. To this end, we use the RUBiS auction system (cf. Section 4.2.3) as running example.

4.3.1 The ECRO Distributed Specification

Every ECRO data type consists of two parts: the data type's implementation and a distributed specification. The implementation encapsulates replicated state (e.g. class fields) and exposes a number of methods, some

[3]Two method calls are sequential if one happened before the other (i.e. one was observed and only then was the other generated) [Lam78]. If neither call happened before the other, the calls are concurrent.

of which mutate the replicated state[4]. The distributed specification describes three aspects of every call to a mutating method m:

Precondition $\texttt{pre}(m(\overline{a}), \sigma)$ Predicate that checks if m can be called with arguments[5] $\overline{a}$ on state σ. The precondition must be true before applying the call at *any* replica.

Postcondition $\texttt{post}(m(\overline{a}), \sigma_i, \sigma_j)$ Predicate describing the effects of applying m with arguments $\overline{a}$ on state σ_i which results in state σ_j. The predicate is true iff σ_j contains the effects of applying $m(\overline{a})$ on σ_i.

Invariant $\texttt{inv}(m(\overline{a}), \sigma_i, \sigma_j)$ Predicate describing the behavior that is expected from (concurrent) calls to method m on state σ_i. The predicate is true iff the result state σ_j respects the invariants that are expected from applying m with arguments $\overline{a}$ on state σ_i. For example, referential integrity requires bids to be linked to existing users. This can be expressed as follows:

$$\texttt{inv}(bid(auction, usr, amount), \sigma_i, \sigma_j) = user(usr) \in \sigma_j$$

Preconditions and postconditions are similar to those in Hoare triples and are used to describe operations using first-order logic predicates that can be analyzed by Ordana. Invariants are used to define the concurrency semantics that is expected from concurrent operations. Thus, invariants correspond to the postconditions in SECROs. In the remainder of this section, we explain how preconditions, postconditions, and invariants are used by Ordana's analyses to answer the aforementioned questions.

4.3.2 Dependency Analysis

This section details how to detect dependencies between method calls. We borrow the notion of dependency from [HL19] and tailor it to meet ECROs' needs. Recall that users invoke methods on a replica and that *method calls* (or calls for short) are propagated to all replicas. Calls are allowed to execute at a replica only if their precondition holds, in which case we say they are enabled.

[4]Our implementation in Scala uses immutable collections. Methods return a modified copy of the state that replaces the old state.

[5]An overline, e.g. $\overline{a}$, means zero or more.

Definition 1: Enabled call

A call c is enabled by a given state σ iff its precondition holds in that state. Formally, $\forall c, \sigma \,.\, \texttt{enabled}(c, \sigma) \iff \texttt{pre}(c, \sigma)$. Two calls c_1 and c_2 are enabled by a state σ iff both are enabled by σ: $\forall c_1, c_2, \sigma \,.\, \texttt{enabled}(c_1, c_2, \sigma) \iff \texttt{enabled}(c_1, \sigma) \wedge \texttt{enabled}(c_2, \sigma)$.

Sequential calls may exhibit dependencies. Intuitively, a call c_2 depends on a prior call c_1 if c_2 cannot execute before c_1.

Definition 2: Independent and dependent calls

Let c_1 be a call that is enabled by state σ_0, σ_1 the state that results from executing c_1 on σ_0, and c_2 a call that is enabled by σ_1. We say that c_2 is independent of c_1 iff c_2 is also enabled by σ_0. Otherwise, c_2 depends on c_1, written as $dep(c_2, c_1)$. Formally, $\forall c_1, c_2 \,.\, \texttt{dep}(c_2, c_1) \iff \exists \sigma_0, \sigma_1 \,.\, \texttt{enabled}(c_1, \sigma_0) \wedge \texttt{post}(c_1, \sigma_0, \sigma_1) \wedge \texttt{enabled}(c_2, \sigma_1) \wedge \neg\texttt{enabled}(c_2, \sigma_0)$.

Ordana's dependency analysis detects potential dependencies between pairs of methods and determines under which conditions these dependencies occur. Let $\langle m_1, m_2 \rangle$ be the method pair we want to analyze. To determine whether m_2 could depend on m_1, the analysis checks the satisfiability of the following formula using an SMT solver:

$$\exists \overline{a}_1, \overline{a}_2 \,.\, c_1 = m_1(\overline{a}_1) \wedge c_2 = m_2(\overline{a}_2) \wedge dep(c_2, c_1)$$

If the formula is unsatisfiable, this constitutes a proof that no call to m_2 exists that is dependent on a call to m_1. If it is satisfiable, a counterexample exists in which a call to m_2 depends on a call to m_1. Ordana then restarts the analysis with equality relations between the methods' arguments to determine the root cause of this dependency. Although our approach cannot unravel the cause of all dependencies, it works well in practice since dependencies often occur due to calls referring to an argument introduced by a previous call. For example, `bid(auction2,20)` depends on `open(auction1)` only when `auction2 = auction1`.

The dependency analysis returns a function $\texttt{dep} :: \texttt{C} \times \texttt{C} \to \mathbb{B}$ that takes two calls and returns true if the first call depends on the second, false otherwise.

4.3.3 Concurrent Commutativity Analysis

Many RDTs leverage commutativity to ensure state convergence without coordinating concurrent method calls. This led researchers to design static analyses capable of detecting methods that commute when executed concurrently, based on some specification [Got+16; Li+12; Bal+15; Kul+11; Dim+14].

However, commutativity is a property of method calls, not of concurrency. ECROs leverage commutativity for both concurrent and sequential calls. In this section, we focus on commutativity of concurrent calls, Section 4.3.4 elaborates on commutativity of sequential calls.

Definition 3: Concurrent commutativity

Let c_1 be a method call generated in some state σ_a and c_1 a *concurrent* method call generated in some state σ_b. We say that c_1 and c_2 *concurrent commute*, written as $c_1 \overset{\text{c}}{\rightleftharpoons} c_2$, iff they commute on every state σ_0. Formally:

$$
\begin{aligned}
&\forall c_1, c_2 \,.\, c_1 \overset{\text{c}}{\rightleftharpoons} c_2 \iff \\
&\quad \forall \sigma_a \,.\, \texttt{enabled}(c_1, \sigma_a) \land \forall \sigma_b \,.\, \texttt{enabled}(c_2, \sigma_b) \land \\
&\quad \forall \sigma_0, \sigma_1 \,.\, \texttt{post}(c_1, \sigma_0, \sigma_1) \land \forall \sigma_2 \,.\, \texttt{post}(c_2, \sigma_0, \sigma_2) \land \\
&\quad \forall \sigma_{12} \,.\, \texttt{post}(c_2, \sigma_1, \sigma_{12}) \land \forall \sigma_{21} \,.\, \texttt{post}(c_1, \sigma_2, \sigma_{21}) \implies \\
&\qquad \texttt{enabled}(c_1, \sigma_0) \land \texttt{enabled}(c_2, \sigma_0) \land \\
&\qquad \texttt{enabled}(c_2, \sigma_1) \land \texttt{enabled}(c_1, \sigma_2) \land \sigma_{12} \equiv \sigma_{21}
\end{aligned}
$$

To detect non-commutative method calls, Ordana analyzes all method pairs. For every method pair $\langle m_1, m_2 \rangle$, the analysis checks whether two concurrent calls to these methods exist that do not commute. To this end, it checks the satisfiability of the following formula using an SMT solver:

$$
\begin{aligned}
&\exists \overline{a}_1, \overline{a}_2 \,.\, c_1 = m_1(\,\overline{a}_1\,) \land c_2 = m_2(\,\overline{a}_2\,) \land \\
&\exists \sigma_a \,.\, \texttt{enabled}(c_1, \sigma_a) \land \exists \sigma_b \,.\, \texttt{enabled}(c_2, \sigma_b) \land \\
&\exists \sigma_0, \sigma_1 \,.\, \texttt{post}(c_1, \sigma_0, \sigma_1) \land \exists \sigma_2 \,.\, \texttt{post}(c_2, \sigma_0, \sigma_2) \land \\
&\exists \sigma_{12} \,.\, \texttt{post}(c_2, \sigma_1, \sigma_{12}) \land \exists \sigma_{21} \,.\, \texttt{post}(c_1, \sigma_2, \sigma_{21}) \land \\
\neg(&\texttt{enabled}(c_1, \sigma_0) \land \texttt{enabled}(c_2, \sigma_0) \land \texttt{enabled}(c_2, \sigma_1) \land \\
&\texttt{enabled}(c_1, \sigma_2) \land \sigma_{12} \equiv \sigma_{21})
\end{aligned}
$$

If the above formula is unsatisfiable, this constitutes a proof that any two concurrent calls to m_1 and m_2, that are enabled by their respective initial states σ_a and σ_b, are enabled and commute. On the other hand, if it is satisfiable, concurrent calls to m_1 and m_2 exist that are not enabled (i.e. they cannot be applied one after the other) or do not commute. Ordana then restarts the analysis with equality relations between the calls' arguments to determine when this occurs. For example, concurrent `bid(auction1,10)` and `close(auction2)` calls always commute except when `auction1 = auction2`. The output of the analysis is a function `commutative` $:: \mathtt{C} \times \mathtt{C} \to \mathbb{B}$ that takes two calls and returns true if the calls concurrent commute, false otherwise.

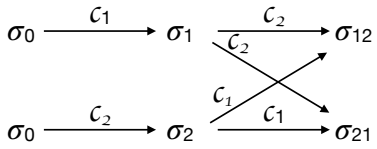


Figure 4.3: State equivalence.

Note that we did not yet define state equivalence because it can be implemented in several ways. We may introduce a predicate that tests for state equivalence. However, this requires programmers to carefully define state equivalence and thus complicates the development of ECROs. Instead, Ordana derives state equivalence from the methods' postconditions. The states that result from applying the calls, in different orders, are equivalent iff they preserve the same effects: $\sigma_{12} \equiv \sigma_{21} \iff \mathtt{post}(c_2, \sigma_1, \sigma_{21}) \wedge \mathtt{post}(c_1, \sigma_2, \sigma_{12})$. This is visualized in Fig. 4.3, horizontal lines stand for sequential executions. The calls concurrent commute iff swapping their order leads to the same states (diagonal lines).

4.3.4 Deriving Sequential Commutativity

Sequential method calls differ from concurrent method calls because they can contain additional dependencies. If a call c_2 depends on a call c_1, then it follows that c_1 happened before c_2 (denoted $c_1 \prec c_2$). However, the inverse does not hold: causal relations do not necessarily imply dependencies. For example, suppose a user opens an auction a_1, and then someone closes auction a_2. The closing of a_2 does not depend on the opening of a_1 because they affect different auctions. ECROs leverage this principle and allow sequential calls to be executed out of order if they commute and are independent; we say that those calls *sequentially commute*.

Definition 4: Sequential commutativity

Let c_1 and c_2 be sequential calls such that $c_1 \prec c_2$. We say that c_2 sequentially commutes with c_1, written as $c_2 \overset{\text{S}}{\rightleftharpoons} c_1$, iff c_2 does not depend on c_1 and they concurrent commute:
$\forall c_1, c_2 \centerdot c_2 \overset{\text{S}}{\rightleftharpoons} c_1 \iff \neg\texttt{dep}(c_2, c_1) \land c_2 \overset{\text{C}}{\rightleftharpoons} c_1.$

Ordana derives sequential commutativity based on the dependency analysis and the concurrent commutativity analysis presented in Sections 4.3.2 and 4.3.3. Since dependencies between calls are asymmetric, sequential commutativity is also an asymmetrical relation. The output of Ordana is a function `seqCommutative` :: $\texttt{C} \times \texttt{C} \to \mathbb{B}$ that takes two calls and returns true if the first call sequentially commutes with the second call, false otherwise.

4.3.5 Safety Analysis

ECROs guarantee that no method call leaves the replicas in a conflicting state, i.e. a state that violates the invariants defined in the data type's distributed specification. To this end, Ordana implements a safety analysis that detects pairs of concurrent methods that could infringe invariants, similar to [Bal+15; Got+16; HL19]. Moreover, Ordana introduces a novel technique to find coordination-free solutions to these conflicts. Before delving into the analysis, we define safety.

Definition 5: Safe calls

Two concurrent method calls are safe iff applying them (i.e. their postconditions) in *any* order preserves the methods' preconditions and invariants, otherwise, they are unsafe.

Definition 6: Safe methods

Two methods are safe iff all pairs of concurrent calls to those methods are safe.

Definition 7: Safe serialization

A serialization of enabled calls is safe iff all pairs of concurrent calls are safe or the ordering of concurrent calls preserves their invariants.

To identify unsafe methods, Ordana analyzes the invariants of all method pairs, and checks if some serialization of concurrent calls to these methods could violate the invariants. Given a method pair $\langle m_1, m_2 \rangle$, the safety analysis checks the satisfiability of the following formula:

$$\begin{gathered}
\exists \overline{a}_1, \overline{a}_2 \,.\, c_1 = m_1(\,\overline{a}_1\,) \wedge c_2 = m_2(\,\overline{a}_2\,) \wedge \\
\exists \sigma_0 \,.\, \texttt{enabled}(c_1, c_2, \sigma_0) \wedge \\
\exists \sigma_1 \,.\, \texttt{post}(c_1, \sigma_0, \sigma_1) \;\wedge \exists \sigma_{res} \,.\, \texttt{post}(c_2, \sigma_1, \sigma_{res}) \wedge \\
\neg(\,\texttt{enabled}(c_2, \sigma_1) \wedge \texttt{inv}(c_1, \sigma_0, \sigma_{res}) \wedge \texttt{inv}(c_2, \sigma_1, \sigma_{res})\,)
\end{gathered}$$

If the above formula is unsatisfiable, any two calls, c_1 to method m_1 and c_2 to method m_2, that are enabled by some initial state σ_0 preserve the calls' preconditions and invariants when applied one after the other ($c_1 < c_2$) on σ_0. This constitutes a proof that c_1 followed by c_2 is a safe serialization. On the other hand, if the formula is satisfiable, applying c_1 and c_2 in order on σ_0 violates a precondition or an invariant. Ordana then restarts the analysis with equality relations between the arguments to determine the cause of the conflict.

The output of the described safety analysis are two functions: $\texttt{restrictions} :: \texttt{C} \to R$ and $\texttt{resolution} :: \texttt{C} \times \texttt{C} \to \{<, >, \top, \bot\}$. The former, `restrictions`, takes a call c and returns a set R of restrictions. These are all the methods that require coordination because they may violate invariants when executed concurrently with c and no safe serialization exists. The set of restrictions informs the replication protocol which locks to acquire before executing a call c. These restrictions are fine-grained and take into account the argument of c that causes the conflict (if detected), in order to lock only part of the data. Consider again the RUBiS application, concurrent calls to `registerUser` may violate the invariant that usernames must be unique. Therefore, the analysis places a restriction on `registerUser` that locks the username passed to the call. As a result, users cannot register the same username concurrently because only one of the users will acquire the lock for the chosen username.

The latter function, `resolution`, takes two concurrent calls and returns $\top$ if the calls are safe. If the calls (say c_1 and c_2) are unsafe but a safe serialization exists it will return an ordering of the calls that is safe ($c_1 < c_2$ or $c_1 > c_2$). Otherwise, it returns $\bot$ since the calls require coordination, i.e. $\texttt{restrictions}(c_1) \neq \emptyset \vee \texttt{restrictions}(c_2) \neq \emptyset$.

4.4 Explicitly Consistent Replicated Objects

We now formally define ECROs and explain how the replication protocol uses the information inferred by Ordana to serialize method calls safely while minimizing coordination.

We represent an ECRO as a tuple $\langle \Sigma, \sigma_0, \mathtt{M}, \mathtt{G}, \mathtt{t}, \mathtt{F} \rangle$, where Σ is the set of possible states, σ_0 is the initial state, `M` is the set of methods, `G` is the object's execution graph, `t` is the current topological order of graph `G`, and `F` is the set of functions produced by Ordana (cf. Section 4.3). The execution graph $\mathtt{G} = \langle \mathtt{C}, \mathtt{E} \rangle$ is a labeled DAG where vertices (`C`) are method calls, and edges (`E`) express relations between calls. The protocol operates on this graph. In a nutshell, it considers three types of edges:

happened-before edges (hb-edges) enforce causality. For every pair of causally related calls a corresponding hb-edge is added to the graph if they do not commute. Causal relations between sequentially commutative calls are ignored since their order does not affect the outcome.

conflict-order edges (co-edges) enforce the invariants defined in the distributed specification (i.e. safety). For every two unsafe concurrent calls, the algorithm checks if a safe serialization of the calls exists. If that is the case, the corresponding co-edge is added between the calls. Consider again the add-wins set from Section 4.2.2. The protocol adds a co-edge from `remove(x)` to concurrent `add(x)` calls since Ordana proves that applying `remove(x)` before `add(x)` guarantees adds to win.

arbitration order edges (ao-edges) enforce state convergence. In some cases, concurrent calls are safe but do not commute. All replicas must execute those calls in the same order to guarantee strong convergence. Replicas order these calls deterministically by adding an ao-edge between them, whose direction is based on the globally unique identifiers of the calls.

By combining these three types of edges, any topological ordering of graph `G` is a safe serialization that preserves dependencies and guarantees strong convergence. Several topological orders may exist because the protocol does not add edges between safe calls that commute. However, different topological orderings only interchange commutative calls and thus lead to equivalent states. We prove this later in Section 4.4.3.

4.4.1 Replication Protocol

Algorithm 3 presents an overview of the replication protocol that is run by each ECRO replica. When the user invokes a method on a replica, it is handled by the replica's `execute_local` function, and later integrated at remote replicas using the `execute_remote` function.

Local Method Calls. Upon receiving a local request to execute method m with arguments $\overline{a}$, the `execute_local` function creates a new call c containing the method and its arguments $m(\overline{a})$, a globally unique identifier[6], and a logical timestamp[7]. If the call is unsafe, it is coordinated by acquiring the necessary locks (line 6). The `restrictions` function (returned by the safety analysis) leverages the call's arguments to ensure the right lock granularity. Since c is a new local request, all calls already contained by the replica's execution graph happened before c. Thus, call c is added to the graph and an `hb`-edge is added between c and every call that does not sequentially commute with c (line 10); these edges do not affect the topological ordering. Next, local call c is appended to the end of the current topological order (line 11), c is applied on the current state σ (line 12), and causally stable calls are committed to keep the graph small (line 13) as will be explained later. Lastly, the call is propagated to the other replicas and acquired locks are released after receiving confirmation from remote replicas that the call has been applied (line 17).

Integrating Remote Method Calls. Upon receiving a (safe or unsafe) remote call c, the `execute_remote` function adds c to the vertices (line 19) and adds the necessary edges to the graph (lines 20-32). For calls that happened before c the approach is the same as for local calls. For concurrent calls (line 23) we distinguish two cases. In the first case, calls c and v are unsafe, but a safe serialization exists. The algorithm then uses the resolution function returned by the safety analysis to determine the direction of the `co`-edge (i.e. how to order calls, lines 24-27). In the second case, calls c and v are safe but do not commute (line 28). To ensure convergence, the function uses the calls' identifiers to deterministically add an `ao`-edge between c and v. A dynamic topological sort [PK07]

[6]We combine Lamport clocks [Lam78] with unique replica identifiers to generate globally unique identifiers that are totally ordered.

[7]We use vector clocks but any logical timestamp that tracks causality can be used.

Algorithm 3 ECRO replication protocol main functions.

1: $\langle \Sigma, \sigma_0, \texttt{M}, \texttt{G}, \texttt{t}, \texttt{F} \rangle$, with $\texttt{G} = \langle C, E \rangle$ ▷ ECRO's internal state
2: σ: Σ ▷ object current state σ
3: **function** EXECUTE_LOCAL(m($\overline{a}$)) ▷ exec method m with args $\overline{a}$ at origin replica
4: c ← ⟨ m($\overline{a}$), uniqueId(), timestamp() ⟩ ▷ tag call with id and timestamp
5: **if** `restrictions`(c) ≠ ∅ **then** ▷ call c may be unsafe
6: acquire_locks(`restrictions`(c))
7: C ← C ∪ { c } ▷ add call c to the graph vertices
8: **for** v ∈ C ∧ v ≠ c **do** ▷ determine relevant `hb`-edges for call c
9: **if** not `seqCommutative`(c, v) **then** ▷ seq calls c and v do not commute
10: E ← E ∪ {⟨v, `hb`, c⟩} ▷ add `hb`-edge from call v to call c
11: `t` ← `t` + c ▷ local call c has no impact on topological order
12: σ ← apply(σ, c) ▷ execute call c on current state σ
13: commitStableCalls() ▷ commits previous calls if there is a single replica
14: propagate(c) ▷ propagate to remote replicas with eventual and causal delivery
15: **if** hasLocks() **then**
16: wait_ack() ▷ wait until all replicas executed the call
17: release_locks(`restrictions`(c))
18: **function** EXECUTE_REMOTE(c) ▷ execution of call c at remote replica
19: C ← C ∪ { c } ▷ add call c to the graph vertices
20: **for** v ∈ C ∧ v ≠ c **do** ▷ determine relevant edges (relations) for call c
21: **if** v ≺ c ∧ not `seqCommutative`(c, v) **then** ▷ call c and v do not commute
22: E ← E ∪ { ⟨v, `hb`, c⟩} ▷ add `hb`-edge between call v and call c
23: **else if** v ∥ c **then** ▷ call v is concurrent with call c
24: **if** `resolution`(c, v) = < **then** ▷ conflict solved by ordering c before v
25: E ← E ∪ { ⟨c, `co`, v⟩} ▷ add `co`-edge from call c to call v
26: **else if** `resolution`(c, v) = > **then** ▷ solve by ordering v before c
27: E ← E ∪ { ⟨v, `co`, c⟩} ▷ add `co`-edge from call v to call c
28: **else if** `resolution`(c, v) = ⊤ ∧ ▷ calls are safe but non-commutative
29: not `commutative`(c, v) **then**
30: **if** `Id`(c) < `Id`(v) **then** ▷ impose a deterministic order based on ids
31: E ← E ∪ { ⟨v, `ao`, c⟩} ▷ add `ao`-edge from call c to call v
32: **else** E ← E ∪ { ⟨v, `ao`, c⟩} ▷ add `ao`-edge from call v to call c
33: `t` ← dynamicTopologicalSort(`G`) ▷ dynamically compute new serialization
34: σ ← apply(σ_0, `t`) ▷ execute calls on initial state σ_0
35: commitStableCalls() ▷ commit prefix of causally stable calls

is performed on the execution graph to recompute only the subgraph that changed (line 33); these changes are limited to calls that are concurrent to c. Line 34 updates the state by applying, in order, the calls from the topological order on the initial state. Finally, on line 35, *causally stable* calls are committed to keep the execution graph small (discussed later in Section 4.4.1.1). Although not shown in the algorithm, if c is an unsafe call an acknowledgment is sent to the origin replica.

4.4.1.1 Optimizations

To keep the protocol efficient and avoid replaying the entire operation history for every incoming call, two optimizations were applied. First, causally stable calls are committed to keep the graph small. Second, snapshots of intermediate states are stored to minimize the calls that have to be recomputed. We now briefly elaborate on these optimizations.

Causal stability. A call c is *causally stable* [BAS17] at a replica r if r knows that all other replicas also observed c. Causal stability can be derived from the logical timestamps carried by calls when they are propagated to replicas; a call c with timestamp ts is causally stable at replica r, if ts happened before or is equal to the latest timestamp received from every other replica[8]. Based on this observation, r knows that no more calls that are concurrent to c can arrive. Replicas can thus - locally and without coordination - commit a prefix of causally stable calls as their positions are known to be fixed within the serialization.

How to commit a prefix of stable calls is defined by the `commitStableCalls` function in Algorithm 4. The *initial* state σ_0 is updated by applying the longest prefix of stable calls (line 9), whereafter, those calls and their incoming and outgoing edges can safely be removed from the graph (line 10 and 11).

Snapshots. Even if replicas commit causally stable calls, they need to replay the entire serialization of calls to compute the current state (line 34 in Algorithm 3). To address this issue, replicas take snapshots of intermediate states which enables efficient rollbacks to prior states. For

[8]Replicas can periodically send a no-op to ensure that calls stabilize even if some replicas do not generate calls.

Algorithm 4 Committing causally stable calls.

```
 1: ⟨Σ, σ0, M, G, t, F⟩, with G = ⟨C, E⟩                ▷ ECRO's internal state
 2: function commitStableCalls
 3:     stablePrefix ← true
 4:     i ← 0                                          ▷ number of causally stable calls
 5:     while i < |t| ∧ stablePrefix do                ▷ iterate over a prefix of stable calls
 6:         call ← t[i]
 7:         stablePrefix ← isStable(call)              ▷ determine stability from timestamp
 8:         if stablePrefix then
 9:             σ0 ← apply(σ0, call)                   ▷ update initial state
10:             C ← C \ { call }
11:             E ← (E \ in(call)) \ out(call)         ▷ remove incoming & outgoing edges
12:             i ← i + 1
13:     t ← drop(i, t)  ▷ remove the prefix of stable calls from the topological order
```

example, if the topological sort (line 33) results in $t = t_1.t_2$ where t_1 is unchanged and t_2 is the part of the serialization that changed, then the replicas can roll back to the snapshot of the state after t_1 such that only the calls in t_2 need to be replayed[9]. Note that if there is no snapshot available that corresponds to the state after t_1, the algorithm needs to roll back to an older snapshot (ideally the one that corresponds to the longest prefix of t_1). ECROs let programmers configure the *snapshot interval*, i.e. after how many calls a snapshot must be taken. This interval is a trade-off between latency and memory. The more snapshots replicas take, the less calls they need to replay but the more memory they use. The less snapshots replicas take, the less memory they consume but the more calls need to be replayed. We argue that the snapshot interval should be smaller (i.e. take more snapshots) if operations are costly and bigger if operations are fast.

4.4.1.2 Cycle Detection and Resolution

In some cases, reordering concurrent operations may introduce cycles. For instance, consider three operations a, b, and c such that $a \prec b$, $c \parallel a$, and $c \parallel b$. If a and b do not commute, replicas add an `hb`-edge from a to b. If b and c are unsafe, replicas may introduce a `co`-edge from b to c if that solves

[9]Note that dynamic topological sorting algorithms can return the index of the first change in the topological order such that we do not have to compute it manually based on the old ordering, which would be costly.

the conflict. Similarly, c and a may be unsafe and replicas may introduce a `co`-edge from c to a to solve the conflict. However, this introduces a cycle: $a \xrightarrow{hb} b \xrightarrow{co} c \xrightarrow{co} a$.

To keep the graph acyclic, we implement an efficient and deterministic approach that detects and solves cycles. The complete specification is in Appendix D. In a nutshell, when a newly added edge $c_1 \rightarrow c_2$ causes a cycle, at least one path from c_2 to c_1 exists. The protocol computes all paths from c_2 to c_1 and breaks them by removing one `ao`-edge on each path. These edges can be removed without putting convergence at risk as they impose an artificial ordering between non-commutative calls. Hence, we solved the cycle while keeping all non-commutative calls ordered. Occasionally, the cycle is caused by a combination of `hb`-edges and `co`-edges. These cannot be removed without violating convergence and safety. Instead, the algorithm deterministically discards a call that breaks the cycle. Information about discarded `ao`-edges and calls is propagated between replicas to ensure that all replicas eliminate the same `ao`-edges and/or calls and thus still converge. Note that discarding the operation that causes a cycle may cause anomalies observed by the clients. Future work could explore alternative ways to solve or avoid cycles.

4.4.2 Consistency Guarantees

The ECRO replication protocol described in Algorithm 3 ensures that every replica executes a serialization of the calls that respects dependencies between calls, totally orders non-commutative calls, and upholds the invariants defined by the specification. Thus, ECROs guarantee Explicit Consistency [Bal+15], a form of eventual consistency that is strengthened with application-level invariants (cf. Section 2.1).

With regard to the four session guarantees for weak consistency (cf. Section 2.1.2), ECROs guarantee Writes Follow Reads because for any write w that has a set of relevant writes (i.e. dependencies) W, the replication protocol orders all the relevant writes $w' \in W$ before w (cf. Line 21 and 22 in Algorithm 3).

If clients are sticky and cycles cannot occur, then ECROs also guarantee Read Your Writes and Monotonic Reads because all previously observed writes are part of the graph (if they are unstable) or part of the state (if they are stable and were committed). However, in our current imple-

mentation, cycles may cause ECROs to discard previously observed calls and thus no longer guarantee Read Your Writes and Monotonic Reads.

To circumvent these problems, Monotonic Reads can be guaranteed by reading only committed state since those calls are stable and will not be discarded. Similarly, ECROs can guarantee Read Your Writes by delaying calls (i.e. writes) until they are stable; from that point on, reads reflect prior writes. While delaying calls may seem unreasonable at first, our evaluation (cf. Section 4.6.5) shows that in geo-replicated systems, calls often stabilize faster than the time it takes to coordinate them. Still, delaying calls may affect the system's availability under network partitions.

Strictly speaking, ECROs do not guarantee Monotonic Writes because that requires writes to execute only after all previous writes within the same session were executed, but ECRO's replication protocol ignores causal relations if the calls commute. Thus, if $c_1 \prec c_2$ but the calls commute then ECROs may apply c_2 before c_1. Nevertheless, the outcome is the same because the operations commute. Thus, in practice, users do observe Monotonic Writes.

4.4.3 Protocol Correctness

We now prove that ECROs guarantee convergence and safety with respect to the data type's distributed specification. Since the replication protocol does not order pairs of calls that commute and are safe, several topological orders of the execution graph may exist. We prove that all topological orderings are safe (Theorem 1) and that replicas that received the same method calls converge to equivalent states (Theorem 2).

Theorem 1: Safe execution graphs

All topological orderings of an execution graph `G` of an ECRO replica are safe serializations.

Proof. By induction on the length of the topological ordering.
Base case. In the base case no calls occurred, so the topological ordering is empty and trivially safe.
Induction step. Assume replica r_1 has a topological order of dimension n that is safe. If a new call c is executed at replica r_1 two cases are possible: either c is a local call or a remote call.

Case 1. If c is a local method call, we distinguish two new cases depending on the information derived by Ordana. In the first case, the call is unsafe and Ordana determined a set of restrictions depending on the method calls that are conflicting. Algorithm 3 acquires the necessary locks to ensure that no unsafe call can execute concurrently, thereby guaranteeing safety. In the second case, Ordana found that c is safe with respect to all other possible calls. Hence, the replica can execute c and the resulting topological order(s) are safe serializations.

Case 2. If call c was propagated by replica r_2, then its execution was locally safe at replica r_2. We distinguish three cases depending on the information derived by Ordana. In the first case, the call is unsafe and the analysis did not find a solution. The originating replica r_2 then coordinated the call such that no conflicting call can execute concurrently, thereby, guaranteeing safety. In the second case, the call is unsafe but the analysis found a solution. If a conflicting call occurs concurrently to c, all replicas add the same `co`-edge between those calls. This `co`-edge guarantees safety since the analysis determined that this ordering preserves the invariants. In the third and final case, the call is safe with regard to all possible concurrent calls and thus cannot violate safety.

Thus, starting from an execution graph with a safe topological order of dimension n and a new call c, Algorithm 3 builds an expanded graph whose topological order of dimension $n+1$ is also a safe serialization. □

Before we can prove that replicas converge to equivalent states, we introduce a number of auxiliary lemmas and definitions.

Lemma 1: Convergent execution graphs

Two replicas of an ECRO that observed the same calls have the same execution graph:

$$\forall r_1 = \langle \Sigma, \sigma_0, \mathtt{M}, \mathtt{G}_1, \mathtt{t}_1, \mathtt{F} \rangle,\ r_2 = \langle \Sigma, \sigma_0, \mathtt{M}, \mathtt{G}_2, \mathtt{t}_2, \mathtt{F} \rangle.$$
$$\mathtt{G}_1 = \langle \mathtt{C}_1, \mathtt{E}_1 \rangle \wedge \mathtt{G}_2 = \langle \mathtt{C}_2, \mathtt{E}_2 \rangle \wedge \mathtt{C}_1 = \mathtt{C}_2 \implies \mathtt{E}_1 = \mathtt{E}_2 \implies \mathtt{G}_1 = \mathtt{G}_2$$

Proof. For every local or remote method call c, Algorithm 3 adds c to the replica's execution graph. Therefore, if both replicas observed the same calls, both execution graphs contain the same vertices. We now show that even if the (concurrent) calls were processed in a different order by these

replicas, their execution graphs contain the same edges. When a (local or remote) call c is received, Algorithm 3 checks the relation between c and every other call. Hence, independent of the order in which calls are processed, every call is eventually compared to every other call. For every pair of calls $\langle c_1, c_2 \rangle$, we distinguish two cases. In the first case, $c_1 \prec c_2$ or $c_2 \prec c_1$. If the operations sequentially commute, their order is not important. If the operations do not sequentially commute, their order is important and the algorithm ensures that both replicas add an `hb`-edge consistent with causality. In the second case, c_1 and c_2 are concurrent. Again, if the operations commute, their order is not important. However, if the operations do not commute, we consider two new cases. In the first case, Ordana's `resolution` function imposes an ordering between the calls. Both replicas will then add the same `co`-edge. In the second case, Ordana does not impose an ordering on these non-commutative calls. Both replicas will then add the same `ao`-edge between these calls based on the calls' globally unique identifiers. Thus, the algorithm ensures that both replicas add the same edges to the graph, therefore both graphs $\mathtt{G_1}$ and $\mathtt{G_2}$ are the same. □

Definition 8: Equivalent serializations

Two serializations t_1 and t_2 of a set of method calls `C` are equivalent iff every pair of non-commutative calls appears in the same order in both t_1 and t_2:

$$\forall t_1, t_2 \,.\, t_1 \equiv t_2 \iff \forall c_1, c_2 \in \mathtt{C} \,.\, \neg\mathtt{commutative}(c_1, c_2) \implies$$
$$(t_1[c_1] < t_1[c_2] \iff t_2[c_1] < t_2[c_2]) \wedge (t_1[c_1] > t_1[c_2] \iff t_2[c_1] > t_2[c_2])$$

where $t[c]$ returns the position of call c in serialization t.

Definition 9: Equivalent replicas

Two replicas r_1 and r_2 of an ECRO are equivalent iff they have the same execution graph G (i.e. observed the same calls) and their topological orderings of G are equivalent:

$$\forall r_1 = \langle \Sigma, \sigma_0, \mathtt{M}, \mathtt{G_1}, \mathtt{t_1}, \mathtt{F} \rangle, r_2 = \langle \Sigma, \sigma_0, \mathtt{M}, \mathtt{G_2}, \mathtt{t_2}, \mathtt{F} \rangle \,.$$
$$r_1 \equiv r_2 \iff \mathtt{G_1} = \mathtt{G_2} \wedge t_1 \equiv t_2$$

Lemma 2: Maximum one edge between method calls

The execution graph $\mathtt{G} = \langle \mathtt{C}, \mathtt{E} \rangle$ of ECRO replicas contains at most one edge between any two method calls.

Proof. When a method is called on a replica, it is handled by the `execute_local` function and later integrated at remote replicas using the `execute_remote` function (Algorithm 3). At the origin replica, `execute_local` adds an `hb`-edge from every previous non-commutative call v in `C` to c (line 10). Hence, there cannot be an `hb`-edge from c to v or any other type of edge between them. For every incoming call c, `execute_remote` considers two disjoint cases: $v \prec c$ and $v \parallel c$. The case where $c \prec v$ cannot occur because calls are propagated in causal order and we already observed v.

Case 1 ($v \prec c$): if v and c sequentially commute the algorithm does nothing, else, it adds an `hb`-edge from v to c (line 22). Hence, there cannot be an `hb`-edge from c to v, nor can there be any other type of edge (`co`-edge or `ao`-edge) between v and c since case 1 and 2 are disjoint.

Case 2 ($v \parallel c$): Calls v and c can be safe or unsafe. We thus distinguish two disjoint subcases.

Case 2.1: If v and c are unsafe then Ordana found an ordering of the calls that solves the conflict (either $c < v$ or $v < c$), otherwise Ordana would have restricted the calls and they cannot have executed concurrently (line 6). If `resolution(c, v) = <` then every replica adds a `co`-edge from c to v (line 25), and there cannot be an edge from v to c, nor can there be any other type of edge between them since case 1 and 2 are disjoint as well as case 2.1 and 2.2. The case where `resolution(c, v) = >` is analogous.

Case 2.2: In this case, calls v and c are safe. If v and c commute the algorithm does nothing, else, it deterministically adds an `ao`-edge from the call with the smallest ID to the call with the biggest ID (line 28 to 32). Since the identifiers are globally unique, all replicas add the same `ao`-edge between v and c and there cannot be an edge in the opposite direction. There also cannot be any other type of edge (`hb`-edge or `co`-edge) between v and c because case 1 and 2 are disjoint as well as case 2.1 and 2.2.

We thus proved that there can be at most one edge between any two method calls in the graph. □

Theorem 2: ECROs guarantee strong convergence

Two ECRO replicas that observed the same calls $\mathtt{C}$ converge to equivalent states. Formally:

$$\forall r_1 = \langle \Sigma, \sigma_0, \mathtt{M}, \langle \mathtt{C}_1, \mathtt{E}_1 \rangle, \mathtt{t}_1, \mathtt{F} \rangle, r_2 = \langle \Sigma, \sigma_0, \mathtt{M}, \langle \mathtt{C}_2, \mathtt{E}_2 \rangle, \mathtt{t}_2, \mathtt{F} \rangle \,.$$
$$\mathtt{C}_1 = \mathtt{C}_2 \implies r_1 \equiv r_2$$

Proof. We follow a proof by contradiction. Since both replicas r_1 and r_2 observed the same calls $\mathtt{C}$, we know from Lemma 1 that they have the same execution graph, $G_1 = G_2$ (where $G_1 = \langle \mathtt{C}_1, \mathtt{E}_1 \rangle$ and $G_2 = \langle \mathtt{C}_2, \mathtt{E}_2 \rangle$). This graph is constructed by successive applications of Algorithm 3. Now, assume that their states diverge, i.e. $\mathit{apply}\,(\sigma_0, \mathtt{t}_1) \not\equiv \mathit{apply}\,(\sigma_0, \mathtt{t}_2)$. Since r_1 and r_2 diverge we know that at least two non-commutative calls c_1 and c_2 occur in a different order in t_1 and t_2. Let's consider the case where $t_1[c_1] < t_1[c_2]$ and $t_2[c_1] \not< t_2[c_2]$. Since calls c_1 and c_2 do not commute we have to consider three distinct cases.

Case 1: the calls c_1 and c_2 are unsafe and Ordana found no safe ordering. The algorithm then coordinates the calls to avoid that they execute concurrently (line 6 in Algorithm 3), thus imposing a happened-before relation (`hb`-edge) between c_1 and c_2 (leading to $t_2[c_1] < t_2[c_2]$). We reach a contradiction since in t_2, by hypothesis, these calls appear in a different order ($t_2[c_1] \not< t_2[c_2]$) but from Lemma 2 it follows that there is at most one edge between any two calls, i.e. there cannot be an edge from c_2 to c_1 since there is already an `hb`-edge from c_1 to c_2.

Case 2: the calls c_1 and c_2 are unsafe but Ordana found a safe ordering of the calls (line 23 to 27 in Algorithm 3). Assuming that the resolution places c_1 before c_2, we again reach a contradiction since in t_2 these calls occur in a different order and there can be at most one edge between them. If the resolution places c_2 before c_1 we reach a similar contradiction because t_1 already has an edge from c_1 to c_2.

Case 3: the calls c_1 and c_2 are safe. Since the calls do not commute, the algorithm uses the calls' globally unique identifiers to deterministically order c_1 and c_2 using an `ao`-edge (line 28 to 32 in Algorithm 3). Assuming the arbitration relation orders c_1 before c_2, we reach a contradiction since in t_2 these calls occur in a different order and there can be at most one edge between them. If the arbitration relation orders c_2 before c_1, we reach a similar contradiction since in t_1 there is already an edge from c_1

to c_2. The other case where $t_1[c_1] > t_1[c_2]$ and $t_2[c_1] \not> t_2[c_2]$ can be argued likewise.

We showed that both topological orderings t_1 and t_2 keep the relative order of all non-commutative calls. It follows from Definitions 8 and 9 that the replicas converge. □

4.4.4 Implementation

We implemented a prototype of the ECRO approach in Scala. Ordana, our analysis tool, consists of two parts: a parser and an analyzer. The parser is based on Indigo [Bal+15] and translates the first-order logic formulas from an ECRO's distributed specification to Z3 formulas. The analyzer implements the analyses presented in Section 4.3, and executes them using a Java binding for Z3. The information derived by Ordana is written to a file and used at runtime by the ECRO replication protocol.

The implementation of the replication protocol uses a dynamic topological sort algorithm [PK07] provided by the JGraphT library[10] and takes snapshots of intermediate states to efficiently roll back concurrent calls when they are reordered. Snapshots are garbage collected once their state is stable. We also remove calls from the replica's execution graph once they are *causally stable* [BAS17]. As explained in Section 4.4.1.1, these optimizations are safe since no more concurrent calls can arrive, i.e. the order of the call in the serialization is stable across all replicas.

We integrated ECROs in Squirrel [DG19], our distributed in-memory key-value store for Scala built atop Akka. Listing 4.4 shows how to store ECROs in Squirrel. First, we set up Squirrel by piggybacking on an existing Akka actor system and cluster (Lines 2 to 4). Then, we define a key and an instance of an add-wins set ECRO (Lines 7 to 8). Afterward, we add the set to the store and Squirrel returns a future that resolves to a replica of the set (Line 11). In the background, Squirrel automatically replicates the set to all instances of the database. Finally, we use our local replica (which is always available) to add 5 to the set (Line 13) and register a callback that prints the elements of the set every time it changes (Line 15), i.e. every time a replica adds or removes an element.

[10] `https://jgrapht.org/`

Listing 4.4: Storing ECROs in the Squirrel distributed key-value store.

```
// Setup Squirrel
val system: ActorSystem = ... // an existing Akka actor system
val squirrel = Squirrel(system)
val store = squirrel.store

// Make a key and an add-wins set ECRO of integers
val key = Key[AWSet[Int]]("my-add-wins-set")
val awset = AWSet[Int]() // add-wins set ECRO

// Store the object in Squirrel
val replicaF: Future[AWSet] = store.add(key, awset)
replicaF.foreach(replica => {
  replica.add(5) // add 5 to the set
  // print the set every time it changes
  store.onChange(key, _ => println(replica.elements))
})
```

4.5 Qualitative Evaluation

We now conduct a qualitative evaluation of our work to assess if the ECRO approach is suitable for building geo-distributed applications. This leads to our first research question:

RQ1. Do ECROs simplify the development of RDTs compared to state-of-the-art approaches?

To answer RQ1, we first design and implement an extensive portfolio of RDTs using the ECRO approach. Then, we compare the implementation of two representative RDTs against implementations in state-of-the-art approaches.

4.5.1 Portfolio of ECRO Data Types

To demonstrate the applicability of the ECRO approach we implemented an extensive portfolio of RDTs and integrated them in Squirrel. Our portfolio covers existing RDTs (counters, flags, sets, maps, lists), new RDTs for which no prior (C)RDT design exists as they require coordination (stacks and queues), and RUBiS, a geo-distributed auction system that is built from sequential data types (i.e. without devising ad-hoc RDTs). We published a software artifact comprising the implementation of the complete portfolio. It is available at `https://doi.org/10.5281/zenodo.5410793`.

Data Type	Description and distributed semantics
Counter	Supports increments and decrements.
EW-Flag	Flag that can be enabled and disabled. Guarantees enable-wins semantics in case the flag is enabled and disabled concurrently.
DW-Flag	Similar to EW-Flag but guarantees disable-wins semantics.
AW-Set	Wrapper around Scala's built-in immutable set. Provides add-wins semantics similar to the OR-Set CRDT [Sha+11a].
RW-Set	Similar to AW-Set but provides remove-wins semantics.
AW-Map	Wrapper around Scala's built-in immutable map. Values can be complex objects and are updated by overriding the key with the new value. Provides add-wins semantics when the same key is added and removed concurrently, and last-writer-wins semantics for concurrent adds of the same key.
RW-Map	Similar to AW-Map but provides remove-wins semantics when a key is added and removed concurrently.
Stack	Stack allowing push, pop, and top operations. Push operations execute optimistically and are totally ordered. Pop operations are coordinated in order not to pop more elements than there are on the stack.
Queue	Wrapper around Scala's built-in immutable queue. Enqueue operations run optimistically and are totally ordered. Dequeue operations are coordinated to avoid dequeueing more elements than there are in the queue.
List	Provides operations to prepend, insert, and delete elements, and to map a function over the list.
RUBiS	eBay-like auction system similar to the RUBiS benchmark [EJ09].

Table 4.1: Portfolio of ECRO data types and their description.

Table 4.1 provides an overview of all the data types included in the portfolio, accompanied by a brief description. In the remainder of this section, we elaborate on the different data types included in the portfolio, except sets as they were already discussed in Section 4.2.2.

Counter. The counter data type stores a single integer value that can be incremented and decremented. Since those operations naturally commute they do not require coordination and are never reordered.

Flags. The flag data types store a boolean value that can be enabled and disabled. Enable sets the flag to true, while disable sets it to false. These two operations do not commute, hence, concurrent calls to enable and disable are totally ordered depending on the flag's semantics. The enable-wins flag requires enable to win over concurrent disables. To this end, it associates an invariant to the enable operation which states that the flag must be enabled after the enable operation and any concurrent operations executed. Similarly, the disable-wins flag ensures that disable wins over concurrent enables by means of a similar invariant atop the disable operation. Ordana found that enable-wins semantics (resp. disable-wins semantics) can be achieved by ordering enable (resp. disable) operations after concurrent disable (resp. enable) operations.

Maps. Similarly to sets, we implemented add-wins and remove-wins maps. If a key is added and removed concurrently, the key will still be in the add-wins map but not in the remove-wins map. If two replicas concurrently add the same key with different values, one of the two will win, ensuring last-writer-wins semantics. To achieve add-wins and remove-wins behavior, we add invariants to the add and remove methods, similar to those of sets, which state that the added/removed key must be present (or not) in the resulting map. Since concurrent additions of the same key with different values do not commute, the ECRO algorithm automatically enforces a total order of these updates, thereby guaranteeing last-writer-wins semantics out of the box.

Stack and queue. We discuss the implementation of stacks and queues together since their distributed behavior is analogous. Stacks and queues allow elements to be pushed (or enqueued) and popped (or dequeued).

New elements are pushed (or enqueued) asynchronously and concurrent calls to push (or enqueue) are totally ordered across all replicas since they do not commute. However, concurrent pops (or dequeues) require coordination otherwise there may be more concurrent pop (or dequeue) calls than there are elements on the stack. To the best of our knowledge these are the first replicated stack and queue data types that preserve the operation's sequential semantics.

List. The list data type provides methods to prepend elements to the list, insert elements after other elements in the list, delete elements from the list, and map functions over the list. We added a precondition to `insert` to ensure that the element after which to insert (called the reference element) exists. We also added an invariant to `insert` to ensure that the inserted element occurs in the resulting list and is not overwritten by a concurrent `map`. All methods are allowed to run optimistically, i.e. they do not require coordination. If two elements are inserted at the same position concurrently, the algorithm totally orders them across all replicas as those calls do not commute.

Concurrent insertions and deletions may lead to conflicts. Figure 4.4 shows the case where replica R1 inserts a new element e_4 behind e_3 while concurrently replica R2 deletes e_3. After exchanging the calls, R2 cannot insert e_4 because e_3 is no longer present in the list, thereby violating insert's precondition. R2 can solve this conflict by reordering the calls such that e_4 is inserted before deleting e_3. As detected by Ordana, this reordering is safe and only needed when deleting the reference element (e_3 in the previous example) of one or more concurrent insertions. Similarly, map operations are reordered before concurrent insertions in order not to modify newly inserted elements (as this would violate the invariant of `insert`).

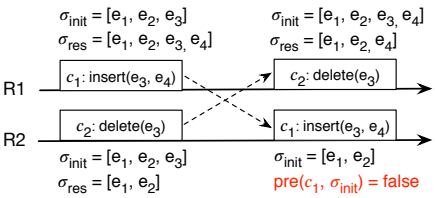


Figure 4.4: Conflict that requires R2 to reorder the calls.

RUBiS. Our RUBiS data type implements the eBay-like auction system introduced in Section 4.2.3. The RUBiS ECRO provides methods for

registering users, selling items, buying items, opening auctions, bidding on auctions, and closing auctions. Users must be unique, the stock of an item may not go below zero, and bids can only be placed on open auctions.

We associated a precondition to the `registerUser` operation that checks that the chosen username does not yet exist. The `sellItem(item, amount)` operation has a precondition that requires the amount that is sold to be strictly positive and an invariant that checks that the stock after applying `sell` and any concurrent operations contains at least the amount that is put for sale. The `storeBuyNow(item, quantity)` operation buys a certain quantity of a given item. Its precondition requires the quantity to be bigger than zero and smaller or equal to the number of items there are in stock. Its invariant requires the resulting state to have a non-negative stock for that item, i.e. concurrent `storeBuyNow` operations may not buy more items than there are in stock.

The `openAuction` operation creates an auction with a certain identifier. Its precondition requires that the auction does not yet exist, or if it already exists the auction must still be open. Thus, this precondition allows users to concurrently open the same auction. The `placeBid` and `closeAuction` operations have been discussed in detail in Section 4.2.3. The precondition of `placeBid` requires the auction to be open, the user to exist, and the bid to be bigger than zero. The `closeAuction` operation has no precondition or invariant. One could consider adding a precondition to `closeAuction` that requires the auction to be open. However, this would preclude users from concurrently closing an auction because only the first `closeAuction` operation would be enabled.

Table 4.2 shows the result of Ordana's safety analysis for the RUBiS RDT. The upper triangle of the matrix is omitted as the relations are

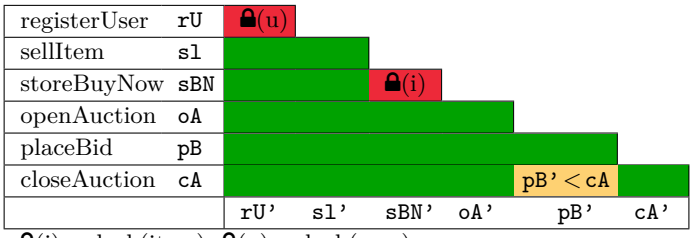


Table 4.2: Outcome of Ordana's safety analysis for RUBiS.

symmetrical. Most method pairs are safe (colored green). `placeBid` and `closeAuction`, however, do not commute and may lead to conflicts when a bid is placed on an auction that is closed concurrently (colored orange). Ordana found a solution to this conflict by ordering `placeBid` calls before concurrent `closeAuction` calls, and can uphold the invariants without coordination. Finally, `storeBuyNow` and `registerUser` are unsafe (colored red) because concurrent calls may lead to a negative stock or duplicate users. These conflicts cannot be avoided by reordering the calls, hence, ECROs coordinate them. To buy an item or register a user, the replica must first acquire a lock on the given item or user. This lock *only* restricts buying/registering the same item/user concurrently.

4.5.2 Comparison of ECROs Against Related Approaches

We now evaluate the impact of ECROs on the design and implementation of applications involving RDTs. We first compare the implementation of replicated sets with ECROs against well-known CRDT implementations [Sha+11a]. Then, we compare the RUBiS ECRO (cf. Section 4.2.3) against existing solutions such as PoR [LPR18] and RedBlue [Li+12].

4.5.2.1 Replicated Sets

We now compare the design and implementation of sets implemented with CRDTs and ECROs. The ECRO implementations have been discussed in detail in Section 4.2.2. Since ECROs exchange operations they are best compared to operation-based CRDTs. Therefore, we focus this discussion on operation-based set CRDTs. We implemented the CRDT designs described by Shapiro et al. [Sha+11a] in Scala. We do not compare to Akka's CRDT implementations because they only provide state-based CRDTs.

Operation-based CRDTs split every operation in two phases: a phase that prepares a message to be broadcast to every replica including itself (`prepare` method), and a downstream phase that applies such incoming messages (`downstream` method).

Listing 4.5 shows our implementation of an Observed-Removed Set (OR-Set) CRDT in Scala, which ensures add-wins semantics by associating a globally unique tag to every element it adds (lines 8-12). When some replica removes an element, it tells all replicas to remove only the tags it observed for that element (line 14). An element is part of the set if its set

of tags is non-empty (line 5). Since this design assumes that messages are delivered in causal order and replicas cannot remove the tags of elements that are added concurrently, the OR-Set guarantees add-wins semantics.

Listing 4.5: Implementation of an OR-Set CRDT in Scala.

```
1  case class Tag[ID](replica: ID, ctr: Int)
2  case class ORSet[V, ID](myID: ID, counter: Int,
3                          elements: Map[V, Set[Tag[ID]]]) {
4    def contains(e: V) =
5      elements.getOrElse(e, Set.empty[Tag[ID]]).nonEmpty
6    def prepareAdd(e: V): (V, Tag[ID]) =
7      (e, Tag(myID, counter + 1))
8    def addDownstream(tup: (V, Tag[ID])) = {
9      val (e, tag) = tup; val tags = elements.getOrElse(e, Set())
10     val newCtr = if (tag.replica == myID) tag.ctr else counter
11     ORSet(myID, newCtr, elements + (e -> (tags + tag)))
12   }
13   def prepareRemove(e: V): (V, Set[Tag[ID]]) =
14     (e, elements.getOrElse(e, Set()))
15   def removeDownstream(tup: (V, Set[Tag[ID]])) = {
16     val (e, tags) = tup
17     val knownTags = elements.getOrElse(e, Set())
18     ORSet(myID, counter, elements + (e -> (knownTags -- tags)))
19   } }
```

Listing 4.6: Implementation of a 2P-Set CRDT in Scala.

```
1  case class TwoPSet[V](added: Set[V], removed: Set[V]) {
2    def contains(element: V) =
3      added.contains(element) && !removed.contains(element)
4
5    def prepareAdd(element: V) = element
6    def addDownstream(element: V) =
7      TwoPSet(added + element, removed)
8
9    def prepareRemove(element: V) = element
10   def removeDownstream(element: V) =
11     TwoPSet(added, removed + element)
12 }
```

Providing remove-wins set semantics requires a completely different CRDT design. Shapiro et al. [Sha+11a] describe a Two-Phase Set (2P-Set) CRDT that guarantees remove-wins semantics. Listing 4.6 shows our implementation of the 2P-Set CRDT in Scala. It is a combination of two grow-only sets: `added` and `removed` (line 1). Elements are added by adding them to the `added` set and removed by *adding* them to the

`removed` set (lines 7 and 11). An element is considered in the set if it is in the `added` set and not in the `removed` set (line 3). A consequence of this design is that a removed element can never be added again.

Comparison. In contrast to CRDTs, ECROs allow developers to change the semantics of the set by modifying the specification instead of the implementation. This enables (1) RDT implementations to be reused, and (2) RDTs can exhibit different distributed semantics based on the application's needs, without rethinking the data type. As shown in Section 4.2.2, the add-wins and remove-wins set ECROs share the same sequential implementation (cf. Listing 4.1) and only differ in their specification (cf. Listing 4.2): the former associates an invariant to the `add` operation that guarantees add-wins semantics, while the latter associates an invariant to the `remove` operation that guarantees remove-wins semantics. In contrast, the OR-Set and 2P-Set CRDTs described in this section are completely different.

4.5.2.2 RUBiS Auction System

We now compare the implementation of RUBiS with ECROs (cf. Section 4.2.3) against its implementation with two state-of-the-art solutions: RedBlue and PoR consistency.

RedBlue and PoR require programmers to *manually* identify all conflicts that may violate application invariants and determine a set of restrictions that avoid these conflicts in order to guarantee state convergence and invariant preservation.

Table 4.3 shows that RedBlue requires 10 restrictions for the RUBiS system because it coordinates *all* unsafe shadow operations (i.e. all red operations are restricted pairwise). For example, the `registerUser` operation is labeled red because concurrent `registerUser` operations may violate the invariant that usernames must be unique. As a result, RedBlue imposes a restriction between `registerUser` and every other red operation, even though only concurrent `registerUser` operations are unsafe!

On the other hand, PoR only requires restrictions between pairs of operations that do not commute or are unsafe. Thus, PoR restricts concurrent `registerUser` operations but does not restrict `registerUser` from running concurrently with other operations. This results in only 3 restrictions for RUBiS.

RedBlue consistency	PoR consistency
r(registerUser, registerUser)	r(registerUser, registerUser)
r(storeBuyNow, storeBuyNow)	r(storeBuyNow, storeBuyNow)
r(placeBid, placeBid)	r(placeBid, closeAuction)
r(closeAuction, closeAuction)	
r(placeBid, closeAuction)	
r(registerUser, storeBuyNow)	ECRO
r(registerUser, placeBid)	r(registerUser, registerUser, <user>)
r(registerUser, closeAuction)	r(storeBuyNow, storeBuyNow, <item>)
r(storeBuyNow, placeBid)	
r(storeBuyNow, closeAuction)	

Table 4.3: Restrictions over the RUBiS operations enforced by RedBlue and PoR, taken from [LPR18] and extended with ECRO.

Comparison. In contrast to RedBlue and PoR, ECROs *automatically* derive a *minimal* set of restrictions based on the results of Ordana's safety analysis shown in Table 4.2. This results in only two restrictions (also shown in Table 4.3) because Ordana finds a solution for the conflict between `placeBid` and `closeAuction` and hence does not impose a restriction on those operations (cf. Section 4.5.1).

Similarly, Sieve [Li+14] automatically derives the restrictions for RedBlue consistency, based on a static analysis of a first-order logic specification of the operations. However, ECROs derive more fine-grained restrictions thanks to Ordana's novel safety analysis. For example, ECROs only restrict specific `registerUser` and `storeBuyNow` operations such that a single username or item is locked, whereas RedBlue and PoR restrict all `registerUser` and `storeBuyNow` operations from running concurrently, thus effectively locking all usernames or items.

We now compare ECROs to related work based on the complexity of programming RDT specifications. Automated approaches such as Sieve [Li+14], CISE [Got+16], and Explicit Consistency [Bal+15] require programmers to write plain strings containing first-order logic formulas describing the application invariants. In contrast, ECROs simplify this task by providing an embedded DSL for first-order logic in Scala. Our DSL simplifies the development of the specifications since (1) it provides support for common tasks (e.g. copying relations between states, expressing uniqueness constraints), (2) syntax errors and type errors are caught

by the compiler, and (3) programmers can leverage Scala's existing abstraction and modularization mechanisms (e.g. classes, traits, etc.).

4.5.3 Conclusion

The variety of RDTs comprised in our portfolio demonstrates that the ECRO approach is suited to implement RDTs from sequential code. Moreover, our comparison of replicated set implementations shows that the separation between the data type's implementation and its semantics (i.e. the distributed specification) facilitates the development and evolution of RDTs as one can modify the semantics without changing the implementation. The comparison of RUBiS implementations shows that our DSL for first-order logic simplifies the development of RDT specifications because programmers can leverage Scala's built-in abstraction mechanisms.

Based on the above observations, we conclude that the ECRO approach simplifies the development of RDTs when compared to state-of-the-art approaches (RQ1).

4.6 Performance Evaluation

We now shift our attention to the performance of the ECRO approach. We conduct several experiments to answer the following research questions:

RQ2. Can the static analyses, described in Section 4.3 and included in Ordana, be used in practice?

RQ3. Does the ECRO algorithm scale?

RQ4. How do ECRO-enabled geo-distributed applications perform compared to other approaches?

4.6.1 Methodology

The performance experiments reported in this section were conducted on Amazon EC2 `m5.xlarge` Virtual Machine (VM) instances. Each VM has 4 virtual CPUs and 16GiB of RAM. All benchmarks are implemented using JMH [Ope], a benchmarking library for the JVM that helps avoid common pitfalls, such as loop optimizations and dead code elimination [Pon14].

Each benchmark starts with a warmup phase, followed by the actual measurement phase consisting of 20 iterations. To avoid run-to-run variance we use the default setting of 5 JVM forks, which repeats the benchmark 5 times in fresh JVMs. This yields a total of 100 samples per benchmark.

4.6.2 Feasibility of the Static Analysis Phase (RQ2)

We measure the execution time of Ordana on the distributed specification of each data type in our portfolio of ECROs, presented in Section 4.5.1. The results, presented in Table 4.4, show that most data types are statically analyzed in less than 200ms. This results from the fact that their specifications are rather simple and concise. The execution times for the list data type and RUBiS applications are considerably higher. For lists, this is due to the complexity of the specification as operations manipulate references between the elements. For RUBiS, this comes from the fact that the data type has a bigger interface and thus more operations to analyze.

	Counter	*EW-Flag*	*DW-Flag*	*AW-Set*	*RW-Set*	
Time (ms)	58	67	73	93	95	
	AW-Map	*RW-Map*	*Stack*	*Queue*	*List*	*RUBiS*
Time (ms)	120	117	199	175	1732	4175

Table 4.4: Average time for Ordana to analyze ECRO specifications.

Based on these results, we conclude that Ordana is suited to analyze the distributed specifications of ECROs, as even the RUBiS application is analyzed in less than 5 seconds. Note that the analyses run at compile-time and *only* reanalyze distributed specifications that changed, thus, enabling their adoption within integrated development environments.

4.6.3 Scalability of the ECRO Protocol (RQ3)

The ECRO protocol maintains a DAG of tentative operations. We evaluate the scalability of the protocol with regard to the size of the DAG (i.e. the number of causally unstable operations [ASB15]) for three types of operations: (1) side-effect free operations, (2) operations that are safe and commute, and (3) unsafe operations that do not commute.

4.6.3.1 Comparison to Sequential Data Types

We now measure the latency of three operations on lists (`last`, `delete`, and `map`) which characterize the aforementioned types of operations. The benchmark runs on a single Amazon EC2 m5.xlarge instance and measures the latency of the three operations on an ECRO list containing 50K elements that is constructed by successive insertions: $insert(e_1, e_2); insert(e_2, e_3); ...; insert(e_{n-1}, e_n)$. We use Scala's built-in list as baseline and normalize the measurements. Figure 4.5 shows the relative latency of each operation.

Last. Returns the last element of the list. Since the operation has no side-effects, it executes immediately (no need to add it to the graph). As a result, we observe no significant performance difference compared to Scala's built-in list; the relative latency is approximately 1.

Delete. Removes the last element of the list. Recall from Section 4.5.1 that `delete(elem)` and `insert(ref, newElem)` commute when $elem \neq ref$. Since the list is built by successive insertions and this operation deletes the last element e_n, there are no dependent `insert(`e_n`, _)` operations in the graph. Hence, delete is safe and commutes with all operations from the graph. The ECRO algorithm adds the operation to the graph (O(1)) whereafter it executes the operation (lines 7 to 12 in Algorithm 3). As a result, the relative latency is also approximately 1.

Map. Maps a function over all elements of the list and does not commute with the insert operations that precede it. Algorithm 3 thus adds the operation to the graph (line 7) and adds an hb-edge between every existing `insert` operation and the new `map` operation (lines 8 to 10), before executing the operation (line 12). As a result, the performance of map decreases with the number of non-commutative operations in the graph (i.e. the number of edges that must be added).

The results show that ECROs exhibit latency similar to their sequential implementation for side-effect free operations and commutative operations. For non-commutative operations the latency decreases with the number of non-commutative operations in the graph. The experiment varied the size of the graph from 0 to 100 operations. In practice, the

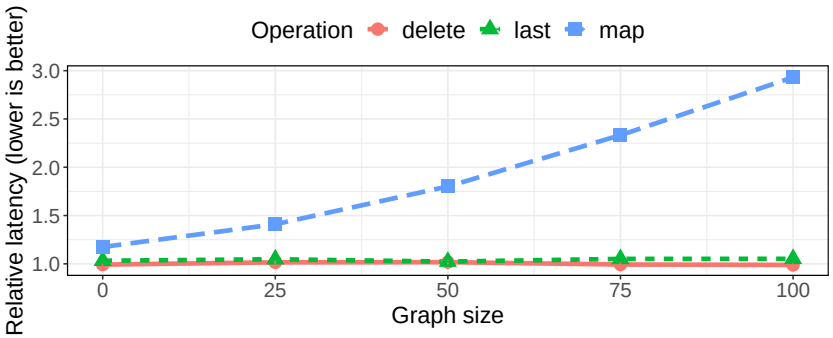


Figure 4.5: Latency of operations on an ECRO list. We disabled the JIT compiler to better show the impact of the graph's size on the latency of operations.

graph will rarely contain as much as 100 operations since the implementation detects causally stable operations and safely removes them from the graph (cf. Algorithm 4). Causal stability was disabled for this experiment in order to study the impact of unstable operations on the algorithm.

RUBiS. The previous list benchmark showcased the worst-case performance of the ECRO protocol since `map` did not commute with any operation in the execution graph. To get a better understanding of the protocol's scalability we conduct a similar experiment for RUBiS. We measure the latency of RUBiS operations on an ECRO and compare it to a sequential Scala implementation. In RUBiS, most operations commute, except when they affect the same auction, e.g. `openAuction`, `placeBid`, and `closeAuction`. The `getHighestBid` operation fetches the highest bid for a given auction and thus has no side effects. We measure the latency for each of these operations on a RUBiS system populated with 100 users, 1000 auctions, and 1000 items. Each operation is executed by a randomly selected user on a random auction.

Figure 4.6 shows that all operations exhibit *constant* latencies, with a negligible constant time difference between ECROs and Scala for mutating operations, which corresponds to the overhead of the ECRO protocol. The operations have constant latencies because only a fraction of the operations affect the same auction (i.e. do not commute). Based on this experiment, we conclude that the latency of ECRO operations depends on the number of non-commutative operations in the graph, which in prac-

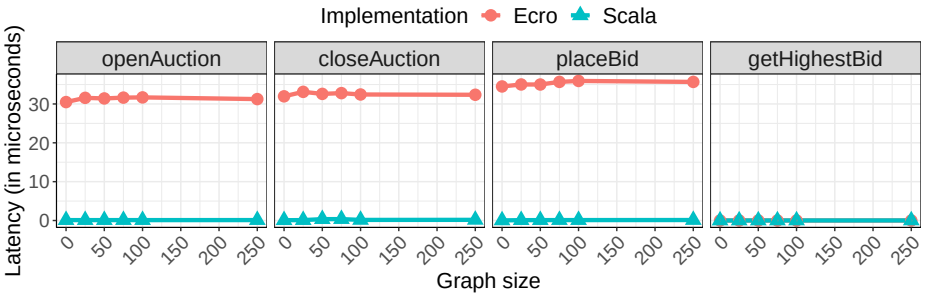


Figure 4.6: Latency of RUBiS operations.

tice is often small, especially if network connectivity is good and replicas commit causally stable operations.

4.6.3.2 Comparison to State-of-the-Art Set RDTs

We now compare the latency of operations for an add-wins set ECRO with the OR-Set CRDT (cf. Section 4.5.2.1) and our implementation of the pure operation-based add-wins set CRDT [BAS17] in Scala. Recall that `add(x)` and `remove(y)` operations commute, except when $x = y$.

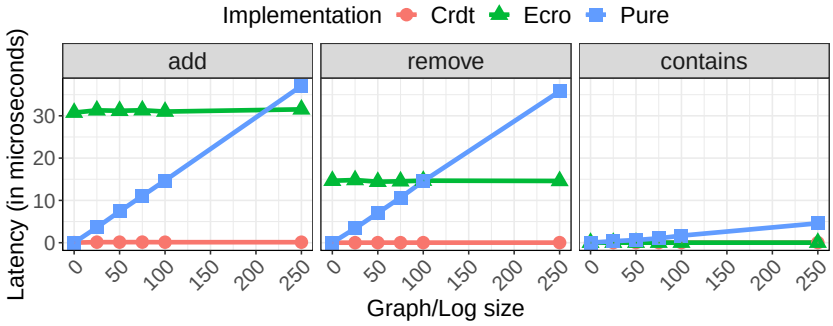


Figure 4.7: Latency of operations on add-wins sets for ECROs, CRDTs, and pure-op CRDTs.

We vary the size of the execution graph (for ECROs) and log (for pure-op CRDTs) by executing a random workload before measuring the latency of operations. Figure 4.7 depicts the results. The latencies remain *constant* for all operations of the add-wins set ECRO and the OR-Set CRDT. The `add` and `remove` operations have a negligible constant time

difference between those implementations (less than 0.03ms). In the pure add-wins set CRDT, the latency of operations is linear to the size of the log. This is because the pure operation-based approach checks new operations against all operations in the log in order to compact the log, whereas ECROs only check new operations against non-commutative operations in the graph. When the set's cardinality is big enough and the set operations are distributed uniformly across this space, the number of non-commutative operations is small and thus yields constant latency for ECROs.

4.6.4 Performance of a Geo-Distributed RUBiS Application (RQ4)

Besides the number of non-commutative operations contained by the graph, the performance of the ECRO algorithm also depends on factors such as the load experienced by the system, the latency between replicas, etc. We now compare ECROs with PoR and RedBlue on RUBiS by means of a variation on the RUBiS benchmark described in [LPR18].

Setup. The benchmark includes three RUBiS replicas and an independent lock server that coordinates unsafe operations. The RUBiS replicas run on Amazon EC2 `m5.xlarge` VM instances located in three geo-distributed Data Centers (DCs): Paris, Ohio, and Tokyo. The lock server runs on an `m5.xlarge` instance located in São Paulo.

The DC in Paris measures the latency of operations, while the DCs in Ohio and Tokyo execute an update-heavy workload consisting of 100 user requests[11] per second with 50% reads (side-effect free operations, e.g. `getStatus`) and 50% writes (mutating operations, e.g. `openAuction`). Workloads are generated from a probabilistic distribution of the operations. Table 4.5 shows the latencies and bandwidth between DCs.

Results. Figure 4.8 shows the average latency of RUBiS operations as observed by the user at DC Paris. The `getStatus` and `openAuction` operations are safe, hence, they are not coordinated, resulting in low latencies. The `storeBuyNow` and `registerUser` operations are unsafe and require coordination in all implementations (see Table 4.3), inducing high laten-

[11] In this context a user request corresponds to a method call on a replica.

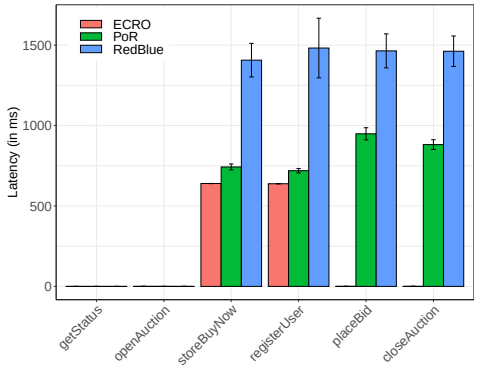


Figure 4.8: Average latency of RUBiS operations as observed by users at DC Paris. Error bars represent the 99.9% confidence interval.

Paris	44.3 us 4.78 Gbps			
Ohio	44.4 ms 140 Mbps	66.2 us 4.31 Gbps		
Tokyo	122 ms 49.5 Mbps	79.2 ms 77.9 Mbps	56.2 us 4.75 Gbps	
São Paulo	99.3 ms 61.8 Mbps	66.2 ms 97.5 Mbps	135 ms 46.3 Mbps	100 us 4.38 Gbps
	Paris	Ohio	Tokyo	São Paulo

Table 4.5: Average round trip latency and bandwidth between data centers.

cies. Nevertheless, the ECRO implementation reduces latency by more than 10% when compared to PoR and RedBlue. This speedup comes from the fact that ECROs use fine-grained locks on a single user/item, whereas PoR and RedBlue use coarse-grained locks on all users/items thereby preventing any `registerUser`/`storeBuyNow` operations from running concurrently. The `placeBid` and `closeAuction` operations exhibit high latencies for PoR and RedBlue because they are unsafe and require coordination (see Table 4.3). ECROs do not coordinate these operations because Ordana found a solution to the conflict, which consists of locally ordering `placeBid` operations before concurrent `closeAuction` operations when they affect the same auction (see Table 4.2 in Section 4.5.1). As a result, ECROs achieve low latency (less than 1ms).

We performed the same experiment with a read-mostly workload consisting of 1000 user requests per second with 95% reads and 5% writes. The results are similar and are explained in Appendix E.

4.6.5 Impact of Causally Unstable Operations on Scalability (RQ3)

As explained in Section 4.6.3, the latency of ECRO operations is related to the number of non-commutative operations in the execution graph.

The graph contains tentative operations, i.e. operations that are not yet causally stable and may be reordered (cf. Section 4.4.1.1).

We now turn our attention back to RQ3 and investigate the impact of causally unstable operations on the scalability of the ECRO algorithm. To this end, we measure the time to causal stability using the geo-distributed RUBiS deployment from Section 4.6.4. Recall that the RUBiS ECRO avoids coordination between the `placeBid` and `closeAuction` operations. Thus, users may close an auction and concurrently place a bid on that same auction. Upon delivery of the bid, `closeAuction` is reverted, the bid is placed, and the auction is closed again. Hence, there is a time window between closing the auction and declaring the winner, during which new bids may still arrive. Only when `closeAuction` becomes causally stable, the replica declares a winner. This is possible because our implementation exposes a hook that programmers can use to be notified when an operation stabilizes.

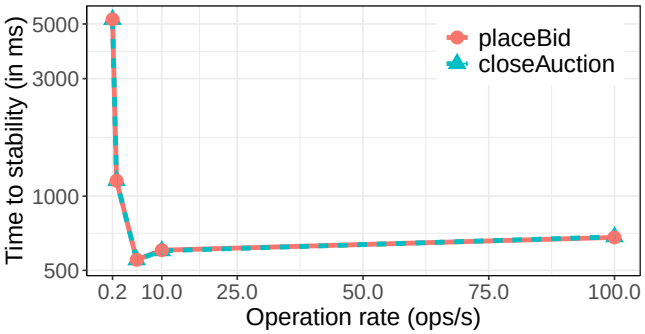


Figure 4.9: Time to stability for `placeBid` and `closeAuction` in function of the rate of operations in a geo-distributed RUBiS deployment.

Figure 4.9 shows the time to stability for the `placeBid` and `closeAuction` operations in function of the rate at which replicas generate operations. The time to stability quickly decreases with the rate of operations because ECROs derive stability from the logical timestamps of incoming operations. If replicas generate an operation every 5 seconds (0.2 ops/s), it takes on average 5 seconds for any operation to stabilize. When replicas generate 5 operations per second, the time to stability decreases to 550ms. Further increasing the rate of operations does not decrease the

time to stability due to network latencies, the load experienced by the system, etc.

Based on the aforementioned results, we conclude that the time to stability is inversely related to the rate at which replicas generate operations. When the rate is high enough (at least a few operations per second), operations stabilize faster than a coordinated execution. Indeed, at a rate of 5 ops/s and more, operations stabilize within 680ms whereas a coordinated execution of `placeBid` or `closeAuction` takes at least 880ms (cf. Fig. 4.8). In a cloud computing context, we can reasonably assume that DCs are well interconnected and generate operations regularly. Therefore, operations stabilize quickly and the replicas' execution graphs remain reasonably small which yields low latency and good scalability. If some replicas do not generate operations regularly, the time to stability can be reduced by sending acknowledgements for incoming operations, or, by having each replica periodically broadcast its logical clock. For our deployment, replicas could broadcast their clock every 140ms which corresponds roughly to the maximum latency between our DCs (cf. Table 4.5).

4.7 Notes on Related Work

As explained in Section 2.2.1.2, invariant-preserving RDTs such as Hamsaz [HL19] and Quelea [SKJ15] guarantee state convergence by coordinating non-commutative operations. The ECRO approach is different as it does not coordinate non-commutative operations, but instead, deterministically orders them at each replica, in a way that respects causality between dependent operations. Similarly, most invariant-preserving RDTs coordinate unsafe operations to avoid conflicts (i.e. invariant violations) at runtime. In contrast, ECROs allow unsafe operations to run concurrently if a safe reordering of the calls exists.

To enable the ECRO replication protocol, our static analysis tool, Ordana, incorporates several static analyses. The first is a commutativity analysis that detects pairs of non-commutative operations and is similar to well-known analyses [Got+16; Li+12; Bal+15; Kul+11; Dim+14]. The second is a dependency analysis that detects dependencies between sequential operations. To this end, it extends the work of Hamsaz [HL19] by taking into account relations between parameters which enables the detection of fine-grained dependencies between operations. The last is a safety

analysis that detects pairs of operations that may infringe application-level invariants when executed concurrently (similar to [Bal+15; Got+16; HL19]), and incorporates a novel technique to find solutions without imposing coordination, based on fast local reorderings of conflicting calls.

Quelea [SKJ15], CISE [Got+16], and Q9 [Kak+18] statically assign a consistency level to each operation of an RDT. Sieve [Li+14] combines static and dynamic analyses to detect invariant-breaking operations and execute them under strong consistency. These approaches may strengthen the consistency level of many operations, thereby, increasing the latency of user requests and deteriorating the system's scalability and availability.

Several programming languages and programming models [MM18; De +20; Köh+20; ZN16; MSD18; Hol+16] support mixing consistency levels safely to some extent. Indigo [Bal+15] coordinates unsafe operations or requires programmers to provide a deterministic and monotonic algorithm to repair broken invariants. IPA [Bal+18] detects operations that break invariants whereafter programmers must incorporate a suitable conflict resolution and/or coordination technique. While both, Indigo and IPA, start from existing RDTs providing state convergence (e.g. CRDTs) and extend them with invariants, the ECRO approach focuses on deriving those RDTs automatically from a sequential implementation and its distributed specification.

4.8 Conclusion

ECROs provide a principled approach to implement RDTs by augmenting sequential data types with a distributed specification that describes the semantics of concurrent operations through invariants over replicated state. Our static analysis tool Ordana analyzes distributed specifications to detect conflicts, unravel their cause, and find appropriate solutions. This suffices to automatically derive a replicated version of the data type that guarantees convergence and preserves program invariants efficiently.

Key to making this approach efficient is the static analysis phase that derives additional information about the data type. Replicas leverage this information to construct a conflict-free serialization of the operations without coordination. ECROs can solve certain types of conflicts by locally reordering the calls instead of coordinating them. This reduces the

latency of operations by several orders of magnitude when compared to state-of-the-art approaches such as RedBlue and PoR.

The ECRO approach partially adheres to the principles outlined in our research vision (cf. Section 1.2). Existing data types are turned into RDTs instead of designing dedicated RDTs from scratch (principle 1). Thanks to the underlying replication protocol, the resulting RDTs converge and preserve application-specific invariants out-of-the-box (principle 2), assuming the specifications are correct. Although programmers need to augment the data types with an additional specification, we built a DSL to program these specifications. The DSL is embedded in Scala and provides language support to implement these specifications by leveraging Scala's traditional abstraction mechanisms (principle 3).

ECROs exemplify the tension between simplicity and efficiency. The ECRO approach achieves excellent performance by introducing a static analysis phase, but this burdens the programmer, who needs to write a separate specification for each RDT.

Chapter 5

A High-Level Programming Language for Efficient RDTs

To simplify the development of RDTs, programming abstractions must strike a good balance between simplicity and efficiency such that they can be used by mainstream software engineers and scale to the workloads experienced by real-world applications.

In our search for programming abstractions for the development of RDTs, we proposed two approaches: SECRO (cf. Chapter 3) and ECRO (cf. Chapter 4). The SECRO approach is simple but inefficient: programmers extend sequential data types with application invariants written in the same language as the RDT implementation, but the replication protocol has to consider all possible serializations of the operations at runtime. In contrast, the ECRO approach is efficient thanks to a static analysis phase that detects conflicts and finds solutions beforehand. However, programmers must write more complicated specifications for their RDTs, using a DSL for first-order logic.

In this chapter, we reconcile both approaches in order to design a *simple* yet *efficient* programming abstraction for RDTs. The result is EFx, a simple programming model for the development of RDTs that is inspired by SECRO and is combined with ECRO's replication protocol.

The remainder of this chapter is structured as follows. Section 5.1 kicks off with a discussion on the programming efforts that are needed to write ECROs, in particular their distributed specification. Then, we introduce the envisioned solution to simplify the development of those specifications.

Section 5.2 introduces EFx and its SECRO-like programming model for the development of efficient RDTs. Section 5.3 defines the semantics of EFx and Section 5.4 details its integration with ECROs. In Section 5.5, we conduct a qualitative evaluation that assesses the programmability of RDTs built atop EFx compared to the original ECRO approach. We evaluate the performance of EFx in Section 5.6. Finally, we discuss the main design decisions behind EFx and its limitations in Section 5.7, compare it to related work in Section 5.8, and conclude in Section 5.9.

5.1 Motivation

We previously introduced the ECRO approach to programming RDTs. This section starts by discussing the shortcomings of hybrid approaches such as ECROs with respect to the problems identified in the introduction (cf. Section 1.1.3). This motivates the need for an improved programming model. We then show how we envision the resulting programming model to be integrated into a novel programming language, called EFx, that enables the development of ECROs without first-order logic specifications.

5.1.1 Shortcomings of Hybrid Approaches

We previously proposed the ECRO programming model which lets programmers build custom RDTs by extending sequential data types with application-specific invariants described in a separate specification. Although ECROs address the problems of non-customizable semantics and limited application invariants (cf. Section 1.1.3), they still feature disconnected specifications much like existing hybrid approaches. This is problematic because programmers need to write separate specifications, typically in first-order logic, such that SMT solvers can analyze them.

Even though ECROs provide a DSL for the development of distributed specifications, writing such specifications for advanced RDTs is cumbersome and error-prone. Moreover, the specifications must evolve along with the data type implementation which complicates software evolution. For example, the ECRO specification of the RUBiS auction system (cf. Section 4.5.1) consists of 110 Lines of Code (LoC) while the data type's implementation is only 73 LoC. Furthermore, ECROs assume that an operation's postcondition captures all effects of that operation but this is

never verified. Subtle errors in the postcondition may cause the analysis to derive wrong information which at runtime may cause replicas to diverge or break invariants.

We believe that the disconnected specifications complicate the development of RDTs with the ECRO approach. Even though ECROs provide excellent performance, having to write specifications in first-order logic hampers mainstream programmers from building custom RDTs.

5.1.2 The Need for a High-Level Analyzable Language

To remove the need for separate specifications, our vision consists of building a novel programming language, called EFx, that can be completely encoded in first-order logic. If every language construct can be encoded in logic, it follows that any program built atop these constructs can also be encoded in that logic. As a result, EFx programs are analyzable out of the box since they can be compiled to first-order logic in order to automatically analyze them using SMT solving.

Based on our experience writing ECRO RDTs, we noticed that the biggest part of their specification consists of the definition of first-order logic relations and the definition of postconditions that describe the effects of operations. However, this information is redundant as the postconditions are a repetition of the operations' implementation but then in logic.

EFx enables programmers to develop RDTs without having to encode the effects of operations in a separate specification because the data type implementation (i.e. the EFx program) itself is the specification. Programmers can mark data types as "replicated" and attach *concurrency contracts* consisting of application-specific preconditions and invariants on the operations. Such contracts are similar to state validators in SECROs but are fully analyzable. Programmers do not need to write postconditions describing the effects of operations because they are automatically derived from the operations themselves. Like SECROs, the contracts are written in EFx itself and not in a separate language. EFx compiles the data type implementation and its concurrency contract to a distributed specification in first-order logic in order to synthesize a correct ECRO.

To exemplify the EFx approach, consider again the implementation of a remove-wins set RDT, akin to the remove-wins set ECRO from Section 4.2.2, but this time using concurrency contracts in EFx. Listing 5.1 depicts the implementation of the `RWSet` class which defines a field `set`

Listing 5.1: Implementation of a Remove-Wins Set RDT in EFx.

```
1 @replicated
2 class RWSet[V](set: Set[V]) {
3   def contains(x: V) = this.set.contains(x)
4   def add(x: V) = new RWSet(this.set.add(x))
5   inv remove(x: V) { !this.contains(x) }
6   def remove(x: V) = new RWSet(this.set.remove(x))
7 }
```

containing the elements and defines methods to check if an element is in the set (`contains`), to add elements to the set (`add`), and to remove elements from the set (`remove`). The class attaches an invariant to the `remove` operation (Line 5) to specify that removed elements should not re-appear, even if the element is added concurrently. The `RWSet` class is marked as `@replicated` which tells EFx to synthesize an ECRO for it. The resulting ECRO can be deployed in distributed systems; e.g. on top of Squirrel [DG19], our distributed-key value store for Scala.

When comparing the implementation of the `RWSet` with the ECRO implementation from Section 4.2.2 we notice two major differences. First, the ECRO distributed specification had to define postconditions for the `add` and `remove` operations in order to encode these operations in first-order logic such that they can be analyzed. In contrast, EFx no longer requires programmers to define postconditions for the operations because it can infer them directly from the implementation. EFx thus avoids the need for postconditions which was the most cumbersome part of the specification since programmers had to define them for *all* operations. Second, the ECRO distributed specification defined an invariant for the `remove` operation which was expressed in first-order logic. In EFx, this invariant is implemented in EFx itself, thereby, alleviating the need for low-level (and disconnected) specifications in first-order logic. We thus designed a remove-wins set RDT in only 7 LoC!

The set implementation in EFx reminds us of the original SECRO programming model in which programmers extend sequential data types with preconditions and invariants (called postconditions in SECROs) written in the same high-level language. However, EFx's integrated analysis capabilities enable it to synthesize correct ECROs from high-level data type implementations. Thus, the resulting RDTs are fully verified and efficient.

5.2 The EFx Language

This section describes EFx, our novel programming language that combines a SECRO-like programming model based on concurrency contracts with automated analyses of those contracts to synthesize correct ECROs.

EFx was designed with three goals in mind: simplicity, efficiency, and correctness. First, EFx must be simple such that mainstream programmers can use it to build custom RDTs. To ensure simplicity, we designed EFx to be reminiscent of Scala and integrated a SECRO-like programming model based on concurrency contracts for the development of RDTs. Second, the resulting RDTs must be efficient such that they can be used in real-world applications. To this end, we integrated EFx with the ECRO analyses and replication protocol. Third, the resulting RDTs must be correct. To this end, we removed the need for separate specifications and instead automatically derive correct distributed specifications from the data type's high-level concurrency contracts. This avoids subtle mismatches between the data type implementation and its specification.

We introduce EFx's overall architecture in Section 5.2.1. Afterward, we present its syntax in Section 5.2.2 and define concurrency contracts in Section 5.2.3. Finally, Section 5.2.4 describes EFx's built-in collections which form the basis for the development of custom RDTs. EFx's type system is defined in Appendix F.

5.2.1 Overall Architecture

Figure 5.1 provides an overview of EFx's architecture. EFx uses Scala Meta[1] to parse EFx source code into an Abstract Syntax Tree (AST) representing the program. This is possible because every piece of EFx code is valid Scala syntax (but not necessarily semantically correct).

RDTs written in EFx or any other EFx program can be analyzed to check certain properties (e.g. commutativity) and can be transpiled to mainstream languages in order to be integrated into existing applications. Transpilation is done by the compiler which features *compiler plugins*. These plugins dictate the compilation of the AST to the target language. Currently, EFx comes with compiler plugins for Scala, JavaScript, and SMT-LIB[2], a standardized language for SMT solvers. Support for other

[1] `https://scalameta.org/`

[2] `http://smtlib.cs.uiowa.edu/`

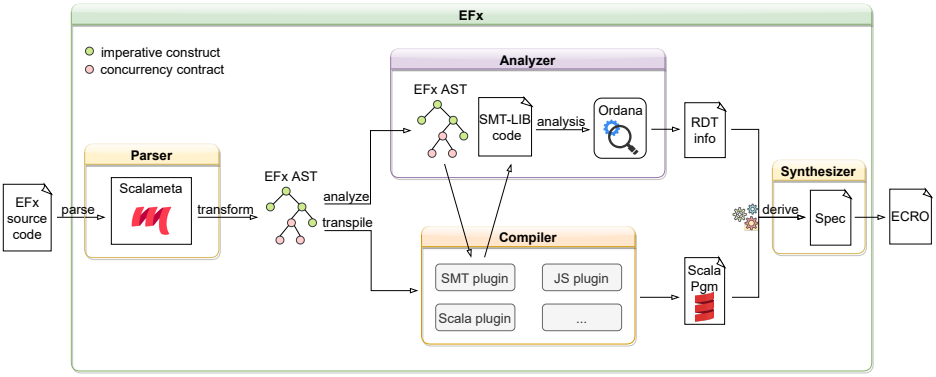


Figure 5.1: EFx's architecture.

languages can be added by implementing a compiler plugin for them. Program analyses are done by the analyzer which uses the SMT compiler plugin to compile the program's AST to SMT-LIB. The resulting SMT-LIB program can then be analyzed using traditional SMT solvers.

Together, EFx's analysis and transpilation capabilities enable the development of efficient RDTs by leveraging ECROs. RDTs written in EFx are compiled to SMT-LIB and the resulting SMT-LIB program is passed to Ordana, our static analysis tool for ECROs. Recall from Section 4.3 that Ordana applies various analyses to derive information about the commutativity and safety of operation pairs. This information is then written to a file. In addition, EFx compiles the RDT's AST to Scala. The synthesizer uses the resulting Scala code and the information derived by Ordana to derive a distributed specification for the data type. The specification is combined with the compiled Scala program in order to synthesize a corresponding ECRO in Scala. The resulting ECRO is equivalent to a manual implementation but the specification's postconditions are derived automatically. At the time of writing, EFx cannot yet generate ECROs in JavaScript because we do not have a JavaScript implementation of the ECRO protocol.

5.2.2 Syntax

As mentioned before, the syntax of EFx is inspired by Scala. Figure 5.2 defines the syntax rules of EFx. The metavariable C ranges over class names; I ranges over trait names; T, P and Q range over types; X and

$$
\begin{aligned}
A &::= @\mathit{replicated} \\
F &::= \texttt{trait}\ I\langle \overline{X} <: \overline{T}\rangle\,\{\,\overline{B}\,\} \\
&\mid \texttt{trait}\ I\langle \overline{X} <: \overline{T}\rangle\ \texttt{extends}\ I\langle \overline{P}\rangle\,\{\,\overline{B}\,\} \\
L &::= \overline{A}\ \texttt{class}\ C\langle \overline{X}\rangle\,(\overline{v} : \overline{T})\,\{\,\overline{D}\,\} \\
&\mid \overline{A}\ \texttt{class}\ C\langle \overline{X}\rangle\,(\overline{v} : \overline{T})\ \texttt{extends}\ I\langle \overline{P}\rangle\{\,\overline{D}\,\} \\
B &::= \mathit{valDecl} \mid \mathit{methodDecl} \mid M \\
D &::= M \mid \mathit{Pre} \mid \mathit{Inv} \\
M &::= \texttt{def}\ m\langle \overline{X}\rangle\,(\overline{x} : \overline{T}) : T = e \\
\mathit{Pre} &::= \texttt{pre}\ m\langle \overline{X}\rangle\,(\overline{x} : \overline{T})\,\{\,e\,\} \\
\mathit{Inv} &::= \texttt{inv}\ m\langle \overline{X}\rangle\,(\overline{x} : \overline{T})\,\{\,e\,\} \\
e &::= \texttt{num} \mid \texttt{str} \mid \texttt{true} \mid \texttt{false} \\
&\mid e + e \mid e - e \mid e * e \mid e / e \mid e \,\&\&\, e \mid e \,||\, e \mid e == e \\
&\mid e \,!= e \mid e < e \mid e <= e \mid e > e \mid e >= e \\
&\mid\ !e \mid x \mid e.v \mid e.m\,\langle \overline{T}\rangle\,(\,\overline{e}\,) \\
&\mid \texttt{val}\ x : T = e\ \texttt{in}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
&\mid (\ddot{x} : \ddot{T}) \Rightarrow e \mid e(\overline{e}) \mid \texttt{new}\ C\langle \overline{T}\rangle(\,\overline{e}\,) \\
\mathit{valDecl} &::= \texttt{val}\ x : T \\
\mathit{methodDecl} &::= \texttt{def}\ m\langle \overline{X}\rangle(\overline{x} : \overline{T}) : T \\
T &::= \texttt{int} \mid \texttt{string} \mid \texttt{bool} \mid C\langle \overline{T}\rangle \mid I\langle \overline{T}\rangle \mid \ddot{\overline{T}} \to T
\end{aligned}
$$

Figure 5.2: Syntax definition of EFx.

Y range over type variables; v ranges over field names; x and y range over parameter and variable names; m ranges over method names; and e ranges over expressions.

EFx programs can define classes $C\langle\overline{X}\rangle$ and traits $I\langle\overline{X}\rangle$ which can be polymorphic and may inherit from a single trait. Classes contain zero or more[3] fields and (polymorphic) methods. The body of a method must contain a well-typed expression e. Traits can define methods with default implementations and declare values and abstract methods that must be provided by concrete classes extending the trait. Traits can express upper type bounds on their type parameters to restrict the possible extensions.

EFx supports a variety of expressions, including literal values, arithmetic and boolean operations, field accesses $e.v$ and method calls $e.m\ \langle\overline{T}\rangle\,(\,\overline{e}\,)$, variable definitions, if tests, anonymous functions, function calls, and class instantiations. Functions are *first-class* and take at least one argument because nullary functions are constants.

[3] An overline, e.g. $\overline{X}$, denotes zero or more. A dashed overline, e.g. $\ddot{\overline{X}}$, denotes one or more.

EFx supports single inheritance from traits to foster code re-use but imposes some limitations. For example, the arguments of a class method must be concrete (i.e. cannot be a trait type) because Ordana must be able to reason about these methods but supertypes require reasoning about all subtypes and these may not necessarily be known at compile time.

5.2.3 Replicated Data Types and Concurrency Contracts

We previously defined EFx's core syntax. We now explain how EFx supports the development of RDTs by means of concurrency contracts.

EFx features an `@replicated` annotation that can be used to mark sequential data types for which EFx must generate an ECRO. Annotated classes can have methods with associated preconditions and invariants which form the data type's concurrency contract. Preconditions (resp. invariants) are defined using the keyword `pre` (resp. `inv`) followed by the name of the method, its parameter list, and a body consisting of a well-typed boolean expression.

Preconditions and invariants are essentially predicates that accept or reject method calls based on the object's state and the arguments of the call, much like SECRO's state validators. Preconditions must hold right before the call to the associated method, while invariants must hold after the associated method call (and possible concurrent calls) executed. Preconditions and invariants can access the object's current state using the `this` keyword. Invariants can also refer to the state of the object as it was before the call using the `old` keyword.

The use of preconditions and invariants is similar to ECROs but they are defined in EFx instead of in a first-order logic specification. For example, the below class C is marked as replicated and defines a number of fields $\overline{v}$, methods $\overline{M}$, preconditions $\overline{Pre}$, and invariants $\overline{Inv}$.

$$@\mathit{replicated}\ \texttt{class}\ C\,\langle\overline{X}\rangle\,(\overline{v} : \overline{T})\,\{\,\overline{M \mid Pre \mid Inv}\,\}$$

We use the notation Pre_M and Inv_M to refer to the precondition, respectively, the invariant of a method M.

Given the above class, EFx can derive a sequential implementation of the data type and a concurrency contract. The sequential implementation consists of a class containing only the fields and the methods:

$$L = \texttt{class}\ C\,\langle\overline{X}\rangle\,(\overline{v} : \overline{T})\,\{\,\overline{M}\,\}$$

The contract maps methods to their precondition and invariant:

$$\mathcal{C}(M) = \langle Pre_M, Inv_M \rangle$$

If a method does not have an associated precondition or invariant it is assumed to be true, i.e. `pre` $m\,\langle\overline{X}\rangle\,(\overline{x} : \overline{T})\,\{\,$`true`$\,\}$. Thus, RDTs in EFx consist of a sequential implementation L and a concurrency contract $\mathcal{C}$.

5.2.4 Functional Collections

EFx features built-in collections including tuples, sets, maps, vectors, and lists. Remarkably, these collections are completely analyzable and can be arbitrarily composed to build custom RDTs. All collections are immutable, "mutators" thus return an updated copy of the object.

Figure 5.3 provides an overview of the interface exposed by EFx's collections, which is heavily inspired by functional programming. We now discuss each data type in this collection in more detail.

Tuples group two values that can be accessed using the `fst` and `snd` fields.

Sets support the typical set operations and can be mapped over or filtered using user-provided functions. The `forall` and `exists` methods check if a given predicate holds for all (respectively for at least one) element of the set.

Maps associate keys to values. Programmers can add key-value pairs, remove keys, and fetch the value that is associated to a certain key. The `keys` (resp. `values`) method returns a set containing all keys (resp. values) contained by the map. The `bijective` method checks if there is a one-to-one correspondence between the keys and the values. Maps support many well-known functional operations; `zip` returns a map of tuples containing only the keys that are present in both maps and stores their values in a tuple; `combine` returns a map containing *all* entries from both maps, using a user-provided function `f` to combine values that are present in both maps.

Vectors represent a sequence of elements which are indexed from `0` to `size-1`. Elements can be written to a certain index which will overwrite

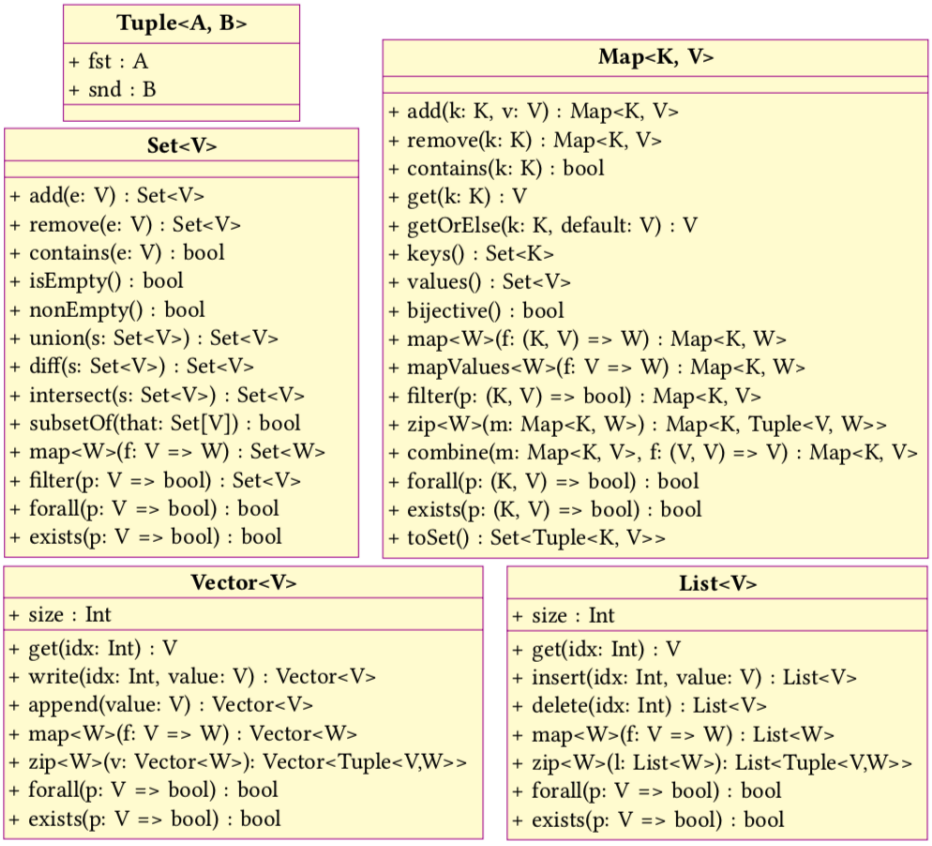


Figure 5.3: An overview of EFx's built-in functional collections.

the existing value at that index. One can append a value to the vector which will write that value at index `size`, thereby, making the vector grow. Like sets and maps, programmers can map functions over vectors, zip vectors, and check predicates for all or for one element of a vector.

Lists represent a sequence of elements in a linked list. Unlike vectors, `insert` does not overwrite the existing value at that index. Instead, the existing value at that index and all subsequent values are moved one position to the right. Elements can also be deleted from a list, making the list shrink.

5.3 Automated Analysis of EFx Programs

As explained in Section 5.2.1, EFx transpiles RDTs to SMT-LIB[4], a standardized language for SMT solvers, and leverages Ordana's analyses from Section 4.3 to derive additional information about the operations. It then uses this information to generate distributed specifications and synthesize correct ECRO data types.

The analyses performed by Ordana on the specification are a form of program verification as they verify commutativity and safety properties of the RDT's operations. For instance, Ordana's commutativity analysis analyzes pairs of operations to check if they commute or not (cf. Section 4.3.3). Given a pair of operations, the analysis tries to prove that all calls to those operations commute. If the proof is rejected, the SMT solver returns a concrete counterexample in which the operations do not commute.

Modern SMT solvers support various specialized theories (for bitvectors, arrays, etc.) and are very powerful if care is taken to encode programs efficiently using these theories. Thus, in order for Ordana's analyses to work properly, the generated specifications must be encoded efficiently. However, SMT-LIB is low-level and is not meant to be used directly by programmers to verify high-level programs. Therefore, we carefully designed EFx such that every language feature has an efficient SMT encoding; leaving out features that break automated verification. For example, EFx does not support traditional loop statements but instead provides higher-order operations (map, filter, etc.) on top of its functional collections. The resulting language is surprisingly expressive given its automated verification capabilities.

In the remainder of this section, we show how EFx compiles programs to a core of SMT-LIB. Afterward, we explain how EFx leverages a specialized theory of arrays to efficiently encode its functional collections. These encodings are key to our approach because they enable fully automated analyses of RDTs built atop EFx's functional collections. To exemplify the compilation rules we finish this section with a concrete example.

[4] `http://smtlib.cs.uiowa.edu/`

5.3.1 Core SMT

The semantics of EFx are defined using translation functions from EFx to Core SMT, a reduced version of SMT-LIB that suffices to analyze EFx programs. For brevity, we may refer to SMT-LIB as just SMT.

$$
\begin{array}{ll}
T ::= \mathsf{int} \mid \mathsf{string} \mid \mathsf{bool} & G ::= \mathsf{adt}\ A\langle \overline{X} \rangle \{ \overline{K(\overline{v} : \overline{T})} \} \\
\quad \mid \mathsf{Array}\langle \vec{T}, T \rangle \mid A\langle \overline{T} \rangle \mid S\langle \overline{T} \rangle & e ::= e[\vec{e}] \mid e[\vec{e}] := e \mid \lambda(\vec{x} : \vec{T}).e \\
C ::= \mathsf{const}\ x\ T & \quad \mid \forall(\vec{x} : \vec{T}).e \mid \exists(\vec{x} : \vec{T}).e \mid \ldots \\
D ::= \mathsf{sort}\ S\ i & R ::= \mathsf{assert}\ e \\
F ::= \mathsf{fun}\ f\langle \overline{X} \rangle(\overline{x} : \overline{T}) : T = e & H ::= \mathsf{check}()
\end{array}
$$

Figure 5.4: Core SMT syntax.

Figure 5.4 defines the syntax of Core SMT. The metavariable S ranges over user-declared sorts[5]; A ranges over names of Algebraic Data Types (ADTs); K ranges over ADT constructor names; X ranges over type variables; v ranges over field names; f ranges over function names; T ranges over types; x ranges over variable names; e ranges over expressions; and i ranges over integers. Valid types include integers, strings, booleans, arrays, ADTs $A\langle \overline{T} \rangle$, and user-declared sorts $S\langle \overline{T} \rangle$. Arrays are *total* and map values of the key types to a value of the element type. Arrays can be multidimensional and map several keys to a value.

Core SMT programs consist of one or more statements which can be the declaration of a constant or sort, assertions, the definition of a function or ADT, or a call to check. Constant declarations take a name and a type. Sort declarations take a name and a non-negative number i representing their arity, i.e. how many type parameters the sort takes. Declared constants and sorts are *uninterpreted* and the SMT solver is free to assign any valid interpretation. Assertions are boolean formulas that constrain the possible interpretations of the program, e.g. assert age >= 18.

Function definitions consist of a name f, optional type parameters $\overline{X}$, formal parameters $\overline{x} : \overline{T}$, a return type T, and a body containing an expression e. Valid expressions include array accesses $e[\,\vec{e}\,]$, array updates $e[\,\vec{e}\,] := e$, anonymous functions, quantified formulas, etc[6]. Updating an

[5]The literature on SMT solvers uses the term "sort" to refer to types and type constructors.

[6]The complete set of expressions is described in Appendix G.

array returns a modified copy of the array. It is important to note that arrays are total and that anonymous functions define an array from the argument types to the return type. For example, $\lambda(x : \mathsf{int}, y : \mathsf{int}).x + y$ defines an $\mathsf{Array}\langle\mathsf{int}, \mathsf{int}, \mathsf{int}\rangle$ that maps two integers to their sum. Since arrays are first-class values in SMT, it follows that lambdas are also first-class.

ADT definitions consist of a name A, optional type parameters $\overline{X}$, and one or more constructors. Every constructor has a name K and optionally defines fields with a name v and a type T. Constructors are invoked like regular functions and return an instance of the data type.

The decision procedure (check) checks the satisfiability of the SMT program. If the program's assertions are satisfiable, check returns a concrete model, i.e. an interpretation of the constants and sorts that satisfies the assertions. A property p can be proven by showing that the negation $\neg p$ is unsatisfiable, i.e. that no counterexample exists.

Note that our Core SMT language includes lambdas and polymorphic functions which are not part of SMT-LIB v2.6. Nevertheless, they are described in the preliminary proposal for SMT-LIB v3.0[7] and Z3 already supports lambdas. For the time being, EFx monomorphizes polymorphic functions when they are compiled to Core SMT. For example, given a polymorphic identity function `id<X> :: X -> X`, EFx creates a monomorphic version `id_int :: int -> int` when encountering a call to `id` with an integer argument.

5.3.2 Compiling EFx to Core SMT

Similar to Dafny (cf. Section 2.3.1), we describe the semantics of EFx by means of translation functions that compile EFx programs to Core SMT. Types are translated by the $[\![]\!]_t$ function:

$$
\begin{array}{llll}
[\![\texttt{bool}]\!]_t = \mathsf{bool} & [\![\texttt{int}]\!]_t = \mathsf{int} & [\![\texttt{string}]\!]_t = \mathsf{string} & \\
[\![C\langle\overline{\texttt{T}}\rangle]\!]_t = C\langle\overline{[\![\texttt{T}]\!]_t}\rangle & & [\![\vec{\overline{T}} \to P]\!]_t = \mathsf{Array}\langle\overline{[\![\vec{T}]\!]_t}, [\![P]\!]_t\rangle &
\end{array}
$$

Primitive types are translated to the corresponding primitive type in Core SMT. Class types keep the same type name and their type arguments are translated recursively $\overline{[\![\texttt{T}]\!]_t}$. Functions are encoded as arrays from the argument types to the return type. Trait types do not exist in the compiled

[7] `http://smtlib.cs.uiowa.edu/version3.shtml`

SMT program because traits are compiled away by EFx, i.e. only the types of the classes that implement the trait exist.

We now take a look at the translation function $def[\![]\!]$ which compiles classes. Classes are encoded as ADTs with one constructor and methods become regular functions:

$$
\begin{aligned}
&def[\![\overline{A}\,\texttt{class}\ C\,\langle\overline{X}\rangle\,(\overline{v}:\overline{T})\,\{\,\overline{M}\,\}\,\texttt{extends}\,I\langle\overline{P}\rangle]\!] \;=\\
&\quad \texttt{adt}\ C\langle\overline{X}\rangle\{\,K(\overline{v}:\overline{[\![T]\!]_t})\,\}\ ;\ \overline{method[\![C,\overline{X},M]\!]}\ ;\ \overline{method[\![C,\overline{X},M'[\overline{P}/\overline{Y}]]\!]}\\
&\qquad \text{where } K = \texttt{str\_concat}(C, \text{“}\_ctor\text{”})\\
&\qquad\ \text{and } I \text{ is defined as } \texttt{trait}\ I\,\langle\overline{Y}\rangle\,\{\,\overline{M'}\ ;\ \ldots\}\\
&method[\![C,\overline{X},\texttt{def}\ m\,\langle\overline{Y}\rangle\,(\overline{x}:\overline{T}):T_r = e]\!] \;=\\
&\quad \texttt{fun}\,f\langle\overline{X},\overline{Y}\rangle(this:C\langle\overline{X}\rangle,\overline{x}:\overline{[\![T]\!]_t}):[\![T_r]\!]_t = [\![e]\!]\\
&\qquad \text{where } f = \texttt{str\_concat}(C, \text{“}\_\text{”}, m)
\end{aligned}
$$

The ADT keeps the name of the class and its type parameters, and defines one constructor containing the class' fields. Since the name of the constructor must differ from the ADT's name, the compiler defines a unique name K which is the name of the class followed by "_ctor". The class' methods $\overline{M}$ are compiled to regular functions by the $method[\![]\!]$ function. Furthermore, the class inherits all concrete methods $\overline{M'}$ that are defined by its super trait and are not overridden by itself. This requires substituting the trait's type parameters $\overline{Y}$ by the concrete type arguments $\overline{P}$ provided by the class. As a result, traits do not exist in the resulting SMT program.

For every method, a function is created with a unique name f that is the name of the class followed by an underscore and the name of the method (this avoids name clashes between methods of different classes that have the same name). In the argument list, the body, and the return type of a method, programmers can refer to type parameters of the class or method. Therefore, the compiled SMT function takes both the class' type parameters $\overline{X}$ and the method's type parameters $\overline{Y}$. Without loss of generality we assume that a method's type parameters do not override the class' type parameters which can be achieved through α-conversion. The method's parameters become parameters of the function. In addition, the function takes an additional parameter `this` referring to the receiver of the method call which should be of the class' type. The types of the parameters and the return type are translated using function $[\![]\!]_t$. The body of the method must be a well-typed expression. Expressions are

compiled by the translation function $[\![]\!]$:

$$
\begin{array}{lcl}
[\![x]\!] & = & x \\
[\![e_1 \oplus e_2]\!] & = & [\![e_1]\!] \oplus [\![e_2]\!] \\
[\![e_1 \otimes e_2]\!] & = & [\![e_1]\!] \otimes [\![e_2]\!] \\
[\![!e]\!] & = & \neg [\![e]\!] \\
[\![\texttt{val}\ x : T = e_1\ \texttt{in}\ e_2]\!] & = & \mathsf{let}\, x = [\![e_1]\!]\, \mathsf{in}\, [\![e_2]\!] \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] & = & \mathsf{if}([\![e_1]\!], [\![e_2]\!], [\![e_3]\!]) \\
[\![(\vec{x} : \vec{T}) \Rightarrow e]\!] & = & \lambda(\vec{x} : \overline{[\![T]\!]_t}).[\![e]\!] \\
[\![e_1(\overline{e_2})]\!] & = & [\![e_1]\!][\ \overline{[\![e_2]\!]}\ ]
\end{array}
$$

Primitive values, variable references, and parameter references remain unchanged in Core SMT. The operands of binary operators (i.e. arithmetic $\oplus$ and boolean operators $\otimes$) are compiled recursively. A negated expression is compiled to the negation of the compiled expression. The definition of an immutable variable is translated to a let expression. Anonymous functions remain anonymous functions in Core SMT, the type of the parameters and the body are compiled recursively. Remember that anonymous functions in SMT define (multidimensional) arrays from one or more arguments to the function's return value. Hence, function calls are translated to array accesses.

$$
\begin{array}{lcl}
[\![\texttt{new}\ C\langle\overline{T}\rangle(\,\overline{e}\,)]\!] & = & C'\langle\overline{[\![T]\!]_t}\rangle(\overline{[\![e]\!]}) \\
\quad \text{where } C' = \texttt{str\_concat}(C, \text{“}\_ctor\text{”}) & & \\
[\![e.v]\!] & = & [\![e]\!].v \\
[\![e_1.m\ \langle\overline{T}\rangle\ (\,\overline{e}\,)]\!] & = & m'\langle\overline{[\![P]\!]_t}, \overline{[\![T]\!]_t}\rangle([\![e_1]\!], \overline{[\![e]\!]}) \\
\quad \text{where } \mathit{typeof}(e_1) = C\langle\overline{P}\rangle & & \\
\quad \text{and } m' = \texttt{str\_concat}(C, \text{“\_”}, m) \text{ and } \overline{P} \cap \overline{T} = \emptyset & &
\end{array}
$$

To instantiate a class or ADT, the compiler calls the data type's constructor function. For classes, the constructor's name is the name of the class followed by "_ctor". To access a field, the compiler translates the expression and accesses the field on the translated expression. To invoke a method m on an object e_1 the compiler calls the corresponding function m' which by convention is the name of the class followed by an underscore and the name of the method. Recall that the function takes both the class' type arguments $\overline{T}$ and the method's type arguments $\overline{P}$ as well as an additional argument e_1 which is the receiver of the call.

5.3.3 Encoding Functional Collections Efficiently in SMT

Unique to EFx is its efficient encoding of functional collections in SMT. Some Intermediate Verification Languages (IVLs) feature collections with rich APIs (e.g. Why3 [FP13]) but encode operations on these collections recursively. Unfortunately, traditional SMT solvers fail to verify recursive definitions automatically because they require inductive proofs, which is beyond the capabilities of most solvers.

However, many SMT solvers support specialized array theories. A key insight of our work consists of efficiently encoding the collections and their operations using the Combinatory Array Logic (CAL) [MB09] which is decidable. As a result, EFx can automatically analyze RDTs that are built by arbitrary compositions of functional collections. In the remainder of this section, we describe the encoding of the different functional collections in this array logic.

5.3.3.1 Set Encoding

Sets are encoded as arrays from the element type to a boolean type that indicates whether the element is in the set:

$$[\![\texttt{Set}\,\langle T\rangle]\!]_t = \mathsf{Array}\langle [\![T]\!]_t, \mathsf{bool}\rangle$$

An empty set corresponds to an array containing false for every element. We can create such an array by defining a lambda that ignores its argument and always returns false:

$$[\![\texttt{new Set}\,\langle T\rangle(\,)]\!] = \lambda(x : [\![T]\!]_t).\mathsf{false}$$

Operations on sets are compiled as follows:

$$\begin{aligned}
&[\![e_1.\mathit{add}(e_2)]\!] = [\![e_1]\!][\,[\![e_2]\!]\,] := \mathsf{true}\\
&[\![e_1.\mathit{remove}(e_2)]\!] = [\![e_1]\!][\,[\![e_2]\!]\,] := \mathsf{false}\\
&[\![e_1.\mathit{contains}(e_2)]\!] = [\![e_1]\!][\,[\![e_2]\!]\,]
\end{aligned}$$

An element e_2 is added to a set e_1 by setting the entry for e_2 in the array that results from transforming e_1 to true. Similarly, an element is removed by setting its entry in the array to false. An element is in the set if its entry is true.

$$[\![e_1.\mathit{filter}(e_2)]\!] = \lambda(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \wedge [\![e_2]\!][\ x\ ]$$
$$\text{where } \mathit{typeof}(e_1) = \texttt{Set}\langle T\rangle \text{ and } \mathit{typeof}(e_2) = T \to \texttt{bool}$$
$$[\![e_1.\mathit{map}(e_2)]\!] = \lambda(y : [\![P]\!]_t).\exists(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \wedge [\![e_2]\!][\ x\ ] = y$$
$$\text{where } \mathit{typeof}(e_1) = \texttt{Set}\langle T\rangle \text{ and } \mathit{typeof}(e_2) = T \to P$$

The `filter` method filters a set e_1 containing elements of type T in order to retain only the elements that fulfill a given predicate $e_2 : T \to \texttt{bool}$. Calls to `filter` are compiled to a lambda that defines a set (i.e. an array from elements to booleans) containing only the elements x that are in the original set e_1 (i.e. $[\![e_1]\!][\ x\ ]$) and fulfil predicate e_2 (i.e. $[\![e_2]\!][\ x\ ]$). Similarly, when calling `map` with a function $e_2 : T \to P$ on a set e_1 of Ts, the method yields a set of Ps. Calls to `map` are compiled to a lambda that defines a set containing elements y of type $[\![P]\!]_t$ such that an element x exists that is in the original set e_1 (i.e. $[\![e_1]\!][\ x\ ]$) and maps to y (i.e. $[\![e_2]\!][\ x\ ] = y$).

$$[\![e_1.\mathit{union}(e_2)]\!] = \lambda(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \vee [\![e_2]\!][\ x\ ]$$
$$\text{where } \mathit{typeof}(e_1) = \texttt{Set}\langle T\rangle \wedge \mathit{typeof}(e_2) = \texttt{Set}\langle T\rangle$$
$$[\![e_1.\mathit{intersect}(e_2)]\!] = \lambda(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \wedge [\![e_2]\!][\ x\ ]$$
$$\text{where } \mathit{typeof}(e_1) = \texttt{Set}\langle T\rangle \wedge \mathit{typeof}(e_2) = \texttt{Set}\langle T\rangle$$
$$[\![e_1.\mathit{diff}(e_2)]\!] = \lambda(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \wedge \neg[\![e_2]\!][\ x\ ]$$
$$\text{where } \mathit{typeof}(e_1) = \texttt{Set}\langle T\rangle \wedge \mathit{typeof}(e_2) = \texttt{Set}\langle T\rangle$$

The `union` method computes the set union of two sets e_1 and e_2. To this end, calls to `union` are compiled to a lambda that defines an array of elements x of type $[\![T]\!]_t$ containing only elements that are in at least one of the two sets, i.e. $[\![e_1]\!][\ x\ ] \vee [\![e_2]\!][\ x\ ]$. Similarly, calls to the `intersect` method are compiled to a lambda which defines an array containing only elements that are in both sets, i.e. $[\![e_1]\!][\ x\ ] \wedge [\![e_2]\!][\ x\ ]$. For set difference, calls to `diff` are compiled to a lambda that defines an array containing only elements that are in e_1 and not in e_2.

$$[\![e_1.\mathit{subsetOf}(e_2)]\!] = \forall(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \implies [\![e_2]\!][\ x\ ]$$
$$\text{where } \mathit{typeof}(e_1) = \texttt{Set}\langle T\rangle \wedge \mathit{typeof}(e_2) = \texttt{Set}\langle T\rangle$$
$$[\![e.\mathit{nonEmpty}()]\!] = \exists(x : [\![T]\!]_t).[\![e]\!][\ x\ ] \quad \text{where } \mathit{typeof}(e) = \texttt{Set}\langle T\rangle$$
$$[\![e.\mathit{isEmpty}()]\!] = \forall(x : [\![T]\!]_t).\neg[\![e]\!][\ x\ ] \quad \text{where } \mathit{typeof}(e) = \texttt{Set}\langle T\rangle$$

The `subsetOf` method checks if e_1 is a subset of e_2. To this end, it checks that all elements from e_1 are also in e_2. The `nonEmpty` method returns true for a set e if at least one element x exists that is in the set, i.e. $[\![e]\!][\ x\ ]$.

Similarly, the `isEmpty` method returns true for a set e if every element x is not in the set.

$$\begin{aligned}
&[\![e.forall(e_p)]\!] = \forall(x : [\![T]\!]_t).[\![e]\!][\ x\ ] \implies [\![e_p]\!][\ x\ ] \\
&\quad \text{where } typeof(e) = \texttt{Set}\langle T\rangle \text{ and } typeof(e_p) = T \to \texttt{bool} \\
&[\![e.exists(e_p)]\!] = \exists(x : [\![T]\!]_t).[\![e]\!][\ x\ ] \wedge [\![e_p]\!][\ x\ ] \\
&\quad \text{where } typeof(e) = \texttt{Set}\langle T\rangle \text{ and } typeof(e_p) = T \to \texttt{bool}
\end{aligned}$$

When calling `forall` with a predicate $e_p : T \to \texttt{bool}$ on a set e of type T, the method checks that for every element x that is in the set, the predicate holds, i.e. $[\![e]\!][\ x\ ] \implies [\![e_p]\!][\ x\ ]$. Similarly, the `exists` method checks that at least one element x exists that is in the set and fulfills the predicate, i.e. $[\![e]\!][\ x\ ] \wedge [\![e_p]\!][\ x\ ]$.

5.3.3.2 Map Encoding

Maps are encoded as arrays from the key type to an optional value:

$$[\![\texttt{Map}\,\langle T, P\rangle]\!]_t = \mathsf{Array}\langle [\![T]\!]_t, \mathsf{Option}\langle [\![P]\!]_t\rangle\rangle$$

Optional values indicate the presence or absence of a value for a certain key. The option type is defined as an ADT with two constructors: $\mathsf{Some}(value)$ which holds a value and $\mathsf{None}()$ indicating the absence of a value. An empty map corresponds to an array containing $\mathsf{None}()$ for every key and is created by a lambda that returns $\mathsf{None}()$ for every key:

$$[\![\texttt{new Map}\,\langle T, P\rangle(\ )]\!] = \lambda(x : [\![T]\!]_t).\mathsf{None}\langle [\![P]\!]_t\rangle()$$

Operations on maps are compiled as follows:

$$\begin{aligned}
&[\![e_m.add(e_k, e_v)]\!] = [\![e_m]\!][\ [\![e_k]\!]\ ] := \mathsf{Some}([\![e_v]\!]) \\
&[\![e_m.remove(e_k)]\!] = [\![e_m]\!][\ [\![e_k]\!]\ ] := \mathsf{None}\langle [\![V]\!]_t\rangle() \\
&[\![e_m.contains(e_k)]\!] = [\![e_m]\!][\ [\![e_k]\!]\ ] \neq \mathsf{None}\langle [\![V]\!]_t\rangle() \\
&[\![e_m.get(e_k)]\!] = [\![e_m]\!][\ [\![e_k]\!]\ ].value \\
&[\![e_m.getOrElse(e_k, e_v)]\!] = \\
&\quad \mathsf{if}([\![e_m]\!][\ [\![e_k]\!]\ ] = \mathsf{None}\langle [\![V]\!]_t\rangle(),\ [\![e_v]\!],\ [\![e_m]\!][\ [\![e_k]\!]\ ].value)
\end{aligned}$$

A key-value pair $e_k \mapsto e_v$ is added to a map e_m by updating the entry for the compiled key $[\![e_k]\!]$ in the compiled array $[\![e_m]\!]$ with the compiled value, $\mathsf{Some}([\![e_v]\!])$. A key e_k is removed from a map e_m by updating the corresponding entry to $\mathsf{None}\langle [\![V]\!]_t\rangle()$, thereby indicating the absence of a value. Note that `None` is polymorphic but the type parameter cannot be

inferred from the arguments; therefore it is passed explicitly. A key e_k is contained by a map e_m if the value that is associated to the key is not $\mathsf{None}\langle[\![V]\!]_t\rangle()$. The `get` method fetches the value that is associated to a key e_k in a map e_m. To this end, the compiled key $[\![e_k]\!]$ is accessed in the compiled map $[\![e_m]\!]$ and the value it holds is then fetched by accessing the `value` field of the Some constructor. Even though the entry that is read from the array is an option type (i.e. a None or a Some) we can access the `value` field because the interpretation of `value` is underspecified in SMT. If the entry is a None, the SMT solver can assign any interpretation to the `value` field. Hence, the `get` method on maps should only be called if the key is known to be present in the map, e.g. after calling `contains`. EFx also features a safe variant, `getOrElse`, that takes a default value and returns that default value it if the key is not present in the map.

We now show how to compile a selection of advanced map operations:

$$
\begin{aligned}
&[\![e_m.keys()]\!] = \lambda(x : [\![K]\!]_t).[\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \\
&\quad \text{where } \mathit{typeof}(e_m) = \mathtt{Map}\langle K, V\rangle \\
&[\![e_m.values()]\!] = \lambda(x : [\![V]\!]_t).\exists(k : [\![K]\!]_t).[\![e_m]\!][\ k\ ] = \mathsf{Some}(x) \\
&\quad \text{where } \mathit{typeof}(e_m) = \mathtt{Map}\langle K, V\rangle
\end{aligned}
$$

The `keys` method returns a set containing only the keys that are present in the map. Calls to `keys` on a map e_m of type $\mathtt{Map}\,\langle \mathtt{K}, \mathtt{V}\rangle$ are compiled to a lambda that defines a set of keys x of the compiled key type $[\![K]\!]_t$ such that a key is present in the set iff it is present in the compiled map, i.e. $[\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle()$. The `values` method returns a set containing all values of the map e_m. To this end, it defines an array containing all values for which at least one key exists that maps to that value in e_m.

$$
\begin{aligned}
&[\![e_m.map(e_f)]\!] = \lambda(x : [\![K]\!]_t).\ \mathsf{if}([\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle(), \\
&\qquad\qquad\qquad\qquad \mathsf{Some}([\![e_f]\!][\ x, [\![e_m]\!][\ x\ ].\mathit{value}\ ]), \\
&\qquad\qquad\qquad\qquad \mathsf{None}\langle[\![W]\!]_t\rangle()) \\
&\quad \text{where } \mathit{typeof}(e_m) = \mathtt{Map}\langle K, V\rangle \text{ and } \mathit{typeof}(e_f) = (K, V) \to W \\
&[\![e_m.mapValues(e_f)]\!] = \lambda(x : [\![K]\!]_t).\ \mathsf{if}([\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle(), \\
&\qquad\qquad\qquad\qquad \mathsf{Some}([\![e_f]\!][\ [\![e_m]\!][\ x\ ].\mathit{value}\ ]), \\
&\qquad\qquad\qquad\qquad \mathsf{None}\langle[\![W]\!]_t\rangle()) \\
&\quad \text{where } \mathit{typeof}(e_m) = \mathtt{Map}\langle K, V\rangle \text{ and } \mathit{typeof}(e_f) = V \to W
\end{aligned}
$$

When calling the `map` method with a function e_f of type $(K, V) \to W$ on a map e_m of type $\mathtt{Map}\langle K, V\rangle$, it returns an updated map of type $\mathtt{Map}\langle K, W\rangle$ whose values are the result of applying e_f on the key-value pairs. The

`map` method is encoded as a lambda that defines an array containing only the keys that are present in the compiled map $[\![e_m]\!]$ and whose values are the result of applying e_f on the key and its associated value, i.e. $\mathsf{Some}([\![e_f]\!][\ x, [\![e_m]\!][\ x\ ].value\ ])$. The `mapValues` method is similar except that it applies the provided function only on the value. The remaining operations on maps are encoded similarly and are defined in Appendix H.

5.3.3.3 Vectors and Lists

Sets and maps are very useful to build new data structures in EFx without having to encode them manually in SMT. For example, vectors and lists are implemented on top of maps in EFx itself.

Listing 5.2 shows an excerpt from the implementation of vectors. The implementation of lists follows a similar strategy. Internally, every vector keeps a dictionary, called `positions`, that maps indices between 0 and $size - 1$ to their value in the vector. The vector data type then defines a traditional interface - containing methods such as e.g. `get`, `write`, `map`, etc. - on top of the underlying map.

The presented implementation of vectors and lists on top of maps is only used for analyses in SMT. When compiling to languages such as Scala or JavaScript, EFx leverages the target language's built-in vector and list data structures.

5.3.4 Compilation Example

We now provide a concrete example of a polymorphic set implemented in EFx and its compiled code in Core SMT. Figure 5.5a shows the `MSet` class

Listing 5.2: Internal vector implementation in EFx.

```
class Vector[V](size: Int = 0,
                positions: Map[Int, V] = new Map[Int, V]()) {
  def get(idx: Int) = this.positions.get(idx)
  def write(idx: Int, elem: V) =
    new Vector(this.size, this.positions.add(idx, elem))
  def map[W](f: V => W) =
    new Vector(this.size, this.positions.mapValues(f))
  // ...
}
```

```
class MSet[V](set: Set[V]) {
  def map[W](f: V => W) =
    new MSet(this.set.map(f))
}
```

(a) A polymorphic class in EFx.

$$\begin{aligned}&\mathsf{adt}\ MSet\langle V\rangle\{\ MSet\_ctor(set : \mathsf{Array}\langle V, \mathsf{bool}\rangle)\ \}\\&\mathsf{fun}\ MSet\_map\langle V, W\rangle(this : MSet\langle V\rangle,\\&\qquad\qquad f : \mathsf{Array}\langle V, W\rangle) : MSet\langle W\rangle =\\&\quad MSet\_ctor(\\&\qquad \lambda(y : W).\exists(x : V).this.set[\ x\ ] \wedge f[\ x\ ] = y)\end{aligned}$$

(b) Compiled Core SMT code.

Figure 5.5: A polymorphic EFx class and its compiled Core SMT code.

in EFx which defines one type parameter `V` corresponding to the type of elements it holds. It also contains one field `set` of type `Set[V]` and defines a polymorphic method `map` that takes a function `f` of type `V => W` and returns a new `MSet` that results from applying `f` on every element.

Figure 5.5b shows the compiled Core SMT code for the `MSet` class. The compiled code defines an ADT `MSet` with one type parameter `V` and one constructor `MSet_ctor`. The constructor defines one field `set` of sort $\mathsf{Array}\langle V, \mathsf{bool}\rangle$ which is the compiled sort for sets. In addition, a polymorphic `MSet_map` function is defined which takes two type parameters `V` and `W` which correspond to `MSet`'s type parameter and `map`'s type parameter respectively. The function takes two arguments, the object that receives the call and the function `f`. The function's body calls the `MSet` constructor with the result of mapping `f` over the set.

5.4 Synthesizing ECROs from Contracts

EFx allows programmers to build custom RDTs by augmenting sequential data types with concurrency contracts that associate preconditions and invariants to the operations. We now explain how EFx compiles sequential data types and their concurrency contract to ECROs.

Recall from Section 5.2.1 that EFx features compiler plugins that transpile EFx code to other languages. Every plugin implements a translation function $[\![]\!]_{\mathcal{L}}$ that compiles EFx code to the target language $\mathcal{L}$. The compiler plugin for SMT-LIB is denoted $[\![]\!]_s$ and implements the translation functions defined in Section 5.3.

Algorithm 5 shows how to synthesize an ECRO from a sequential data type (i.e. a class $L =$ `class` $C\,\langle\overline{X}\rangle\,(\overline{v} : \overline{T})\,\{\,\overline{M}\,\}$) and its concurrency contract (i.e. a mapping from the class' methods to their preconditions and

Algorithm 5 Synthesizing ECROs from sequential data types and their concurrency contract.

function SYNTHESIZE(L, $\mathcal{C}$, $\mathcal{L}$)
  $\mathcal{I}_\mathcal{L} \leftarrow [\![L]\!]_\mathcal{L}$ ▷ transpile class L to $\mathcal{L}$ using compiler plugin
  $\mathcal{I}_s \leftarrow [\![L]\!]_s$ ▷ transpile class L to SMT
  $\langle sorts, \mathcal{S}\rangle \leftarrow$ makeSpec(L, $\mathcal{C}$) ▷ make an SMT specification
  $F \leftarrow$ analyze($sorts$, $\mathcal{I}_s$, $\mathcal{S}$) ▷ apply ECRO analyses
  return $\langle \mathcal{I}_\mathcal{L}, F\rangle$
function COMPILEPRE(Pre, P)
  $\texttt{pre}\, m\, \langle \overline{Y}\rangle\, (\overline{x} : \overline{T})\, \{\, e\,\} \leftarrow Pre$ ▷ deconstruct Pre
  return $\lambda(this : [\![P]\!]_s, \overline{x} : \overline{[\![T]\!]_s}).[\![e]\!]_s$
function COMPILEMETHOD(M, P)
  $\texttt{def}\ m\, \langle \overline{Y}\rangle\, (\overline{x} : \overline{T}) : T = e \leftarrow M$ ▷ deconstruct M
  return $\lambda(this : [\![P]\!]_s, \overline{x} : \overline{[\![T]\!]_s}).[\![e]\!]_s$
function COMPILEINV(Inv, P)
  $\texttt{inv}\, m\, \langle \overline{Y}\rangle\, (\overline{x} : \overline{T})\, \{\, e\,\} \leftarrow Inv$ ▷ deconstruct Inv
  return $\lambda(old : [\![P]\!]_s, this : [\![P]\!]_s, \overline{x} : \overline{[\![T]\!]_s}).[\![e]\!]_s$
function MAKESPEC(L, $\mathcal{C}$)
  $\mathcal{S} \leftarrow \{\}$ ▷ define an empty spec
  $sorts \leftarrow \emptyset$ ▷ accumulator for uninterpreted sorts
  $\texttt{class}\ C\, \langle \overline{X}\rangle\, (\overline{v} : \overline{T})\, \{\, \overline{M}\,\} \leftarrow L$ ▷ deconstruct L
  for $T_x \in \overline{X}$ **do**
    $sorts \leftarrow sorts \cup \{\, \mathsf{sort}\, [\![T_x]\!]_s\, 0\,\}$
  for $\texttt{def}\ m\, \langle \overline{Y}\rangle\, (\overline{x} : \overline{T}) : T = e \in \overline{M}$ **do**
    $M \leftarrow \texttt{def}\ m\, \langle \overline{Y}\rangle\, (\overline{x} : \overline{T}) : T = e$
    $\langle Pre, Inv\rangle \leftarrow \mathcal{C}(M)$ ▷ lookup in contract
    for $T_y \in \overline{Y}$ **do**
      $sorts \leftarrow sorts \cup \{\, \mathsf{sort}\, [\![T_y]\!]_s\, 0\,\}$
      $pre \leftarrow$ compilePre(Pre, $C\langle\overline{X}\rangle$)
      $post \leftarrow$ compileMethod(M, $C\langle\overline{X}\rangle$)
      $inv \leftarrow$ compileInv(Inv, $C\langle\overline{X}\rangle$)
      $\mathcal{S} \leftarrow \mathcal{S} + m \rightarrow \langle pre, post, inv\rangle$ ▷ store in spec
  return $\langle sorts, \mathcal{S}\rangle$

invariants, $\mathcal{C}(M) = \langle Pre_M, Inv_M \rangle$). The *synthesize* function compiles the sequential data type L and its concurrency contract $\mathcal{C}$ to an ECRO in some target language $\mathcal{L}$. To this end, EFx first transpiles the sequential data type L to an equivalent data type $\mathcal{I}_\mathcal{L}$ in the target language (Line 2). Similarly, it transpiles the data type to SMT (Line 3). Then, EFx derives an SMT specification $\mathcal{S}$ from the data type L and its concurrency contract $\mathcal{C}$ (Line 4). To this end, it first declares uninterpreted sorts for the class' type parameters (Line 21). For each method, it fetches the method's precondition and invariant[8] (Line 24) and declares uninterpreted sorts for the method's type parameters (Line 26). Then, it compiles the precondition, the method itself, and the invariant to SMT and stores them in the specification $\mathcal{S}$ (Line 30). Note that the postcondition is automatically compiled from the actual implementation of the method, whereas ECROs required programmers to manually define the effects of operations using first-order logic postconditions. Finally, on Line 5, the compiled SMT data type, its SMT specification, and the uninterpreted sorts are analyzed using the ECRO analyses from Chapter 4. The analysis result, F, together with the compiled data type $\mathcal{I}_\mathcal{L}$, form the synthesized ECRO.

Currently, our prototype implementation of EFx compiles RDTs written in EFx to ECROs in Scala. This could be extended to any language for which EFx has a compiler plugin and that features an implementation of the ECRO protocol. For example, EFx features a compiler plugin for JavaScript but we do not yet have a JavaScript implementation of ECROs.

5.5 Qualitative Evaluation

We now present a qualitative evaluation of our approach to assess whether EFx simplifies the development of RDTs in distributed systems. Section 5.5.1 presents a broad portfolio of RDTs implemented in EFx. Section 5.5.2 goes into more detail about the implementation of application-specific RDTs based on real-world use cases. Then, Section 5.5.3 focuses on the implementation of a complete distributed voting game inspired by modern TV shows. Finally, Section 5.5.4 compares our portfolio of RDTs written in EFx against their original ECRO counterpart in order to assess EFx's improvement compared to the traditional ECRO approach.

[8] If the method has no associated precondition or invariant it defaults to a function that always returns `true`.

5.5.1 Portfolio of Replicated Data Types

Data Type	LoC	C	M	Description and distributed semantics
Counter	6	1	2	Supports increments and decrements.
EW-Flag	13	1	2	Flag that can be enabled and disabled. Enable wins over concurrent disable operations.
DW-Flag	13	1	2	Similar to EW-Flag but guarantees disable-wins semantics.
AW-Set	12	1	2	Set providing add-wins semantics for concurrent adds and removes of the same element.
RW-Set	12	1	2	Set providing remove-wins semantics.
LWW-Set	11	1	2	Set providing last-writer-wins semantics.
LWW-Array	21	1	1	Array providing last-writer-wins semantics for concurrent writes on the same index.
Sync-Array	24	1	1	Array with coordinated writes (locks index before writing).
AW-Map	16	1	2	Map with add-wins semantics for concurrent adds and removes of the same key, and last-writer-wins semantics for concurrent adds of the same key.
RW-Map	16	1	2	Similar to AW-Map but remove-wins semantics for concurrent adds and removes of the same key.
Stack	14	1	2	Stack allowing push, pop, and top operations. Push operations execute optimistically and are totally ordered. Pop operations are coordinated in order not to pop more elements than there are on the stack.
Queue	12	1	2	Enqueue operations run optimistically and are totally ordered. Dequeue operations are coordinated to avoid dequeueing more elements than there are in the queue.
VotingGame	53	3	2	A distributed voting game inspired by contemporary TV-shows [Cet+14].
SmallBank	90	2	4	Banking application corresponding to the SmallBank benchmark [Alo+08].
RUBiS	87	2	6	Auction system similar to the RUBiS benchmark [EJ09].
Airline	285	9	9	An airline reservation system inspired by Acme Air [TS].

Table 5.1: Portfolio of RDTs implemented in EFx together with a description and code metrics. The C column is the number of classes, the M column the number of mutators exposed by the RDT.

To assess the applicability of our approach we implemented an extensive portfolio of RDTs using concurrency contracts in EFx. We will later compare them to the original ECRO portfolio in Section 5.5.4. Table 5.1 provides an overview of all RDTs included in our portfolio, accompanied by some code metrics and a brief description of the data types' semantics. Our portfolio includes all RDTs from the original ECRO approach, except

lists due to their recursive nature[9]. In addition, the portfolio also contains other RDTs such as a Last-Writer-Wins Set, two variations on arrays, and application-specific RDTs for a distributed voting game, the SmallBank application, and an airline reservation system. In the remainder of this section we elaborate on the set and array RDTs. Next section focuses on two application-specific RDTs, namely, the SmallBank application and the airline reservation system.

Sets. As explained in Section 4.2.2, concurrent adds and removes on a replicated set conflict if they try to add and remove the same element because the operations do not commute. The add-wins set solves such conflicts by letting adds win over concurrent removes of the same element. Similarly, the remove-wins set lets removes win over concurrent adds. The Last-Writer-Wins (LWW) set solves conflicts differently as it arbitrarily but deterministically lets one operation win over the other based on the unique IDs of the operations. As a result, sometimes adds may win and sometimes removes may win depending on the ID of the operations.

Arrays. The `LWW-Array` and `Sync-Array` are initialized with a fixed length and an initial value. Programmers can write a value at a certain index in the array, which overwrites the previously stored value at that index. Concurrently writing different values to the same index leads to a conflict because the operations do not commute. Therefore, the `LWW-Array` deterministically picks one write over the other, whereas, the `Sync-Array` locks the index before writing which avoids the conflict.

5.5.2 Application-Specific RDTs

We now move our attention to two application-specific RDTs, one for a banking application and one for an airline reservation system. We implemented these RDTs in EFx and tailored them to the needs of the applications.

SmallBank. The SmallBank RDT implements the operations described by the SmallBank benchmark [Alo+08]. Customers have a checking and a savings account and can fetch their balance, deposit money on their checking account, deposit or withdraw money from their savings account, move all funds to another customer, and write a cheque to another account. If the customer's balance is less than the amount of the cheque,

[9]EFx cannot analyze RDTs with recursive operations because those require inductive reasoning which is hard to automate with SMT solving.

the customer is charged an extra fee for the overdraft. All operations run without coordination, except withdrawals on the same account because customers may not withdraw more money than they have.

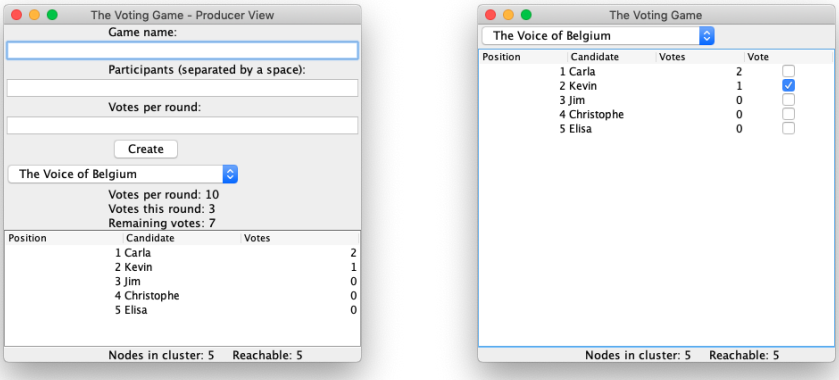
Airline. The airline reservation system, inspired by Acme Air [TS], keeps track of users and flights. Users can register, update their profile, deposit money, withdraw money, and book flights. Flights have a limited capacity but overbookings are allowed. New flights can be added to the system and existing flights can be modified. When users book a seat on a flight it is reserved, whereafter, the user proceeds to the payment. Once the booking is paid the seat is confirmed and can no longer be canceled (not even by a concurrent cancel operation). The payment system ensures that the user's account is debited exactly once. If the user does not pay within a certain time frame, the booking is canceled. While a user can book several flights concurrently, the payments are coordinated to avoid overdrafts.

5.5.3 Application Case: A Distributed Voting Game

We now report on the design and implementation of a full-fledged distributed voting game inspired by contemporary TV-shows [Cet+14]. The game features a number of candidates performing live acts (e.g. singing, dancing, etc.). Viewers at home can vote for their favorite candidate but can cast only a single vote. Periodically the candidate with the least amount of votes is eliminated. The viewers that voted on that candidate regain the right to vote. The game continues until a single candidate remains, which is the winner.

We build a variation of this game that runs atop a weakly consistent distributed system. Figure 5.6 shows the two main user interfaces of the application. One node acts as the producer of the tv show (Fig. 5.6a), while the remaining nodes are viewers (Fig. 5.6b). The producer can start new games and decides after how many votes candidates are eliminated. Viewers can participate in games and vote on candidates even if the producer is temporarily unreachable.

The application is implemented in Scala and consists of four main parts: a user interface, the game logic, a replicated game object, and a distributed system. Figure 5.7 provides an overview of these parts in terms



(a) User interface of the producer. (b) User interface of the viewer.

Figure 5.6: A distributed voting game inspired by contemporary tv-shows.

of LoC. The User Interface (UI) is built on top of the Swing UI library[10] and accounts for the biggest part of the application. The game logic mainly consists of the implementation of the producer and the viewers and is concerned with creating and discovering new games, casting votes and reacting to incoming votes, eliminating candidates, etc. The distributed part of the game uses Squirrel [DG19], our distributed key-value store, in order to setup a cluster of machines and replicate objects across the cluster. Finally, we implemented a custom Game RDT in EFx, compiled it to an ECRO in Scala, and deployed the resulting ECRO on top of our distributed system. Every node has a local replica of the game on which it can apply operations (i.e. the Game RDT is highly available). The state of the replica is visualized in the UI and replicas eventually converge when all updates are propagated.

We now briefly discuss the implementation of the Game RDT in EFx, shown in Listing 5.3. The `Game` class defines two fields: `votes` which maps candidates to a set of viewers that voted on them, and `voted` which is a set containing all the viewers that already voted. The game's main operations are `cast` and `eliminate`. The `cast` method registers a vote from a viewer `v` on a candidate `c`. If the viewer did not yet vote, they are added to the set of viewers that voted on the candidate (Line 10) and to the set

[10] `https://index.scala-lang.org/scala/scala-swing`

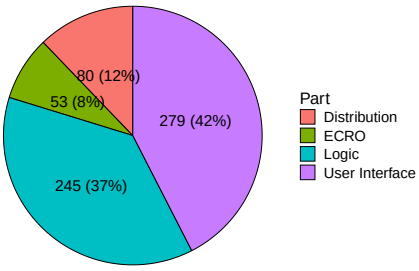


Figure 5.7: Overview of the distributed voting game in terms of LoC.

of viewers that already voted (Line 11). The `eliminate` method is called by the producer to eliminate a candidate. The candidate is removed from the map of votes (Line 16) and the viewers that voted on the candidate are removed from the `voted` set such that they regain the right to vote (Line 16).

As found by ECRO's concurrent commutativity analysis (cf. Section 4.3.3), concurrent `cast` and `eliminate` methods do not commute when the candidate that is eliminated is the same as the one on which a vote is cast. For example, a candidate c may have 5 votes; then, a user votes on c while concurrently c is eliminated. If replicas first cast the vote and then eliminate c, c is no longer part of the game. However, if replicas first eliminate c and then cast the vote, c reappears in the game. The situation where c reappears is undesirable for this application, therefore, the `Game` class adds an invariant on `eliminate` which states that once `eliminate` and all concurrent operations are executed, the candidate may not be part of the game. This ensures custom "eliminate-wins" semantics.

Interestingly, ECRO's safety analysis (cf. Section 4.3.5) detects the aforementioned anomaly and finds a coordination-free solution that consists of reordering concurrent `cast` and `eliminate` operations such that replicas always first cast the votes on a candidate before eliminating that candidate.

Based on the implemented `Game` RDT, EFx automatically synthesizes a corresponding ECRO. We integrated the ECRO in our voting application in Scala. When a producer starts a new game, a `Game` RDT is created and stored in Squirrel. Squirrel automatically replicates the object across all machines of the cluster. When the viewers discover this new `Game` object,

Listing 5.3: Excerpt from the replicated Game data type in EFx.

```
@replicated
class Game(votes: Map[Candidate, Set[Viewer]],
           voted: Set[Viewer]) {
  def getVotesFor(c: Candidate) =
    this.votes.getOrElse(c, new Set[Viewer]())
  def cast(v: Viewer, c: Candidate) = {
    if (this.viewerAlreadyVoted(v))
      this // ignore the vote
    else {
      val newVotes = this.getVotesFor(c).add(v)
      new Game(this.votes.add(c, newVotes), this.voted.add(v))
    }
  }
  def eliminate(c: Candidate) = {
    val viewers = this.getVotesFor(c)
    new Game(this.votes.remove(c), this.voted.diff(viewers))
  }
  inv eliminate(c: Candidate) {
    !this.votes.contains(c)
  }
  // ...
}
```

they acquire a local replica and can start voting on candidates. The votes are automatically propagated to all replicas and the replicas will converge to the same state when the updates are fully propagated across the cluster.

5.5.4 Comparison to the Original ECRO Approach

EFx was designed to simplify the development of RDTs by leveraging the ECRO protocol and combining it with an automated analysis of the RDT's implementation. To assess EFx's improvements, we compare the implementations of the RDTs that are shared between the original ECRO portfolio (cf. Table 4.1) and EFx's portfolio (cf. Table 5.1). We only focus on the programming efforts needed to implement these RDTs because the synthesized ECROs have the same performance as the original approach.

We compare the implementation of RDTs in EFx against their original ECRO implementation in terms of absolute code size and the distribution of code across components[11]. This is visualized by the stacked bar charts

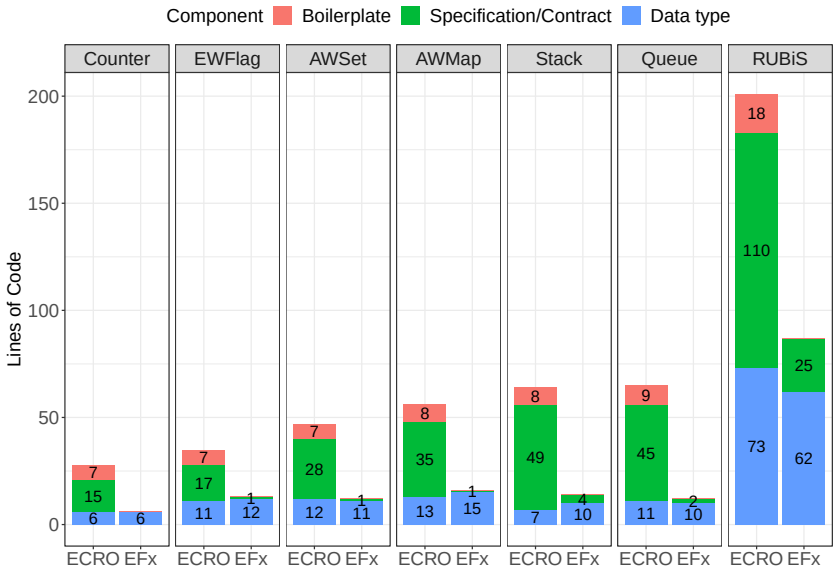
[11] To improve the readability of the plots we only plot one variant of each RDT because variants of the same RDT have similar code sizes.

in Figs. 5.8a and 5.8b. The code is categorized into three components: the data type, its specification, and boilerplate.

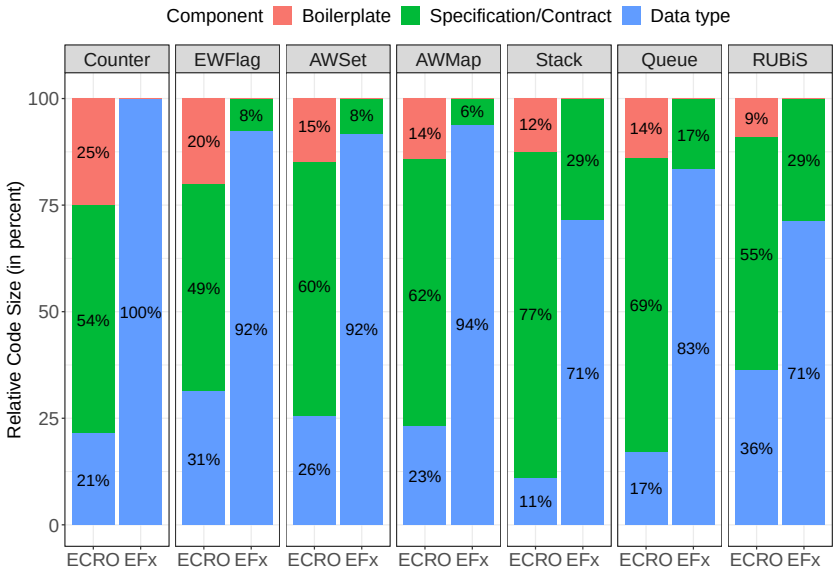
In the ECRO approach, the implementation is dominated by the RDT's specification (often 50% or more!) and the data type itself only forms a small part (mostly around 20% to 30%) of the overall implementation. A considerable part of the implementation consists of boilerplate code. In contrast, the EFx implementations are much smaller and mainly consist of the implementation of the data type itself. EFx and ECROs require approximately the same number of LoC for the implementation of the data types. However, EFx's concurrency contracts are much smaller than the ECRO specifications because programmers do not need to define postconditions. Moreover, boilerplate code is completely compiled away by EFx. Based on these observations, we conclude that EFx allows programmers to focus on the data type logic instead of the specification.

We now compare the ECRO specifications and their equivalent EFx contracts in more detail. Figures 5.9a and 5.9b depict the specifications and contracts and split them into four distinct parts: First-Order Logic (FOL) definitions, preconditions, postconditions, and invariants. The ECRO specifications are dominated by the definition of FOL relations which are needed to define the operations' preconditions, postconditions, and invariants. These relations are not needed in EFx because the contracts are integrated into the language; hence, there is no need for a separate specification language. Another significant part of the ECRO specifications consists of postconditions that must be defined for every update operation. In contrast, EFx does not require postconditions because they are automatically inferred from the operations which is possible since the operations are assumed to be correct. Hence, EFx contracts exclusively consist of RDT-specific preconditions and invariants while in the ECRO implementations these form only a small part of the specification.

Based on these observations, we believe that EFx simplifies the implementation of RDTs when compared to the original ECRO approach because the concurrency contracts expressed in EFx are considerably smaller than the ECRO specifications and let programmers focus on RDT-specific code instead of developing a complete FOL specification. To further validate this claim, Section 5.5.4.1 compares the implementation of an RDT for the RUBiS auction system (cf. Section 4.2.3) in EFx against an equivalent implementation with ECROs in Scala.

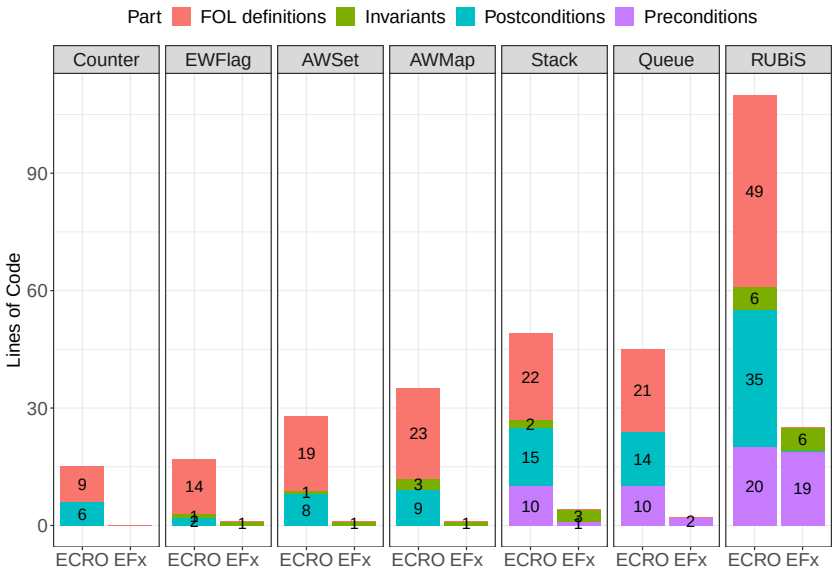


(a) Absolute code size of the RDT implementations.

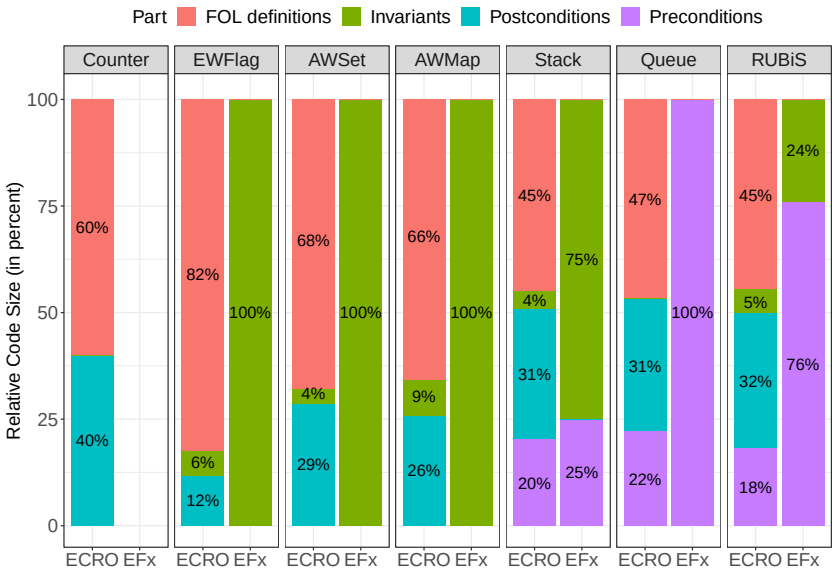


(b) Distribution of code across RDT components.

Figure 5.8: Comparison of RDTs implemented in EFx against ECROs.



(a) Absolute code size of the different parts of the specifications.



(b) Distribution of code across the different parts of the specifications.

Figure 5.9: Comparison of RDT specifications implemented in EFx against ECROs.

5.5.4.1 RUBiS Auction System

To further investigate the differences between EFx and ECROs, we now implement the RUBiS auction system in EFx and compare it to its original ECRO implementation. We chose the RUBiS application because it is a real-world application involving several invariants, it requires an RDT that is tailored to the needs of the application, and it has previously been adapted for geo-distributed systems [LPR18].

Recall from Section 4.2.3 that the RUBiS auction system allows users to create auctions, bid on auctions, and close auctions. Usernames should be unique, and every bid must be linked to an existing user (an invariant known as "referential integrity").

Listing 5.4 shows an excerpt from the implementation of a custom RDT for RUBiS in EFx. The code snippet depicts the implementation of the `registerUser` and `placeBid` operations. Similarly to the original ECRO implementation discussed in Section 4.2.3, we associate a precondition to each operation. The precondition of `registerUser` checks that the user does not yet exist (Line 4). If the precondition holds, the `registerUser` operation adds the user to the set of all users (Line 6). The precondition of `placeBid` requires the user to exist, the auction to exist, and the bid to be bigger than zero (Line 9). If the precondition holds, the `placeBid` operation fetches the list of bids for the given auction and appends the new bid to the list (Lines 11-16).

Listing 5.5 shows the implementation of the same RDT but implemented with ECROs. The RDT is split in two parts, a class named `Rubis` that implements the data type's operations in a sequential manner (Lines 1-17), and the class' companion object (Lines 20-50). The `Rubis` class extends the `ECRO` trait which requires some boilerplate code such as specifying the type of the class, its interface, and providing a method to create new replicas (Lines 15-16). The class' companion object defines the data type's distributed specification. The specification defines three first-order logic predicates: `auction`, `user`, and `bid`. The `auction(auctionId, status, state)` predicate relates the id of an auction to its status (true if it is open, false if it is closed) in a given state. The `user(username, state)` predicate indicates whether or not the username exists in the given state. The `bid(auction, user, amount, state)` predicate represents a bid of some amount from a user on an auction in a given state. These predicates are used to define the preconditions,

Listing 5.4: Excerpt from the replicated RUBiS data type in EFx.

```
1  @replicated
2  class Rubis(users: Set[String], auctions: Map[String, Status],
3              bids: Map[String, List[Bid]]) {
4    pre registerUser(user: String) { !this.users.contains(user) }
5    def registerUser(user: String) =
6      new Rubis(this.users.add(user), this.auctions, this.bids)
7
8    pre placeBid(auction: String, user: String, price: Int) {
9      this.auctions.contains(auction) && this.isOpen(auction) &&
       this.users.contains(user) && price > 0
10   }
11   def placeBid(auction: String, user: String, price: Int) = {
12     val theBids = this.bids.getOrElse(auction, new List[Bid]())
13     val newBids = theBids.append(new Bid(user, price))
14     new Rubis(this.users, this.auctions,
15               this.bids.add(auction, newBids))
16   }
17   // ...
18 }
```

postconditions, and invariants of the operations. The preconditions of the `registerUser` and `placeBid` operations are similar to those of the EFx implementation but are expressed in terms of the aforementioned relations. Since auctions are encoded as predicates, all preconditions must explicitly state that auctions cannot be open and closed at the same time (Line 27-29); a consequence of the way how the status is encoded in the `auction` predicate. In addition, the specification also needs to define a postcondition for every update operation. The postcondition of `registerUser` extends the old state with a new user and explicitly copies all the other user predicates, i.e. the `copyExcept` statement on Line 38 copies all the user predicates from the old state to the new state except the one for `usr`. The postcondition of `placeBid` extends the state with the new bid.

Comparison. Although both implementations are equivalent, we observe that the ECRO implementation is more cumbersome because programmers have to build a complete first-order logic specification for the data type. Building such specifications is non-trivial and error-prone. For instance, the RUBiS specification had to define the necessary first-order logic relations and describe the effects of operations on the state in terms of these relations, whereas the EFx implementation derives all this information from the data type's implementation. We conclude that EFx's

Listing 5.5: Excerpt from the replicated RUBiS data type implemented in Scala with ECROs.

```
case class Rubis(users: Set[String],
                 auctions: Map[String, Status],
                 bids: Map[String,SortedSet[Bid]]) extends ECRO{
  // main operations
  def registerUser(usr: String) = copy(users + usr)
  def placeBid(auct: String, usr: String, price: Int) = {
    val theBids =
      bids.getOrElse(auct, SortedSet.empty[Bid](bidOrdering))
    val newBids = theBids + Bid(usr, price)
    copy(bids = bids + (auct -> newBids))
  }
  // ...

  // boilerplate code required by ECRO trait
  override type T = Rubis; override type I = RubisInterface
  def replicate() = () => Rubis(users, auctions, bids)
}

// Companion object that defines the distributed specification
object Rubis extends DistributedSpec {
  // Define the necessary FOL relations
  val auction: Predicate = ...;
  val user: Predicate = ...; val bid: Predicate = ...
  val relations: Set[Relation] = Set(auction, user, bid)

  // auctions are open xor closed
  def openOrClosed(state: State) =
    not(exists(x :: Stringg) :-
      auction(x, open, state) /\ auction(x, closed, state))

  // define preconditions, postconditions, and invariants
  val preRegisterUser =
    (s: CurrentState) =>
      openOrClosed(s) /\ not (user(username, s))
  val postRegisterUser = (oldS: OldState, newS: NewState) => {
    oldS +
    user(usr, newS) /\ // register user
    user.copyExcept(oldS -> newS, user.name === usr)
  }
  val prePlaceBid = (state: CurrentState) => {
    auction(auct, open, state) /\ // auction is open
    user(usr, state) /\ // user exists
    (price >> 0) /\ openOrClosed(state)
  }
  val postPlaceBid = (oldS: OldState, newS: NewState) => {
    oldS +
    bid(auct, usr, price, newS) /\ // add bid
    bid.copy(oldS -> newS) // copy all bid relations
  }
}
```

contract system greatly reduces the burden that is put on the programmer because the preconditions and invariants are integrated into the language and programmers no longer have to define additional postconditions.

5.6 Performance Evaluation

We now conduct a performance evaluation to assess the practical feasibility of our approach. In particular, we evaluate the EFx compiler in Section 5.6.2 by measuring the time it takes to synthesize ECROs from RDTs implemented in EFx. In Section 5.6.3, we compare the analysis times for RDTs implemented in EFx against the analysis times of the original ECRO implementations. Note that the synthesized ECROs are equivalent to their manually implemented counterparts, therefore we do not repeat a performance evaluation against related work.

5.6.1 Methodology

Similarly to Chapter 4, all experiments reported in this section were conducted on AWS using an `m5.xlarge` VM with 4 virtual CPUs and 16 GiB of RAM, and all benchmarks are implemented with JMH [Ope]. We configured JMH to execute 20 warmup iterations followed by 20 measurement iterations for every benchmark. To avoid run-to-run variance JMH repeats every benchmark in 5 fresh JVM forks, yielding a total of 100 samples per benchmark.

5.6.2 Synthesis Evaluation

To assess if the EFx compiler is fast enough to be used in practice, we measure the total synthesis time for all RDTs implemented in EFx. The results are depicted by the boxplots in Fig. 5.10. Most RDTs (counters, flags, sets, maps, arrays, stacks and queues) are synthesized in less than 6 seconds. The synthesis times of custom RDTs for real-world applications (colored pink) are higher, which is to be expected since their implementations are also bigger (cf. Table 5.1).

Remember that EFx synthesizes ECROs in three steps. First, the RDT and its contract are compiled to SMT. Then, the resulting SMT code is used to analyze the RDT using the ECRO analyses presented in Chapter 4. Finally, the EFx implementation of the RDT is compiled to

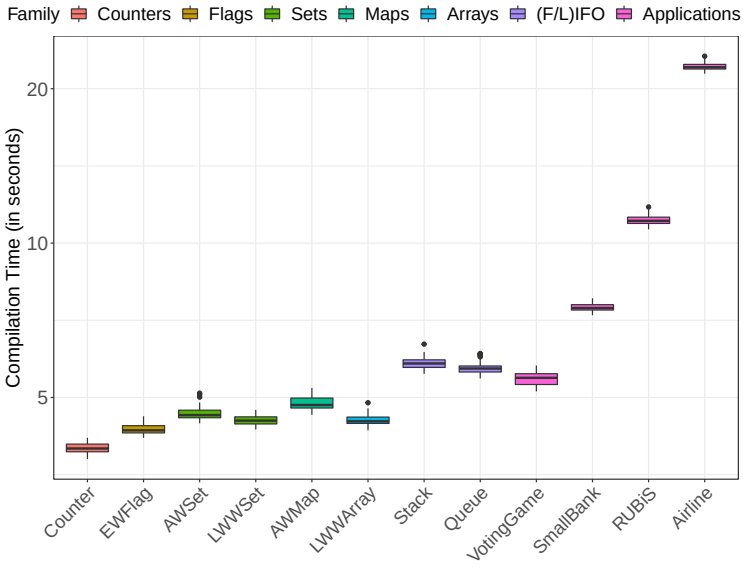


Figure 5.10: Synthesis time of RDTs implemented in EFx.

Scala and transformed into an ECRO using the information derived from the analyses.

Figure 5.11 breaks down the total compilation time in those three phases. For most RDTs, the total compilation time is dominated by the compilation step to SMT and the time spent analyzing the RDT is only a fraction of the total compilation time. However, the portion of the total compilation time that is spent on analyzing the data type increases with the complexity of the RDT. For example, the analysis of the RDTs for the SmallBank and RUBiS applications approximately takes up 30% of the total compilation time and 50% for the airline reservation system, in contrast to only 5% to 10% for the other (simpler) RDTs. This can be explained by the fact that the advanced RDTs expose more operations and the analysis has to consider all operation pairs. We further discuss the analysis times in Section 5.6.3.

Based on the aforementioned observations, we conclude that EFx is suited to implement RDTs because the absolute compilation times are low (in the order of seconds), even for advanced RDTs.

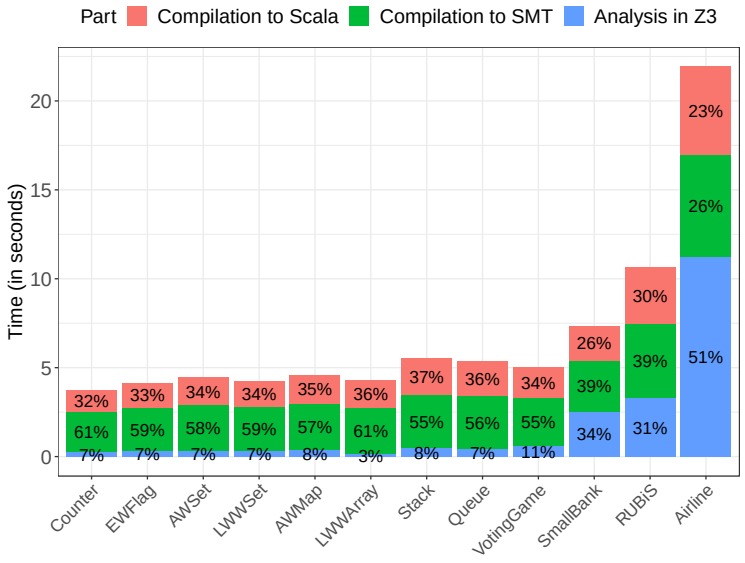


Figure 5.11: Breakdown of the compilation time.

5.6.3 Feasibility of Analyzing High-Level EFx Programs

Recall from Section 5.1.2 that EFx analyzes high-level implementations rather than abstract specifications. We now assess the feasibility of analyzing RDT implementations directly instead of analyzing abstract specifications, as is traditionally the case with hybrid approaches such as ECRO, PoR, and RedBlue.

Table 5.2 compares the analysis times of RDTs implemented in EFx against the original ECROs. For most RDTs, we notice an increase in the analysis times by approximately 200ms for EFx compared to the original ECRO approach, which corresponds to a 2 to 4 times increase in the analysis times. This can be explained by the fact that the analyses execute on the complete RDT implementation which is more complex than an abstract specification (that often ignores implementation details).

For example, consider the remove-wins set. In the ECRO approach, the specification of this set consists of a single predicate `contains` and operations are described in terms of this predicate (cf. Section 4.2.2). However, in EFx, the implementation is analyzed which involves method calls, set operations, and class instantiations (cf. Section 5.1.2).

	Counter	EW-Flag	DW-Flag	AW-Set	RW-Set
EFx	252	303	323	330	325
ECRO	58	67	73	93	95
Δ in ms	+194	+236	+250	+237	+230
Δ in %	+334%	+352%	+342%	+255%	+242%
	AW-Map	RW-Map	Stack	Queue	RUBiS
EFx	367	379	458	398	3290
ECRO	120	117	199	175	4175
Δ in ms	+247	+262	+259	+223	-885
Δ in %	+206%	+224%	+130%	+127%	-21%

Table 5.2: Comparison of the average analysis times (in milliseconds) of RDTs implemented in EFx and ECROs.

Interestingly, the analysis of the RUBiS RDT implemented in EFx is 21% faster than the ECRO implementation. We believe that this comes from the fact that the RUBiS RDT is considerably more complex than the other RDTs. As a result, the first-order logic specification that was manually developed in Chapter 4 is suboptimal and it is faster to directly analyze the implementation.

We conclude that EFx is suited to analyze RDT implementations directly since the absolute analysis times are low (in the order of milliseconds and seconds) and are close to the analysis times of the abstract specifications in the original ECRO approach. For advanced RDTs like RUBiS the analysis times are even improved.

5.7 Discussion

We now elaborate on the main design choices behind EFx. In particular, we focus on the limitations of EFx's traits and functional collections.

Traits. For simplicity, EFx currently supports single inheritance from traits. This could, however, be extended to support multiple inheritance. Traits are not meant for subtyping because subtyping complicates analyses as every subtype needs to be analyzed but these are not necessarily known at compile time. Hence, class fields, method parameters, local variables, etc. cannot be of a trait type.

Note that traits can define type parameters with upper type bounds. These type bounds are only used by the type checker to ensure that every extending class or trait is well-typed. The compiled SMT program does not contain traits as they are effectively compiled away (cf. Section 5.3). Classes and class methods cannot have type parameters with type bounds because the compiler does not know all subtypes.

Functional collections. EFx encodes higher order functions on collections (e.g. `map`, `filter`, etc.) using arrays, which are treated as function spaces in the Combinatory Array Logic (CAL) [MB09]. Hence, anonymous functions (lambdas) merely define arrays which are first-class. SMT solvers can efficiently reason about EFx's functional collections and their higher order operations because CAL is decidable. However, some operations are encoded using universal or existential quantifiers which may hamper decidability. In practice, we were able to analyze RDTs involving complex functional operations.

Unfortunately, EFx's collections do not provide aggregation methods (e.g. `fold` and `reduce`) because this is beyond the capabilities of CAL. Instead, programmers need to manually aggregate the collection by writing recursive methods that loop over the values of the collection. While looping over finite collections works, most SMT solvers will not provide inductive proofs which are needed to analyze recursive functions.

5.8 Notes on Related Work

A lot of work is being put into designing RDTs that serve as basic building blocks for the development of highly available distributed systems. Traditionally, RDTs only guaranteed state convergence. Recently, new approaches have extended RDTs with application-specific invariants.

Hybrid approaches. Indigo [Bal+15] coordinates unsafe operations or requires programmers to provide a deterministic and monotonic algorithm to repair broken invariants. IPA [Bal+18] detects invariant-breaking operations and proposes modifications to the operations in order to preserve the invariants. Q9 [Kak+18] leverages symbolic execution of RDTs built atop their CRDT library to detect anomalies with regard to application invariants. However, these approaches start from *existing* RDTs which are assumed to be correct, while EFx lets programmers build correct RDTs from sequential data types. PoR [LPR18] requires programmers to

specify restrictions over operations to guarantee convergence and maintain invariants. Sieve [Li+14] statically analyzes RDTs and automatically classifies operations as red (unsafe) or blue (safe) in order to guarantee RedBlue Consistency [Li+12]. Hamsaz [HL19] and Hampa [LHL20] derive suitable coordination protocols from the data type's specification.

The aforementioned hybrid approaches require programmers to provide abstract specifications describing the data types' operations and invariants and thus suffer from the problem of disconnected specifications. In contrast, EFx leverages automated analyses of high-level RDT implementations which alleviates the need for separate specifications.

Quelea [SKJ15] supports application-level invariants through contracts that can be associated with individual operations or span several objects. Contracts are first-order logic expressions that use primitive consistency relations to restrict the set of legal executions. Contracts are statically analyzed and mapped to an appropriate consistency level. In contrast, EFx does not require low-level logic expressions since concurrency contracts are integrated into the language.

Mixed-consistency approaches. Several programming languages and models [MM18; De +20; Köh+20; ZN16; MSD18; Hol+16] support mixing consistency levels safely to some extent. However, they cannot automatically derive commutative operations (or custom merge procedures) from sequential data types without programmer intervention. Observable Atomic Consistency [ZH18; ZH20] requires programmers to choose an appropriate consistency level for RDT operations. In contrast, EFx automatically derives correct RDTs from sequential data types based on a concurrency contract.

5.9 Conclusion

This chapter explored a high-level programming language that is powerful enough to implement RDTs, yet simple enough to analyze them automatically without requiring annotations or programmer intervention of any kind. EFx shows that automated analyses of RDTs based on SMT solving removes the need for abstract specifications and thus avoids subtle mismatches that could lead to runtime anomalies.

EFx enables programmers to build custom RDTs from sequential data types by writing concurrency contracts that describe the desired seman-

tics. The data types and their contracts are compiled to SMT where they are analyzed using the ECRO analyses from Chapter 4. The resulting information is used to synthesize correct ECROs for the RDTs.

Our evaluation shows that EFx strikes a good balance between simplicity and efficiency. RDTs written in EFx implement high-level concurrency contracts that are considerably smaller than the low-level ECRO specifications. Moreover, our experiments show that RDT implementations and their concurrency contracts can be analyzed directly and efficiently (as opposed to analyzing abstract specifications which is traditionally the case). The resulting information is used to synthesize ECROs in a matter of seconds. At runtime, the RDTs execute the ECRO protocol which guarantees high availability and low latency.

EFx effectively solves the problems outlined in the introduction according to our vision (cf. Section 1.1.3 and Section 1.2). Programmers can replicate *existing* data types with *custom* concurrency semantics and application-specific *invariants*. EFx analyzes the data type implementation to synthesize an ECRO that is *correct* out-of-the-box.

The key insight behind EFx consists of designing the language such that every feature has an efficient SMT encoding. As such, any EFx program can be compiled to SMT and analyzed using traditional SMT solvers without requiring separate specifications. In the next chapter, we exploit this idea to implement and verify other RDTs beyond ECROs.

Chapter 6

Automated Verification of Replicated Data Types

So far, we proposed several principled approaches (SECRO, ECRO, and EFx) for the development of application-specific RDTs. Nevertheless, there are many ad-hoc designs [Sha+11b; Sha+11a; ASB15; KB17; BAS17; Kak+19; Bur+12; Sha17; Bie+12], and programmers still find themselves modifying these designs to fit their applications. However, designing new RDTs is difficult, as demonstrated by the fact that even seasoned researchers miss subtle corner cases when designing basic replicated data structures such as maps [Kle22]. Therefore, as argued in Chapter 1, it is essential to not only support the development of RDTs but also their verification in order to avoid anomalies.

Although most RDT papers include correctness proofs, those are mostly paper proofs and are subject to reasoning flaws. To avoid the pitfalls of paper proofs, Zeller et al. [ZBP14] and Gomes et al. [Gom+17] propose formal frameworks to verify CRDTs using proof assistants. Such interactive proofs are more convincing because the proof logic is machine-checked. However, these frameworks verify abstract specifications that are disconnected from actual implementations (e.g. Akka's CRDT implementations in Scala). Hence, a particular implementation may be flawed, even though the specification was proven correct. Moreover, interactive proofs require significant programmer intervention, which is time-consuming and reserved to verification experts [LM10; OHe18].

Recent research efforts try to automate (part of) the verification process of CRDTs. Nagar and Jagannathan [NJ19] automatically verify CRDTs under different consistency models but require a first-order logic specification of the CRDTs' operations which is cumbersome and error-prone. Liu et al. [Liu+20] leverage an SMT solver to automate part of the verification process, but significant parts still need to be proven manually due to the way how the language constructs are encoded in SMT. For example, their Map CRDT required more than 1000 lines of proof code.

To simplify the design and implementation of correct RDTs, this chapter introduces VeriFx: a high-level programming language that features a novel proof construct that enables programmers to express custom correctness properties that are verified *automatically*. VeriFx borrows EFx's idea of a fully SMT-encodable language to automate the verification of RDTs. We argue that the ability to implement RDTs and automatically verify them in the *same* language without requiring separate specifications allows programmers to catch mistakes early during the development process and remedy them.

6.1 The Need for a Fully Verifiable Language

To motivate the need for VeriFx, consider a distributed application in Scala with replicated data on top of Akka's highly-available distributed key-value store[1]. The store provides built-in CRDTs, e.g. sets, counters, etc. However, our application requires a Two-Phase Set (2PSet) CRDT [Sha+11a] that is not provided by Akka. We thus need to implement it and verify our implementation.

For the implementation, we can take the specification from Shapiro et al. [Sha+11a]. For the verification, we typically need a complete formalization of the implementation and its correctness conditions which can then be proven manually using proof assistants. The resulting interactive proofs are complex and require considerable expertise. For example, Nieto et al.'s [Nie+22] implementation of a 2PSet in OCaml is only 25 LoC but its specification in Coq is 80 LoC and requires an additional 73 LoC to verify. Alternatively, programmers could resort to Liu et al.'s [Liu+20] extension of Liquid Haskell [Vaz+14] which automates part of the verification process. However, non-trivial RDTs still require

[1] `https://doc.akka.io/docs/akka/current/distributed-data.html`

significant manual proof efforts: 200+ LoC for a replicated set and 1000+ LoC for a replicated map [Liu+20]. Thus, we cannot reasonably assume that programmers have the time nor the skills to manually verify their implementation [LM10; OHe18].

We argue that verification needs to be fully automatic in order to be accessible to non-experts. Figure 6.1 depicts our envisioned workflow for developing RDTs. Programmers start from a new or existing RDT design and implement it in VeriFx which verifies the implementation automatically without requiring a separate formalization. If the implementation is not correct, VeriFx returns a concrete counterexample in which the replicas diverge. In contrast, Liquid Haskell would raise a type error without providing additional information as to why the refinement type is not met. After interpreting the counterexample, the programmer needs to correct the RDT implementation and verify it again. This iterative process repeats until the implementation is correct. Verified RDT implementations can be transpiled to mainstream languages (e.g. Scala or JavaScript) where they can be deployed in an actual system.

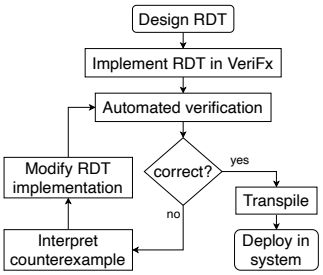


Figure 6.1: Workflow for developing RDTs.

Our envisioned workflow thus verifies RDT implementations right before deployment. In contrast, the traditional workflow only verifies the initial design (cf. Section 1.1.3) but programmers may introduce subtle errors in the implementation phase because they do not always understand the subtleties underlying RDT designs. Moreover, our workflow benefits from a feedback loop allowing programmers to correct implementations based on concrete counterexamples. In contrast, traditional verification techniques such as interactive theorem provers do not provide such feedback; when programmers fail to verify a property, they do not know if the implementation is flawed or if the chosen proof strategy is not suited. To alleviate this issue, counterexample generators based on SAT solving exist [Web08; BN10] some of which are integrated in Isabelle/HOL [BN10].

In the remainder of this section we illustrate each step of our workflow by implementing and verifying an existing 2PSet design in VeriFx, transpiling it to Scala, and deploying it on top of Akka.

6.1.1 Design and Implementation

Algorithm 6 shows the design of the 2PSet CRDT taken from Shapiro et al. [Sha+11a]. The 2PSet is a state-based CRDT whose state (the A and R sets) thus forms a join semilattice, i.e. a partial order $\leq_v$ with a least upper bound (LUB) $\sqcup_v$ for all states. Elements are added to the 2PSet by adding them to the A set and removed by *adding* them to the R set. An element is in the 2PSet if it is in A and not in R. Hence, removed elements can never be added again. Replicas are merged by computing the LUB of their states, which in this case is the union of their respective A and R sets.

The `compare(S,T)` operation checks if $S \leq_v T$ and is used to define state equivalence: $S \equiv T \iff S \leq_v T \wedge T \leq_v S$. Note that state equivalence is defined in terms of $\leq_v$ on the lattice so that replicas may be considered equivalent even though they are not identical. This is relevant for CRDTs that keep additional information. For example, CRDTs often use Lamport clocks together with unique replica identifiers to generate globally unique IDs. The replica identifier is different at every replica and is not part of the lattice even though it is part of the state.

Listing 6.1 shows the implementation of the 2PSet CRDT in VeriFx, which is a straightforward translation of the specification. The `TwoPSet` class is polymorphic in the type of values it stores. It defines the `added` and `removed` fields which correspond to the A and R sets respectively. The `add` and `remove` methods return an updated copy of the state. The class extends the `CvRDT` trait that is provided by VeriFx's CRDT library for building state-based CRDTs (explained later in Section 6.4.1). This trait requires the class to implement the `compare` and `merge` methods.

6.1.2 Verification

We now verify our 2PSet implementation in VeriFx. State-based CRDTs converge if the merge function is idempotent, commutative, and associative [Sha+11b]. VeriFx's CRDT library includes several `CvRDTProof` traits which encode these correctness conditions (explained later in Section 6.4.1). To verify our `TwoPSet`, we define a `TwoPSetProof` object that extends the `CvRDTProof1` trait and passes the type constructor of the CRDT we want to verify (i.e. `TwoPSet`) as a type argument to the trait:

```
object TwoPSetProof extends CvRDTProof1[TwoPSet]
```

Algorithm 6 2PSet CRDT taken from Shapiro et al. [Sha+11a].

payload set A, set R
    initial $\varnothing$, $\varnothing$
query *lookup* (element e) : boolean b
    let $b = (e \in A \wedge e \notin R)$
update *add* (element e)
    $A := A \cup \{e\}$
update *remove* (element e)
    pre *lookup*(e)
    $R := R \cup \{e\}$
compare (S, T) : boolean b
    let $b = (S.A \subseteq T.A \vee S.R \subseteq T.R)$
merge (S, T) : payload U
    let $U.A = S.A \cup T.A$
    let $U.R = S.R \cup T.R$

Listing 6.1: 2PSet implementation in VeriFx, based on Algorithm 6.

```
class TwoPSet[V](added: Set[V],
                 removed: Set[V]) extends CvRDT[TwoPSet[V]] {
  def lookup(element: V) =
    this.added.contains(element) &&
    !this.removed.contains(element)
  def add(element: V) =
    new TwoPSet(this.added.add(element), this.removed)
  def remove(element: V) =
    new TwoPSet(this.added, this.removed.add(element))
  def compare(that: TwoPSet[V]) =
    this.added.subsetOf(that.added) ||
    this.removed.subsetOf(that.removed)
  def merge(that: TwoPSet[V]) =
    new TwoPSet(this.added.union(that.added),
                this.removed.union(that.removed))
}
```

The `TwoPSetProof` object inherits an automated correctness proof for the polymorphic `TwoPSet` CRDT. When executing this object, VeriFx will automatically try to verify this proof. In this case, VeriFx proves that the `TwoPSet` guarantees convergence (independent of the type of elements it holds) according to the notion of state equivalence that is derived from `compare`. However, VeriFx raises a warning that this notion of equivalence does not correspond to structural equality. As explained before, this may be normal in some CRDT designs but it requires further investigation.

VeriFx provides a counterexample consisting of two states S = `TwoPSet({x},{})` and T = `TwoPSet({x},{x})` which are considered equivalent $S \equiv T$ but are not identical $S \neq T$. These two states should indeed not be considered equivalent since $x \in S$ but $x \notin T$ according to `lookup`. Looking back at Algorithm 6, we notice that `compare` defines replica S to be smaller or equal to replica T iff $S.A \subseteq T.A$ *or* $S.R \subseteq T.R$. Since $S.A = T.A$ it follows that $S \leq_v T \wedge T \leq_v S$ and thus they are considered equal ($S \equiv T$) without even considering the removed elements (i.e. the R sets). Thus, after further investigation of the counterexample returned by the warning, we notice that the implementation of `compare` is wrong and we modify it such that it considers both the A *and* R sets as follows:

```
def compare(that: TwoPSet[V]) =
  this.added.subsetOf(that.added) &&
    this.removed.subsetOf(that.removed)
```

We verify the implementation again to check that it still guarantees convergence according to this modified definition of equivalence. VeriFx automatically proves that the modified implementation is correct and the warning about equivalence is now gone. Thus, the definition of equality that is derived from `compare` now corresponds to structural equality, i.e. $s_1 \equiv s_2 \iff s_1 = s_2$.

This example showcases the importance of automated verification as it detected an error in the specification that would have percolated to the implementation. We successfully completed the verification of the 2PSet CRDT in VeriFx without providing any verification-specific code.

6.1.3 Deployment

The final step in our envisioned workflow consists of automatically transpiling the CRDT implementation from VeriFx to Scala and integrating

the CRDT in our distributed application which uses Akka's distributed key-value store.

Listing 6.2: Transpiled 2PSet in Scala.

```
1  case class TwoPSet[V](
2      added: Set[V], removed: Set[V]) extends CvRDT[TwoPSet[V]] {
3    def lookup(element: V) = this.added.contains(element) &&
4                             !this.removed.contains(element)
5    def add(element: V): TwoPSet[V] =
6      TwoPSet[V](this.added + element, this.removed)
7    def remove(element: V): TwoPSet[V] =
8      TwoPSet[V](this.added, this.removed + element)
9    def compare(that: TwoPSet[V]): Boolean =
10     this.added.subsetOf(that.added) &&
11     this.removed.subsetOf(that.removed)
12   def merge(that: TwoPSet[V]): TwoPSet[V] =
13     TwoPSet[V](this.added.union(that.added),
14                this.removed.union(that.removed))
15 }
```

Listing 6.3: Modified 2PSet implementation for integration with Akka's distributed key-value store.

```
1 @SerialVersionUID(1L)
2 case class TwoPSet[V](added: Set[V], removed: Set[V]) extends
      CvRDT[TwoPSet[V]] with ReplicatedData with Serializable {
3   type T = TwoPSet[V]
4   // The remainder of the implementation is unchanged
5 }
```

Listing 6.2 shows the transpiled implementation of the 2PSet in Scala. In order to store the RDT in Akka's distributed key-value store, we need two manual modifications which are shown in Listing 6.3. First, the RDT must extend Akka's `ReplicatedData` trait (Line 2) which requires at least the definition of a type member T corresponding to the actual type of the CRDT (Line 3) and a `merge` method for CRDTs of that type (which we already have). Second, the RDT must be serializable. For simplicity, we use Java's built-in serializer[2]. Hence, it suffices to extend the `Serializable` trait (Line 2) and to annotate the class with a serial version (Line 1). After applying these modifications, our verified `TwoPSet` can be stored in

[2]In production it is safer and more efficient to implement a custom serializer [Lig], for example with Protobuf [Goo].

Akka's distributed key-value store and will automatically be replicated across the cluster and be kept eventually consistent.

6.2 The VeriFx Language

VeriFx aims to be a familiar high-level programming language that is suited to implement and *automatically* verify RDTs. The main challenge consists of efficiently encoding every feature of the language without breaking automatic verification, much as we did in the previous chapter.

We designed VeriFx to be a functional object-oriented programming language with Scala-like syntax and a type system that extends EFx's type system. VeriFx advocates for the object-oriented programming paradigm as it is widespread across programmers and fits the conceptual representation of replicated data as "shared" objects. The functional aspect of the language, in particular its extensive immutable collections akin to those of EFx (cf. Section 5.2.4), make the language suitable for implementing and integrating RDTs in distributed systems, as argued by Helland [Hel15].

VeriFx features a novel proof construct to express application-specific correctness properties. For every proof construct, a proof obligation is derived that is discharged automatically through SMT solving (cf. Section 6.3).

While EFx was designed specifically to simplify the development of ECROs, VeriFx is more general as it is designed to implement and verify RDTs from any family. For instance, we used VeriFx to verify RDTs from the CRDT [Sha+11b] and OT [EG89] approaches (cf. Section 6.5).

The remainder of this section is organized into three parts. First, we give an overview of VeriFx's architecture. Second, we define its syntax. Third, we describe its type system.

6.2.1 Overall Architecture

VeriFx can be seen as an extension of EFx that targets the development of RDTs that go beyond ECROs and features automated verification capabilities to check that the implemented RDTs are correct. Figure 6.2 provides an overview of VeriFx's architecture. VeriFx programs consist of imperative code and proof code (i.e. logic statements).

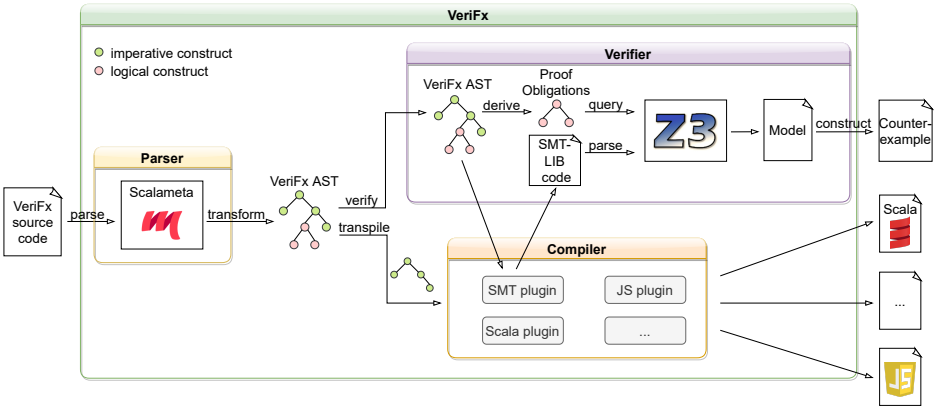


Figure 6.2: VeriFx's architecture.

Source code is parsed into an AST representing the program. The AST can be verified or transpiled to other languages (currently Scala and JavaScript). Transpilation to mainstream languages is done by the compiler which features a plugin architecture akin to that of EFx. Support for other languages can be added by implementing a compiler plugin for them.

To verify the proofs that are defined by a VeriFx program, the verifier derives the necessary proof obligations from the AST. VeriFx then transpiles the program to SMT-LIB and automatically discharges the proof obligations using the Z3 SMT solver [MB08]. For every proof, the outcome is: accepted, rejected, or unknown. Accepted means that the property holds, rejected means that a counterexample was found for which the property does not hold, and unknown means that the property could not be verified within a certain time frame (which is configurable). When a proof is rejected by Z3, VeriFx constructs a high-level counterexample that consists of a concrete assignment of values to variables that violates the given property.

Note that VeriFx can automatically verify application-specific properties because it derives the necessary proof obligations from the program itself. In contrast, EFx leverages Ordana which hardcodes the proof obligations that are needed for the ECRO analyses presented in Chapter 4.

6.2.2 Syntax

Figure 6.3 defines the syntax of VeriFx which is an adaption of EFx. Changes and extensions are marked in blue. We use the same meta variables as in EFx (cf. Section 5.2.2) and introduce some new ones: O ranges over object names; E ranges over enumeration names; K ranges over constructor names of enumerations; p ranges over proof names.

VeriFx programs consist of one or more statements which can be the definition of an object O, a class $C\langle\overline{X}\rangle$, a trait $I\langle\overline{X}\rangle$, or an enumeration $E\langle\overline{X}\rangle$. Objects, classes, enumerations, and traits can be polymorphic and inherit from a single trait (except enumerations). Objects define zero or more methods and proofs. Classes contain zero or more fields and (polymorphic) methods. The body of a method must contain a well-typed expression e. Traits can declare values and methods that need to be provided by concrete classes extending the trait, and define (polymorphic) methods and proofs with default implementations. Traits can express upper type bounds on their type parameters to restrict the possible extensions. Enumerations (enums for short) define one or more constructors, each of which contains zero or more fields.

Unique to VeriFx is its `proof` construct which defines a name for the proof p and whose body must be a well-typed boolean expression e representing the property to verify. A proof is accepted if VeriFx can show that its body always evaluates to true. It is rejected if VeriFx found a concrete counterexample for which the property does not hold. It may also timeout in which case we do not know if the property holds.

Proofs can be polymorphic, which means that they verify a property for all possible type instantiations of the proof's type parameters. Poly-

$$
\begin{array}{rcl}
L & ::= & \texttt{class}\ C\,\langle\overline{X}\rangle\,(\overline{v}:\overline{T})\,\{\,\overline{M}\,\} \\
 & \mid & \texttt{class}\ C\,\langle\overline{X}\rangle\,(\overline{v}:\overline{T})\ \texttt{extends}\ I\,\langle\overline{P}\rangle\{\,\overline{M}\,\} \\
J & ::= & \texttt{object}\ O\,\{\,\overline{D}\,\} \\
 & \mid & \texttt{object}\ O\ \texttt{extends}\ I\,\langle\overline{T}\rangle\,\{\,\overline{D}\,\} \\
F & ::= & \texttt{trait}\ I\,\langle\overline{X}<:\overline{T}\rangle\,\{\,\overline{B}\,\} \\
 & \mid & \texttt{trait}\ I\,\langle\overline{X}<:\overline{T}\rangle\ \texttt{extends}\ I\,\langle\overline{P}\rangle\,\{\,\overline{B}\,\} \\
N & ::= & \texttt{enum}\ E\,\langle\overline{X}\rangle\,\{\,K\,(\overline{v}:\overline{T})\,\} \\
D & ::= & M \mid R \\
B & ::= & \mathit{valDecl} \mid \mathit{methodDecl} \mid M \mid R \\
R & ::= & \texttt{proof}\ p\,\langle\overline{X}\rangle\,\{\,e\,\} \\
\mathit{valDecl} & ::= & \texttt{val}\ x:T \\
\mathit{methodDecl} & ::= & \texttt{def}\ m\,\langle\overline{X}\rangle(\overline{x}:\overline{T}):T
\end{array}
\qquad
\begin{array}{rcl}
M & ::= & \texttt{def}\ m\,\langle\overline{X}\rangle\,(\overline{x}:\overline{T}):T=e \\
T & ::= & \texttt{int} \mid \texttt{string} \mid \texttt{bool} \mid C\,\langle\overline{T}\rangle \mid I\,\langle\overline{T}\rangle \\
 & \mid & E\,\langle\overline{T}\rangle \mid \tilde{T}\to T \\
e & ::= & \texttt{num} \mid \texttt{str} \mid \texttt{true} \mid \texttt{false} \\
 & \mid & e\oplus e \mid e\otimes e \mid \texttt{!}e \mid x \mid e.v \mid e.m\,\langle\overline{T}\rangle\,(\overline{e}) \\
 & \mid & \texttt{val}\ x:T=e\ \texttt{in}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & \mid & (\tilde{x}:\tilde{T})\Rightarrow e \mid e(\overline{e}) \\
 & \mid & \texttt{new}\ C\langle\overline{T}\rangle(\overline{e}) \mid \texttt{new}\ K\langle\overline{T}\rangle(\overline{e}) \\
 & \mid & e\ \texttt{match}\ \{\texttt{case}\ r\Rightarrow e\} \\
 & \mid & \texttt{forall}\,(\tilde{x}:\tilde{T})\,.\,e \mid \texttt{exists}\,(\tilde{x}:\tilde{T})\,.\,e \\
 & \mid & e\implies e \\
r & ::= & K(\overline{x}) \mid x \mid \_
\end{array}
$$

Figure 6.3: VeriFx syntax.

morphic proofs are useful to verify that a polymorphic RDT converges independent of the type of values it operates on. For example, a polymorphic proof can verify that a set RDT converges independent of the type of values (integers, strings, etc.) it holds. This avoids having to define a proof for every type instantiation of the set.

Compared to EFx, VeriFx introduces three new expressions. The first is the instantiation of enumerations E by calling one of the enumeration's constructors K. The second is a `match` expression to pattern match on enumerations. The third consists of logic expressions such as quantified boolean formulas and logical implication.

Similar to EFx, VeriFx does not support true subtyping because that requires proofs about supertypes to verify the property for all subtypes but those are not necessarily known at compile time when translating the program to SMT-LIB. As a result, traits merely serve for code reuse by moving recurring code to an abstract trait that is inherited (i.e. mixed-in) by all extending traits and classes. In contrast, enumerations are supported because their constructors are fixed and known at compile time.

6.2.3 Type System

VeriFx's type system is an extension of EFx's type system (cf. Appendix F) with typing rules for the aforementioned new constructs such as proofs, enumerations, logic expressions, etc.

We extend the judgment for type well-formedness with a rule for enumeration types (WF-Enum). An enumeration type is well-formed if a corresponding definition exists and the type arguments are well-formed.

$$\frac{\Delta \vdash \overline{T}\ \text{ok} \qquad \texttt{enum}\ E\,\langle \overline{X} \rangle\,\{\ldots\}}{\Delta \vdash E\langle \overline{T} \rangle\ \text{ok}}\ (\textit{WF-Enum})$$

We now introduce two auxiliary functions that are used by the typing rules. The *ftypes* function takes an enumeration type and the name of one of its constructors and returns the type of the fields of that constructor:

$$\frac{\texttt{enum}\ E\,\langle \overline{X} \rangle\,\{\,K(\overline{v} : \overline{T}), \ldots\}}{\mathit{ftypes}(E\langle \overline{P} \rangle, K) = [\overline{P}/\overline{X}]\ \overline{T}}\ (\text{FT-ENUM})$$

The *ctors* function takes an enumeration type and returns the names of its constructors:

$$\frac{\texttt{enum}\ E\,\langle \overline{X}\rangle\,\{\,\overdots{K\,(\overline{x} : \overline{T})}\,\}}{ctors(E\langle \overline{P}\rangle) = \overdots{K}}\ (\text{C-ENUM})$$

Figure 6.4 extends EFx's well-formedness judgment (cf. Fig. F.2) with rules for enumerations and proofs. Enumerations are well-formed if the types of the fields of its constructors are well-formed. Proofs are well-formed if the body is a well-typed boolean expression. We also modify the rules for classes (T-CLASS1 and T-CLASS2) because, unlike EFx, VeriFx does not feature preconditions and invariants. Those were only used in EFx to analyze and synthesize ECROs which is not the goal of VeriFx. A class definition is well-formed if it implements the fields and methods required by its hierarchy of super traits and all methods are well-formed.

$$\frac{\Delta = \overline{X} \quad \Delta \vdash \overline{T}\ \text{ok}}{\texttt{enum}\ E\,\langle \overline{X}\rangle\,\{\,\overdots{K\,(\overline{v} : \overline{T})}\,\}\ \mathit{OK}}\ (\text{T-ENUM})$$

$$\frac{\Delta = \overline{X} \quad \Delta; \emptyset \vdash e : \texttt{bool}}{\texttt{proof}\ p\,\langle \overline{X}\rangle\,\{\,e\,\}\ \mathit{OK}}\ (\text{T-PROOF})$$

$$\frac{\begin{array}{c}\Delta = \overline{X} \quad \Delta \vdash \overline{T}\ \text{ok} \\ \overline{M}\ \mathit{OK}\ \mathit{IN}\ C\langle \overline{X}\rangle\end{array}}{\texttt{class}\ C\,\langle \overline{X}\rangle\,(\overline{v} : \overline{T})\,\{\,\overline{M}\,\}\ \mathit{OK}}\ (\text{T-CLASS1})$$

$$\frac{\begin{array}{c}\Delta = \overline{X} \qquad \Delta \vdash \overline{T}\ \text{ok} \qquad \Delta \vdash I\langle \overline{P}\rangle\ \text{ok} \\ \texttt{trait}\ I\,\langle \ldots\rangle\,\{\,B\,\}\ \mathit{or}\ \texttt{trait}\ I\,\langle \ldots\rangle\ \texttt{extends}\ \ldots\,\{\,B\,\} \\ \overline{M}\ \mathit{OK}\ \mathit{IN}\ C\langle \overline{X}\rangle \\ \mathit{valNames}(I\langle \overline{P}\rangle) \subset \overline{v} \qquad \mathit{declaredMethods}(I\langle \overline{P}\rangle) \subset \overline{M}\end{array}}{\texttt{class}\ C\,\langle \overline{X}\rangle\,(\overline{v} : \overline{T})\ \texttt{extends}\ I\langle \overline{P}\rangle\{\,\overline{M}\,\}\ \mathit{OK}}\ (\text{T-CLASS2})$$

Figure 6.4: Judgments for well-formedness of enumerations and proofs in VeriFx.

Figure 6.5 extends EFx's type system with typing rules for the new VeriFx expressions, namely: logic expressions, enumeration instantiations, and pattern matching. We introduce three typing rules for logic expressions. Quantified formulas are well-typed boolean expressions if their body

types to a boolean expression in the environment that is extended with the quantified variables (T-UNI and T-EXI rules). Logical implication is a well-typed boolean expression if both the antecedent and the consequent are boolean expressions (T-IMPL rule).

When instantiating an enumeration through one of its constructors $\texttt{new}\ K\langle\overline{P}\rangle(\,\overline{e}\,)$, the provided arguments $\overline{e}$ need to match the types of the constructor's fields, and the resulting object is of the enumeration type $E\langle\overline{P}\rangle$.

Programmers can pattern match on enumerations but the cases must be exhaustive, i.e. every constructor must be matched by at least one case. In addition, all cases must have the same type T in order for the pattern match expression to be well-typed and also have type T.

$$\frac{\Delta \vdash \ddot{\overline{T}}\ \text{ok} \quad \Delta;\Gamma,\ddot{\overline{x}} : \ddot{\overline{T}} \vdash e : \texttt{bool}}{\Delta;\Gamma \vdash \texttt{forall}\,(\ddot{\overline{x}} : \ddot{\overline{T}})\texttt{.}\,e : \texttt{bool}}\ (\text{T-UNI}) \qquad \frac{\Delta \vdash \ddot{\overline{T}}\ \text{ok} \quad \Delta;\Gamma,\ddot{\overline{x}} : \ddot{\overline{T}} \vdash e : \texttt{bool}}{\Delta;\Gamma \vdash \texttt{exists}\,(\ddot{\overline{x}} : \ddot{\overline{T}})\texttt{.}\,e : \texttt{bool}}\ (\text{T-EXI})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \texttt{bool} \quad \Delta;\Gamma \vdash e_2 : \texttt{bool}}{\Delta;\Gamma \vdash e_1 \implies e_2 : \texttt{bool}}\ (\text{T-IMPL}) \qquad \frac{\begin{array}{c}\mathit{ctors}(E\langle\overline{P}\rangle) = \ddot{\overline{K}} \quad K \in \ddot{\overline{K}} \\ \mathit{ftypes}(E\langle\overline{P}\rangle, K) = \overline{T} \\ \Delta \vdash E\langle\overline{P}\rangle\ \text{ok} \quad \Delta;\Gamma \vdash \overline{e} : \overline{T}\end{array}}{\Delta;\Gamma \vdash \texttt{new}\ K\langle\overline{P}\rangle(\,\overline{e}\,) : E\langle\overline{P}\rangle}\ (\text{T-NEW-ENUM})$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_0 : E\langle\overline{P}\rangle \\ (\mathit{ctors}(E\langle\overline{P}\rangle) \setminus \dot{c} = \emptyset) \vee (\texttt{case}\ x \Rightarrow e \in \dot{c}) \vee (\texttt{case}\ \_ \Rightarrow e \in \dot{c}) \\ \mathit{for\ each}\ c \in \dot{c} : \Delta;\Gamma \vdash c : T\ \mathit{IN}\ e_0\ \texttt{match}\ \{\ldots\}\end{array}}{\Delta;\Gamma \vdash e_0\ \texttt{match}\ \{\dot{c}\} : T}\ (\text{T-MATCH})$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_0 : E\langle\overline{P}\rangle \\ \mathit{ftypes}(E\langle\overline{P}\rangle, K) = \overline{Q} \\ \Delta;\Gamma,\overline{x} : \overline{Q} \vdash e : T\end{array}}{\Delta;\Gamma \vdash \texttt{case}\ K(\overline{x}) \Rightarrow e : T\ \mathit{IN}\ e_0\ \texttt{match}\ \{\ldots\}}\ (\text{T-CTOR-PTN})$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_0 : E\langle\overline{P}\rangle \\ \Delta;\Gamma, x : E\langle\overline{P}\rangle \vdash e : T\end{array}}{\Delta;\Gamma \vdash \texttt{case}\ x \Rightarrow e : T\ \mathit{IN}\ e_0\ \texttt{match}\ \{\ldots\}}\ (\text{T-NAMED-PTN})$$

$$\frac{\Delta;\Gamma \vdash e : T}{\Delta;\Gamma \vdash \texttt{case}\ \_ \Rightarrow e : T\ \mathit{IN}\ e_0\ \texttt{match}\ \{\ldots\}}\ (\text{T-WCARD-PTN})$$

Figure 6.5: Typing rules for the new VeriFx expressions.

6.3 Automated Proof Verification

Like EFx, VeriFx leverages SMT solving to enable automated verification of user-defined proofs. To this end, VeriFx programs are compiled to SMT-LIB, the language of SMT solvers. As explained in Section 5.3, SMT-LIB is low-level and is not meant to be used directly by programmers to verify high-level programs. Instead, SMT-LIB is often used internally by IVLs to discharge proof obligations using an appropriate SMT solver. IVLs like Dafny [Lei10], Spec# [BLS05], and Why3 [FP13]) are designed to be general-purpose but this breaks automated verification and forces programmers to specify preconditions and postconditions on methods, loop invariants, etc. (cf. Section 2.3.1).

VeriFx can be regarded as a specialized high-level IVL, that like EFx, was carefully designed such that all language features have an *efficient* SMT encoding; leaving out features that break automated verification.

In the remainder of this section we show how to compile VeriFx to Core SMT and how to derive proof obligations that can be discharged automatically by SMT solvers.

6.3.1 Compiling VeriFx to Core SMT

Similar to EFx, we define the semantics of VeriFx by means of translation functions that compile VeriFx programs to Core SMT (cf. Section 5.3.1). We extend the translation functions of EFx (cf. Section 5.3.2) with appropriate rules for VeriFx's new language features.

First, we extend the translation function for types $[\![]\!]_t$ with a rule for enumeration types that keeps the enumeration's name and recursively compiles the provided type arguments:

$$[\![E\langle \overline{\mathsf{T}} \rangle]\!]_t = E\langle \overline{[\![\mathsf{T}]\!]_t} \rangle$$

We also extend the translation function $def[\![]\!]$ with a rule for compiling enumeration definitions. For every enumeration an ADT is constructed with the same name, type parameters, and constructors. The types of the fields are translated recursively:

$$def[\![\texttt{enum}\ E\ \langle \overline{X} \rangle\ \{\ \overline{K\ (\overline{v} : \overline{T})}\ \}]\!]\ =\ \mathsf{adt}\ E\langle \overline{X} \rangle \{ \overline{K(\overline{v} : \overline{[\![T]\!]_t})} \}$$

VeriFx allows programmers to instantiate enumerations by using the `new` keyword with one of the enumeration's constructors. Since enumera-

tions are represented by ADTs in Core SMT, the enumeration's constructors are plain SMT functions. Thus, instantiations are compiled to a call of the corresponding constructor function:

$$[\![\texttt{new}\ K\langle\overline{T}\rangle(\,\overline{e}\,)]\!] = K\langle\overline{[\![T]\!]_t}\rangle(\overline{[\![e]\!]})$$

To use values of an enumeration type, programmers must first pattern match the value against the possible constructors. Figure 6.6 extends the translation function for expressions $[\![]\!]$, originally defined in Section 5.3.2, with a rule for pattern match expressions that compiles the expression to a similar pattern match expression in Core SMT. To this end, every pattern is compiled using a new $pat[\![]\!]$ function. Core SMT supports two types of patterns: constructor patterns $K(\overline{x})$ that match an algebraic data type constructor K and bind its fields to the provided names $\overline{x}$, and wildcard patterns x that match any value and give it a name x. A wildcard pattern may use an underscore to match any value without binding it to a name. Every VeriFx pattern is compiled to the corresponding Core SMT pattern.

$$\begin{array}{lcl}
[\![e\ \texttt{match}\ \{\overline{\texttt{case}\ r \Rightarrow e_c}\}]\!] & = & \mathsf{match}([\![e]\!],\ \overline{pat[\![\texttt{case}\ r \Rightarrow e_c]\!]}) \\
pat[\![\texttt{case}\ K(\overline{x}) \Rightarrow e]\!] & = & \mathsf{case}(K(\overline{x}),\ [\![e]\!]) \\
pat[\![\texttt{case}\ x \Rightarrow e]\!] & = & \mathsf{case}(x,\ [\![e]\!]) \\
pat[\![\texttt{case}\ _ \Rightarrow e]\!] & = & \mathsf{case}(_,\ [\![e]\!])
\end{array}$$

Figure 6.6: Compiling pattern match expressions to Core SMT.

In addition to enumerations, VeriFx also introduces logical expressions such as quantified formulas and logical implication. Figure 6.7 defines the compilation rules for each expression. Quantified formulas are compiled to the corresponding SMT formula and the types of the variables $\vec{T}$ and the formula e are compiled recursively. Similarly, logical implication is compiled to logical implication in SMT and the antecedent and the consequent are compiled recursively.

$$\begin{array}{lcl}
[\![\texttt{forall}\,(\vec{x} : \vec{T})\,\texttt{.}\,e]\!] & = & \forall(\vec{x} : \vec{[\![T]\!]_t}).[\![e]\!] \\
[\![\texttt{exists}\,(\vec{x} : \vec{T})\,\texttt{.}\,e]\!] & = & \exists(\vec{x} : \vec{[\![T]\!]_t}).[\![e]\!] \\
[\![e_1 \implies e_2]\!] & = & [\![e_1]\!] \implies [\![e_2]\!]
\end{array}$$

Figure 6.7: Compiling logical expressions to Core SMT.

Finally, objects are singletons that can define methods and proofs, and are compiled as follows:

$$\begin{aligned} &\mathit{def}[\![\texttt{object}\ O\ \texttt{extends}\ I\langle\overline{T}\rangle\ \{\ \overline{M}\ ;\ \overline{R}\ \}]\!] \ = \\ &\quad \mathit{def}[\![\texttt{class}\ O'()\ \{\overline{M}\}\ \texttt{extends}\ I\langle\overline{T}\rangle]\!]\ ;\ \mathsf{const}\ O\ O'\ ;\ \mathsf{assert}\ O == O'()\ ;\ \mathit{def}[\![\overline{R}]\!] \end{aligned}$$

The object is compiled to a regular class with a fresh name O'. Then, a single instance of that class is created and assigned to a constant named after the object O. The proofs defined by the object are compiled to functions. How to translate proofs into functions is the subject of the next section.

6.3.2 Deriving Proof Obligations

VeriFx features libraries for CRDTs [Sha+11b] and OT [EG89] which internally use our novel proof construct to define the necessary correctness properties (discussed later in Section 6.4). However, programmers can also define custom proofs themselves, for instance, to verify data invariants.

We now explain how proof obligations are derived from user-defined proofs in VeriFx programs. Proofs are compiled to regular functions without arguments. The name and type parameters remain unchanged and the body of the proof is compiled and becomes the function's body. Proofs always return a boolean since the body is a logical formula whose satisfiability must be checked.

$$\mathit{def}[\![\texttt{proof}\ p\ \langle\overline{X}\rangle\ \{\ e\ \}]\!] \ = \ \mathsf{fun}\ p\langle\overline{X}\rangle() : \mathsf{bool} = [\![e]\!]$$

To check if the property described by a proof holds, the negation of the proof must be unsatisfiable. In other words, if no counterexample exists, it constitutes a proof that the property is correct. A (polymorphic) proof called p with i type parameters is checked as follows:

$$\begin{aligned} \mathit{prove}(p, i) \quad = \quad & \mathsf{sort}\ S_1\ 0\ ;\ \ldots\ ;\ \mathsf{sort}\ S_i\ 0\ \ ; \\ & \mathsf{assert}\ \neg p\langle S_1, \ldots, S_i\rangle()\ \ ; \\ & \mathsf{check}() == \mathsf{UNSAT} \end{aligned}$$

For every type parameter i an *uninterpreted* sort with a unique name S_i is declared. Then, the proof function is called with those sorts as type arguments and we check that the negation is unsatisfiable. If the negation is unsatisfiable, the (polymorphic) proof holds for all possible instantiations of its type parameters. The underlying SMT solver can generate an

actual proof which could be reconstructed by proof assistants as shown in [Böh+11; BW10]. On the other hand, if the negation is satisfiable that means that a model (i.e. an assignment of values to variables) exists for which the property does not hold. In the next section, we explain how VeriFx queries this low-level model returned by the SMT solver to construct a high-level counterexample that VeriFx programmers can understand.

6.3.3 Constructing High-Level Counterexamples

We now detail how VeriFx constructs high-level counterexamples for rejected proofs. Remember that VeriFx introduces an uninterpreted sort for every type parameter of a generic proof. When a proof is rejected, the underlying SMT solver returns a concrete model that violates the property we are trying to verify. Such a model M consists of two parts:

- An interpretation for the uninterpreted sorts: $Sorts = \{\langle S_1, V_1\rangle, \ldots, \langle S_i, V_i\rangle\}$ where S_i corresponds to the i-th declared sort and $V_i = \{v_1, v_2, \ldots, v_n\}$ is the set of all values belonging to S_i.
- A set of assignments of values to variables: $Vars = \{\langle var, val\rangle, \ldots\}$.

For a generic proof, $Sorts$ corresponds to the interpretation of the proof's type parameters and $Vars$ assigns values to the proof's variables such that the proof does not hold. For example, consider the generic proof below which states that for any type, any two values are always equal:

```
proof alwaysEquals[A] {
  forall (x: A, y: A) {
    x == y
  }
}
```

Clearly, this property does not hold for types that contain at least two distinct values. Thus, the proof is rejected and the underlying SMT solver returns a concrete model: $M = \langle Sorts, Vars\rangle$ where $Sorts = \{\langle A, \{a_0, a_1\}\rangle\}$ and $Vars = \{\langle x, a_0\rangle, \langle y, a_1\rangle\}$. This model defines the type parameter A to be a sort containing two values a_0 and a_1, and assigns these two distinct values to the variables x and y defined by the universally quantified formula. Although the model represents a counterexample for the proof, it is hard to understand for programmers because the model is in Core SMT format, i.e. the sorts and values are represented in Core SMT which does not match the high-level VeriFx code written by the programmer.

To provide programmers with *high-level* counterexamples, VeriFx translates low-level SMT models to VeriFx. To this end, it uses a translation function $[\![]\!]^{-1}$ that transforms Core SMT code back to VeriFx. Note that $[\![]\!]^{-1}$ is the inverse of the translation function $[\![]\!]$ that compiles VeriFx code to Core SMT and was defined in Section 5.3.2.

Algorithm 7 Constructing high-level counterexamples in VeriFx from low-level SMT models.

```
1:  function constructCounterExample(M)
2:      ⟨Sorts, Vars⟩ ← M
3:      tparams ← ∅
4:      vars ← ∅
5:      for ⟨tparam, values⟩ ∈ Sorts do
6:          v_1, ..., v_n ← values
7:          tparams ← tparams ∪ { enum E { v_1 | ... | v_n } }
8:      for ⟨var, val⟩ ∈ Vars do
9:          vars ← vars ∪ { ⟨var, [[val]]^{-1}⟩ }
10:     return ⟨tparams, vars⟩
```

Algorithm 7 defines how VeriFx constructs a high-level counterexample from a low-level SMT model M. Every uninterpreted sort corresponds to a type parameter of the proof. For every such sort, the algorithm defines a corresponding enumeration containing the values defined by the model (line 7). Then, for every variable assignment $\langle var, val\rangle$ in the model, the algorithm creates a corresponding assignment to var but translates the assigned SMT value val back to VeriFx using the inverse translation function $[\![val]\!]^{-1}$ (line 9). Finally, the algorithm returns a tuple containing the translated sorts and variable assignments.

6.4 Libraries for Implementing and Verifying Replicated Data Types

VeriFx aims to simplify the development of correct RDTs by integrating automated verification capabilities in the language. Based on our experience with implementing RDTs, we noticed that RDTs need to fulfill specific correctness properties that are well-defined for each RDT family. Therefore, we built libraries for the development and automated verification of two well-known RDT families: CRDTs [Sha+11b] and OT [EG89].

These libraries are written in VeriFx and define proofs that encode the necessary correctness properties such that programmers do not need to redefine these proofs for every RDT they implement.

The remainder of this section discusses the aforementioned libraries. For each library, we formally define the correctness properties that must be verified for that specific RDT family. Section 6.4.1 describes the implementation of a general execution model for CRDTs and its verification library in VeriFx. Then, Section 6.4.2 focuses on the OT library and the verification of transformation functions. Finally, Section 6.4.3 explains how to encode common assumptions such as causal delivery in VeriFx since the CRDT and OT libraries do not make specific assumptions.

6.4.1 CRDT Library

CRDTs guarantee SEC, a consistency model that strengthens eventual consistency with the *strong convergence* property which requires replicas that received the same updates, possibly in a different order, to be in the same state (cf. Section 2.1.2). VeriFx's CRDT library supports several families of CRDTs, including state-based [Sha+11b], op-based [Sha+11b], and pure op-based CRDTs [BAS17]. The remainder details each family.

6.4.1.1 State-Based CRDTs

State-based CRDTs - also known as Convergent Replicated Data Types (CvRDTs) - periodically broadcast their state to all replicas and merge incoming states by computing the Least Upper Bound (LUB) of the incoming state and their own state. Shapiro et al. [Sha+11b] showed that CvRDTs converge if the merge function $\sqcup_v$ is idempotent, commutative, and associative. Based on their work, we define these properties as follows:

Idempotent: $\forall x \in \Sigma : \mathit{reachable}(x) \implies x \equiv x \sqcup_v x$

Commutative: $\forall x, y \in \Sigma : \mathit{reachable}(x) \land \mathit{reachable}(y) \land \mathit{compatible}(x, y)$
$\implies (x \sqcup_v y \equiv y \sqcup_v x) \land \mathit{reachable}(x \sqcup_v y)$

Associative: $\forall x, y, z \in \Sigma : \mathit{reachable}(x) \land \mathit{reachable}(y) \land \mathit{reachable}(z) \land$
$\mathit{compatible}(x, y) \land \mathit{compatible}(x, z) \land \mathit{compatible}(y, z)$
$\implies ((x \sqcup_v y) \sqcup_v z \equiv x \sqcup_v (y \sqcup_v z)) \land \mathit{reachable}((x \sqcup_v y) \sqcup_v z)$

Σ denotes the set of all states. A state is *reachable* if it can be reached starting from the initial state and applying only supported operations.

Two states are *compatible* if they represent different replicas of the same CRDT object[3]. As explained in Section 6.1.1, state equivalence is defined in terms of $\leq_v$ on the lattice: $S \equiv T \iff S \leq_v T \wedge T \leq_v S$.

Listing 6.4: Trait for the implementation of CvRDTs in VeriFx.

```
trait CvRDT[T <: CvRDT[T]] {
  def merge(that: T): T
  def compare(that: T): Boolean
  def reachable(): Boolean = true
  def compatible(that: T): Boolean = true
  def equals(that: T): Boolean = {
    this.asInstanceOf[T].compare(that) &&
    that.compare(this.asInstanceOf[T])
  }
}
```

VeriFx's CRDT library provides traits for the implementation and verification of CvRDTs, shown in Listings 6.4 and 6.5 respectively. Listing 6.4 shows the `CvRDT` trait that was used in Listing 6.1 to implement the `TwoPSet` CRDT. Every state-based CRDT that extends the `CvRDT` trait must provide a type argument which is the actual type of the CRDT and provide an implementation for the `merge` and `compare` methods. By default, all states are considered reachable and compatible, and state equivalence is defined in terms of `compare`. These methods can be overridden by concrete CRDTs that implement the trait.

Listing 6.5 shows the `CvRDTProof` trait used to verify CvRDT implementations. This trait defines one type parameter `T` that must be a CvRDT type and defines proofs to check that its merge function adheres to the aforementioned properties (i.e. is idempotent, commutative, and associative). It also defines an additional proof `equalityCheck` that checks if the notion of state equivalence (that is derived from `compare`) corresponds to structural equality.

Objects can extend the `CvRDTProof` trait to inherit automated correctness proofs for the given CRDT type. However, the trait's type parameter `T` expects a concrete CvRDT type (e.g. `PNCounter`) and will not work for polymorphic CvRDTs (e.g. `TwoPSet`) because those are type constructors. Instead, the CRDT library provides additional `CvRDTProof1`,

[3]The `compatible` predicate can be used to encode certain assumptions. For example, to encode that replicas have a unique ID which enables them to generate unique tags.

Listing 6.5: Trait for the verification of CvRDTs in VeriFx. The arrow function `=>:` implements logical implication.

```
1  trait CvRDTProof[T <: CvRDT[T]] {
2    proof mergeIdempotent {
3      forall (x: T) {
4        x.reachable() =>: x.merge(x).equals(x)
5    } }
6    proof mergeCommutative {
7      forall (x: T, y: T) {
8        (x.reachable() && y.reachable() && x.compatible(y)) =>:
9        (x.merge(y).equals(y.merge(x)) && x.merge(y).reachable())
10   } }
11   proof mergeAssociative {
12     forall (x: T, y: T, z: T) {
13       (x.reachable() && y.reachable() && z.reachable() &&
14        x.compatible(y) && x.compatible(z) && y.compatible(z))
15       =>: (x.merge(y).merge(z).equals(x.merge(y.merge(z))) &&
16            x.merge(y).merge(z).reachable())
17   } }
18   proof equalityCheck {
19     forall (x: T, y: T) {
20       x.equals(y) == (x == y)
21 } } }
```

`CvRDTProof2`, and `CvRDTProof3` traits to verify polymorphic CvRDTs that expect 1, 2, or 3 type arguments respectively. For example, the `TwoPSetProof` object defined in Section 6.1.2 extends the `CvRDTProof1` trait because the `TwoPSet` expects one type argument.

6.4.1.2 Operation-Based CRDTs

Operation-based CRDTs - also known as Commutative Replicated Data Types (CmRDTs) - execute update operations in two phases, called *prepare* and *effect*. The prepare phase executes locally at the source replica (only if its source precondition holds) and prepares a message to be broadcast[4] to all replicas (including itself). The effect phase applies such incoming messages and updates the state (only if its downstream precondition holds, otherwise the message is ignored).

Shapiro et al. [Sha+11b] and Gomes et al. [Gom+17] have shown that CmRDTs guarantee SEC if all concurrent operations commute. Hence,

[4]While some CmRDT designs do not require causal delivery, the overall model assumes reliable causal broadcast.

for any CmRDT it suffices to show that all pairs of concurrent operations commute. Formally, for any operation o_1 that is enabled by some reachable replica state s_1 (i.e. o_1's source precondition holds in s_1) and any operation o_2 that is enabled by some reachable replica state s_2, if these operations can be concurrent, and s_1, s_2, and s_3 are compatible replica states, then we must show that on any reachable replica state s_3 the operations commute and the intermediate and resulting states are all reachable:

$$\begin{aligned}&\forall s_1, s_2, s_3 \in \Sigma, \forall o_1, o_2 \in \Sigma \to \Sigma : \mathit{reachable}(s_1) \land \mathit{reachable}(s_2) \land \mathit{reachable}(s_3) \land \\ &\quad \mathit{enabledSrc}(o_1, s_1) \land \mathit{enabledSrc}(o_2, s_2) \land \mathit{canConcur}(o_1, o_2) \land \\ &\quad \mathit{compatible}(s_1, s_2) \land \mathit{compatible}(s_1, s_3) \land \mathit{compatible}(s_2, s_3) \\ &\quad\quad \implies o_2 \cdot o_1 \cdot s_3 \equiv o_1 \cdot o_2 \cdot s_3 \land \mathit{reachable}(o_1 \cdot s_3) \land \\ &\quad\quad\quad \mathit{reachable}(o_2 \cdot s_3) \land \mathit{reachable}(o_1 \cdot o_2 \cdot s_3)\end{aligned}$$

We use the notation $o \cdot s$ to denote the application of an operation o on state s if its downstream precondition holds, otherwise, it returns the state unchanged.

Listing 6.6: Polymorphic `CmRDT` trait to implement op-based CRDTs in VeriFx.

```
1  trait CmRDT[Op, Msg, T <: CmRDT[Op, Msg, T]] {
2    def prepare(op: Op): Msg
3    def effect(msg: Msg): T
4    def tryEffect(msg: Msg): T =
5      if (this.enabledDown(msg)) this.effect(msg)
6      else this.asInstanceOf[T]
7    // by default all states are considered reachable
8    def reachable(): Boolean = true
9    // by default all operations can occur concurrently
10   def canConcur(x: Msg, y: Msg): Boolean = true
11   // by default all states are compatible
12   def compatible(that: T): Boolean = true
13   // by default there are no source preconditions
14   def enabledSrc(op: Op): Boolean = true
15   // by default there are no downstream preconditions
16   def enabledDown(msg: Msg): Boolean = true
17   def equals(that: T): Boolean = this == that
18 }
```

Listing 6.6 shows the `CmRDT` trait that can be used to implement op-based CRDTs. The implementing CRDT needs to provide concrete type arguments for the supported operations, the exchanged messages, and the CRDT type itself. Every CRDT that extends the `CmRDT` trait must implement the `prepare` and `effect` methods. The `tryEffect` method has a default implementation that applies the operation if its downstream

precondition holds, otherwise, it returns the state unchanged. By default, all states are considered reachable, all operations are enabled at the source and downstream, all operations can occur concurrently, and all states are compatible. Some CmRDTs make other assumptions that can be encoded by overriding the appropriate method. For example, in an Observed-Removed Set [Sha+11a] it is not possible to delete tags that are added concurrently; this can be encoded by overriding `canConcur`. VeriFx only considers concurrent operations that fulfill the `canConcur` predicate. Hence, for the Observed-Removed Set CRDT, VeriFx will not check concurrent add and delete operations that have tags in common.

Listing 6.7: Trait to verify CmRDTs in VeriFx.

```
1  trait CmRDTProof[Op, Msg, T <: CmRDT[Op, Msg, T]] {
2    proof is_a_CmRDT {
3      forall (s1: T, s2: T, s3: T, x: Op, y: Op) {
4        // Apply operations x and y concurrently
5        // replica s1 locally invokes operation x
6        val msg1 = s1.prepare(x)
7        // replica s2 locally invokes operation y
8        val msg2 = s2.prepare(y)
9
10       (s1.reachable() && s2.reachable() && s3.reachable() &&
11        s1.enabledSrc(x) && s2.enabledSrc(y) &&
12        s1.canConcur(msg1, msg2) && s1.compatibleS(s2) &&
13        s1.compatibleS(s3) && s2.compatibleS(s3)) =>: {
14           // The effectors must commute
15           s3.tryEffect(msg1).tryEffect(msg2)
16             .equals(s3.tryEffect(msg2).tryEffect(msg1)) &&
17           s3.tryEffect(msg1).reachable() &&
18           s3.tryEffect(msg2).reachable() &&
19           s3.tryEffect(msg1).tryEffect(msg2).reachable()
20       }
21     }
22   }
23 }
```

Similar to CvRDTs, our CRDT library provides a `CmRDTProof` trait shown in Listing 6.7 to verify CmRDT implementations. This trait defines a general proof of correctness that checks that all operations commute based on the previously described formula. The library also provides several numbered variants of the trait to verify polymorphic CmRDTs.

6.4.1.3 Pure Operation-Based CRDTs

Pure operation-based CRDTs are a family of operation-based CRDTs that exchange only the operations instead of data-type specific messages constructed by the prepare phase. The effect phase stores incoming operations in a partially ordered log of (concurrent) operations. Queries are computed against the log and operations do not need to commute. Data-type-specific redundancy relations dictate which operations to store in the log and when to remove operations from the log.

VeriFx's CRDT library provides a `PureOpCRDT` trait which is shown in Listing 6.8 and is used for implementing pure op-based CRDTs. The trait extends the `CmRDT` trait since pure op-based CRDTs are a special variant of op-based CRDTs. Operations are tagged with a version vector and the messages constructed by the prepare phase are simply the tagged operations themselves (Line 14). The effect phase is also the same for every pure op-based CRDT; it removes the operations that are made redundant by the incoming operation (Line 16) and adds the incoming operation to the log if it is not redundant (Lines 17 to 21). Importantly, the trait also overrides the `compatible` predicate to state that concurrent operations must have concurrent version vectors (Lines 10 and 11).

Pure op-based CRDTs that extend the `PureOpCRDT` trait inherit the generic prepare and effect phase and only need to implement the data-type-specific redundancy relations: `selfRedundant` and `redundantBy`. Since `PureOpCRDTs` are also `CmRDTs` programmers can reuse the `CmRDTProof` traits to verify pure-op based CRDT implementations.

6.4.2 Operational Transformation Library

Recall from Section 2.3.2.1 that the Operational Transformation (OT) [EG89] approach applies operations locally and propagates them asynchronously to the other replicas. Incoming operations are transformed against previously executed concurrent operations such that the modified operation preserves the intended effect. Operations are functions from state to state: $Op : \Sigma \rightarrow \Sigma$ and are transformed using a type-specific transformation function $T : Op \times Op \rightarrow Op$. Thus, $T(o_1, o_2)$ denotes the operation that results from transforming o_1 against a previously executed concurrent operation o_2.

Listing 6.8: Traits to implement and verify pure op-based CRDTs in VeriFx.

```
class TaggedOp[Op](t: VersionVector, o: Op)
trait PureOpCRDT[Op, T <: PureOpCRDT[Op, T]] extends
    CmRDT[TaggedOp[Op], TaggedOp[Op], T] {
  val polog: Set[TaggedOp[Op]]
  def copy(newPolog: Set[TaggedOp[Op]]): T

  // Data-type-specific redundancy relations
  def selfRedundant(op: TaggedOp[Op]): Boolean
  def redundantBy(x: TaggedOp[Op], y: TaggedOp[Op]): Boolean

  override def compatible(x: TaggedOp[Op], y: TaggedOp[Op]) =
    x.t.concurrent(y.t)

  // Generic prepare and effect methods
  def prepare(o: TaggedOp[Op]): TaggedOp[Op] = o
  def effect(taggedOp: TaggedOp[Op]): T = {
    val prunedPolog = this.polog.filter((x: TaggedOp[Op]) =>
    !this.redundantBy(x, taggedOp))
    val newPolog: Set[TaggedOp[Op]] =
      if (this.selfRedundant(taggedOp))
        prunedPolog
      else
        prunedPolog.add(taggedOp)
    this.copy(newPolog)
  }
}
```

Ressel et al. [RNG96] proved that replicas eventually converge if the transformation function satisfies two transformation properties: TP_1 and TP_2. Property TP_1 states that any two enabled concurrent operations o_i and o_j must commute after transforming them:

$$\begin{aligned}&\forall o_i, o_j \in Op, \forall s \in \Sigma : \\ &\quad enabled(o_i, s) \land enabled(o_j, s) \land canConcur(o_i, o_j) \\ &\quad\quad \implies T(o_j, o_i)(o_i(s)) = T(o_i, o_j)(o_j(s))\end{aligned}$$

Property TP_2 states that given three enabled concurrent operations o_i, o_j, and o_k, the transformation of o_k does not depend on the order in which operations o_i and o_j are transformed:

$$\begin{aligned}&\forall o_i, o_j, o_k \in Op, \forall s \in \Sigma : \\ &\quad enabled(o_i, s) \land enabled(o_j, s) \land enabled(o_k, s) \land \\ &\quad canConcur(o_i, o_j) \land canConcur(o_j, o_k) \land canConcur(o_i, o_k) \\ &\quad\quad \implies T(T(o_k, o_i), T(o_j, o_i)) = T(T(o_k, o_j), T(o_i, o_j))\end{aligned}$$

Note that properties TP_1 and TP_2 only need to hold for states in which the operations can be generated, represented by the relation $enabled : Op \times \Sigma \to \mathbb{B}$, and only if the two operations can occur concurrently, represented by the relation $canConcur : Op \times Op \to \mathbb{B}$.

VeriFx provides a library for implementing and verifying RDTs that use operational transformations. Programmers can build custom RDTs by extending the `OT` trait shown in Listing 6.9. Every RDT that extends the `OT` trait must provide concrete type arguments for the state and operations, and implement the `transform` and `apply` methods. The `transform` method transforms an incoming operation against a previously executed concurrent operation. The `apply` method applies an operation on the state. By extending this trait, the RDT inherits proofs for TP_1 and TP_2. These proofs assume that all operations are enabled and that all operations can occur concurrently. If this is not the case, the RDT can override the `enabled` and `canConcur` methods respectively.

Although VeriFx supports the general execution model of OT, most transformation functions described by the literature were specifically designed for collaborative text editing. They model text documents as a sequence of characters and operations insert or delete characters at a given position in the document. OT papers thus describe four transformations functions, one for every pair of operations: insert-insert, insert-delete, delete-insert, and delete-delete.

Listing 6.9: Polymorphic `OT` trait to implement and verify RDTs using operational transformation in VeriFx.

```
1  trait OT[State, Op] {
2    def transform(x: Op, y: Op): Op
3    def apply(state: State, op: Op): State
4
5    // by default operations are enabled on all states
6    def enabled(op: Op, state: State): Boolean = true
7    // by default all operations can be concurrent
8    def canConcur(x: Op, y: Op): Boolean = true
9
10   proof TP1 {
11     forall (opI: Op, opJ: Op, st: State) {
12       (this.enabled(opI, st) && this.enabled(opJ, st) &&
13        this.canConcur(opI, opJ)) =>: {
14         this.apply(this.apply(st, opI),
15                    this.transform(opJ, opI)) ==
16         this.apply(this.apply(st, opJ),
17                    this.transform(opI, opJ))
18       }
19     }
20   }
21
22   proof TP2 {
23     forall (opI: Op, opJ: Op, opK: Op, st: State) {
24       (this.enabled(opI, st) && this.enabled(opJ, st) &&
25        this.enabled(opK, st) && this.canConcur(opI, opJ) &&
26        this.canConcur(opJ,opK) && this.canConcur(opI,opK)) =>: {
27         this.transform(this.transform(opK, opI),
28                        this.transform(opJ, opI)) ==
29         this.transform(this.transform(opK, opJ),
30                        this.transform(opI, opJ))
31       }
32     }
33   }
34 }
```

Likewise, VeriFx's OT library provides a `ListOT` trait that models the state as a list of values and supports insertions and deletions. Listing 6.10 shows the implementation of the `ListOT` trait which extends the `OT` trait and is polymorphic in the type of values it stores (in practice, collaborative text editors store characters). RDTs extending the `ListOT` trait need to implement four methods (`Tii`, `Tid`, `Tdi`, `Tdd`) corresponding to the transformation functions for transforming insertions against insertions (`Tii`), insertions against deletions (`Tid`), deletions against insertions (`Tdi`), and deletions against deletions (`Tdd`). The trait provides a default implementation of `transform` that dispatches to the corresponding transformation function based on the type of operations, and a default implementation of `apply` that inserts or deletes a value from the underlying list.

Listing 6.10: Polymorphic `ListOT` trait to implement and verify OT functions for collaborative text editing.

```
trait ListOT[V, Op] extends OT[List[V], Op] {
  def isInsert(x: Op): Boolean; def isDelete(x: Op): Boolean
  def getPosition(x: Op): Int; def getValue(x: Op): V

  def Tii(x: Op, y: Op): Op // required
  def Tid(x: Op, y: Op): Op // required
  def Tdi(x: Op, y: Op): Op // required
  def Tdd(x: Op, y: Op): Op // required

  def transform(x: Op, y: Op): Op = {
    if (this.isInsert(x) && this.isInsert(y))
      this.Tii(x, y)
    else if (this.isInsert(x) && this.isDelete(y))
      this.Tid(x, y)
    else if (this.isDelete(x) && this.isInsert(y))
      this.Tdi(x, y)
    else if (this.isDelete(x) && this.isDelete(y))
      this.Tdd(x, y)
    else x // must be an identity operation
  }
  def apply(lst: List[V], op: Op): List[V] = {
    if (this.isInsert(op)) {
      val pos = this.getPosition(op)
      val char = this.getValue(op)
      lst.insert(pos, char)
    }
    else if (this.isDelete(op)) {
      val pos = this.getPosition(op)
      lst.delete(pos)
    }
    else lst // identity operation
  }
  // ...
}
```

6.4.3 Encoding RDT-Specific Assumptions

It is not uncommon for RDTs to assume causal delivery of operations but VeriFx (and its CRDT and OT libraries) does not make any assumptions. Specific assumptions must either be guaranteed by the RDT's implementation or be explicitly encoded in the proofs.

For example, as mentioned in Section 4.5.2.1, the Observed-Removed Set CRDT [Sha+11a] assumes that 1) replicas can generate globally unique tags, and 2) add and remove operations of the same element are delivered in causal order. As a result of these assumptions, replicas cannot remove a tag and concurrently add the same tag. The former assumption can be guaranteed by the RDT implementation if every replica has a unique ID that is combined with a local counter that increases monotonically to generate unique tags. The latter assumption that results from causal delivery can be explicitly encoded in the proof. However, programmers need to be careful when encoding assumptions explicitly in proofs because they are not checked. To remove the latter assumption, one could model the underlying causal communication protocols in VeriFx.

Listing 6.11 shows an excerpt from the implementation of the Observed-Removed Set CRDT [Sha+11a] in which we override the `compatible` predicate (Line 11 and 12) to encode the fact that replicas have unique IDs, and we override the `canConcur` predicate (Line 13 to 27) such that the proof does not consider `add` and `remove` operations if the tag generated by `add` is contained in the set of tags that are removed (because causal delivery precludes `remove` from having observed that tag). This example demonstrates how to use the predicates provided by VeriFx's CRDT and OT libraries to encode RDT-specific assumptions.

6.5 Evaluation

We now evaluate the applicability of VeriFx to implement and verify RDTs. Our evaluation is twofold. First, we implement and verify an extensive portfolio comprising 37 CRDTs which were taken from literature [Sha+11a; BAS17; Sha17; Bie+12; Kle22] and industrial databases [Akk; Anta; Antb]. To the best of our knowledge, we are the first to mechanically verify all CRDTs from Shapiro et al. [Sha+11a], the pure op-based CRDTs from Baquero et al. [BAS17], and the map CRDTs from Kleppmann [Kle22]. Afterward, we reproduce a study [Imi+03] on

Listing 6.11: Excerpt from the implementation of the OR-Set CRDT [Sha+11a] in VeriFx.

```
class Tag[ID](replica: ID, counter: Int)
enum SetOp[V, ID] { Add(e: V) | Remove(e: V) }
enum SetMsg[V, ID] {
  AddMsg(e: V, tag: Tag[ID]) |
  RemoveMsg(e: V, tags: Set[Tag[ID]])
}
class ORSet[V, ID]
    (id: ID, counter: Int, elements: Map[V, Set[Tag[ID]]])
    extends CmRDT[SetOp[V, ID], SetMsg[V, ID], ORSet[V, ID]] {
  // ...
  override def compatible(that: ORSet[V, ID]) =
    this.id != that.id // replicas have unique IDs
  override def canConcur(x: SetMsg[V, ID], y: SetMsg[V, ID]) =
    x match {
      case AddMsg(_, tag) =>
        y match {
          case AddMsg(_, _) => true
          // tag cannot be in tags because of causal delivery
          case RemoveMsg(_, tags) => !tags.contains(tag)
        }
      case RemoveMsg(_, tags) =>
        y match {
          // tag cannot be in tags because of causal delivery
          case AddMsg(_, tag) => !tags.contains(tag)
          case RemoveMsg(_, _) => true
        }
}   }
```

the correctness of OT functions. To this end, we verify 5 well-known operational transformation functions for collaborative text editing. Moreover, we verify 4 unpublished OT designs for replicated registers and stacks.

6.5.1 Methodology

All experiments reported in this section were conducted on AWS using an m5.xlarge VM with 4 virtual CPUs and 16 GiB of RAM. VeriFx uses Z3 version 4.8.14.0 to discharge the necessary proof obligations. As in previous chapters, we implemented all benchmarks with JMH [Ope]. We configured JMH to execute 20 warmup iterations followed by 20 measurement iterations for every benchmark. To avoid run-to-run variance JMH repeats every benchmark in 3 fresh JVM forks, yielding a total of 60 samples per benchmark.

6.5.2 Verifying Conflict-free Replicated Data Types

Table 6.1 depicts the 37 CRDTs we implemented and verified in VeriFx. For each CRDT, the table shows the type of CRDT, the code size, and the average verification time. The table features five CRDTs that were adapted from the original specification (marked with an @) and two that are new. For example, the state-based 2P2P Graph is an adaptation from the operation-based 2P2P Graph specification found in [Sha+11a].

VeriFx was able to verify all CRDTs except the Replicated Growable Array (RGA) [Sha+11a] due to the recursive nature of the insertion algorithm. We found that the Two-Phase Set CRDT (described in Section 6.1) converges but is not functionally correct, that the original Map CRDT proposed by Kleppmann [Kle22] diverges as VeriFx found the same counterexample as described in their technical report, and that the Molli, Weiss, Skaf (MWS) Set is incomplete. In the remainder of this section, we first describe the implementation and verification of the MWS Set, and afterward, focus on the map CRDTs from [Kle22].

6.5.2.1 MWS Set

Algorithm 8 describes the MWS Set, which associates a count to every element. An element is considered in the set if its count is strictly positive. `remove` decreases the element's count, while `add` increments the count by

CRDT	Type	LoC	Correct	Time	Source
Counter	O	17	✓	3.2 s	[Sha+11a]
Grow-Only Counter	S	33	✓	4.3 s	[Sha+11a]
Positive-Negative Counter	S	15	✓	5.9 s	[Sha+11a]
Dynamic PN-Counter	S	41	✓	7.1 s	[Akk]
Enable-Wins Flag	P	18	✓	4.0 s	[BAS17]
Enable-Wins Flag	O	44	✓	3.6 s	[Antb]
Disable-Wins Flag	P	20	✓	3.9 s	[BAS17]
Disable-Wins Flag	O	50	✓	3.8 s	[Anta]
Multi-Value Register	S	63	✓	8.8 s	[Sha+11a]
Multi-Value Register	P	18	✓	4.1 s	[BAS17]
Last-Writer-Wins Register	S	16	✓	5.3 s	[Sha+11a]
Last-Writer-Wins Register	O	38	✓	4.4 s	[Sha+11a]
Grow-Only Set	S	10	✓	5.3 s	[Sha+11a]
Two-Phase Set	O	27	✓	4.4 s	[Sha+11a]
Two-Phase Set	S	26	✗	6.3 s	[Sha+11a]
Unique Set	O	39	✓	4.4 s	[Sha+11a]
Add-Wins Set	P	28	✓	4.3 s	[BAS17]
Remove-Wins Set	P	42	✓	4.5 s	[BAS17]
Last-Writer-Wins Set	S	36	✓	6.6 s	[Sha+11a]
Optimized LWW-Set	S	37	✓	6.5 s	new
Positive-Negative Set	S	36	✓	9.6 s	[Sha+11a]
Observed-Removed Set	O	75	✓	6.2 s	[Sha+11a]
Observed-Removed Set	S	34	✓	7.6 s	[Sha17]
Optimized OR-Set	S	78	✓	30.2 s	[Bie+12]
Molli, Weiss, Skaf Set	O	45	✓	4.7 s	ⓘ [Sha+11a]
Grow-Only Map	S	32	✓	9.1 s	new
Buggy Map	O	87	✗	65.2 s	[Kle22]
Corrected Map	O	101	✓	49.4 s	[Kle22]
2P2P Graph	O	58	✓	7.8 s	[Sha+11a]
2P2P Graph	S	41	✓	10.7 s	ⓐ [Sha+11a]
Add-Only DAG	O	42	✓	4.7 s	[Sha+11a]
Add-Only DAG	S	30	✓	8.7 s	ⓐ [Sha+11a]
Add-Remove Partial Order	O	61	✓	10.4 s	[Sha+11a]
Add-Remove Partial Order	S	49	✓	13.2 s	ⓐ [Sha+11a]
Replicated Growable Array	O	156	◷	/	[Sha+11a]
Continuous Sequence	O	108	✓	9.2 s	ⓐ [Sha+11a]
Continuous Sequence	S	53	✓	11.4 s	ⓐ [Sha+11a]

Table 6.1: Verification results for CRDTs implemented and verified in VeriFx. S = state-based, O = op-based, P = pure op-based CRDT. ◷ = timeout, ⓐ = adaptation of an existing CRDT, ⓘ = incomplete definition.

the amount that is needed to make it positive (or by 1 if it is already positive). Listing 6.12 shows the implementation of the MWS Set in VeriFx as a polymorphic class that extends the `CmRDT` trait (cf. Section 6.4.1.2). The type arguments passed to `CmRDT` correspond to the supported operations (`SetOp`s), the messages that are exchanged (`SetMsg`s), and the CRDT type itself (`MWSSet`). The `SetOp` enumeration defines two types of operations: `Add(e)` and `Remove(e)`.

Algorithm 8 Op-based MWS Set CRDT taken from [Sha+11a].

payload set $S = \{(element, count), \ldots\}$
  initial $E \times \{0\}$
query *lookup* (element e) : boolean b
  let $b = ((e, k) \in S \wedge k > 0)$
update *add* (element e)
  atSource (e) : integer j
    if $\exists (e, k) \in S : k \leq 0$ **then**
      let $j = |k| + 1$
    else
      let $j = 1$
  downstream (e, j)
    let $k' : (e, k') \in S$
    $S := S \backslash \{(e, k')\} \cup \{(e, k' + j)\}$
update *remove* (element e)
  atSource (e)
    pre *lookup*(e)
  downstream (e)
    $S := S \backslash \{(e, \mathbf{k'})\} \cup \{(e, \mathbf{k'} - 1)\}$

Algorithm 9 Remove with k' defined at source.

update *remove* (element e)
  atSource (e) : integer k'
    pre *lookup*(e)
    let $k' : (e, k') \in S$
  downstream (e, k')
    $S := S \backslash \{(e, k')\} \cup \{(e, k' - 1)\}$

Algorithm 10 Remove with k' defined in downstream.

update *remove* (element e)
  atSource (e)
    pre *lookup*(e)
  downstream (e)
    let $k' : (e, k') \in S$
    $S := S \backslash \{(e, k')\} \cup \{(e, k' - 1)\}$

Listing 6.12: MWS Set implementation in VeriFx.

```
enum SetOp[V] { Add(e: V) | Remove(e: V) }
enum SetMsg[V] { AddMsg(e: V, dt: Int) | RmvMsg(e: V) }
class MWSSet[V](elements: Map[V, Int]) extends CmRDT[SetOp[V],
    SetMsg[V], MWSSet[V]] {
  override def enabledSrc(op: SetOp[V]) = op match {
    case Add(_) => true
    case Remove(e) => this.preRemove(e)
  }
  def prepare(op: SetOp[V]) = op match {
    case Add(e) => this.add(e)
    case Remove(e) => this.remove(e)
  }
  def effect(msg: SetMsg[V]) = msg match {
    case AddMsg(e, dt) => this.addDownstream(e, dt)
    case RmvMsg(e) => this.removeDownstream(e)
  }
  def lookup(e: V) = this.elements.getOrElse(e, 0) > 0
  def add(e: V): SetMsg[V] = {
    val count = this.elements.getOrElse(e, 0)
    val dt = if (count <= 0) (count * -1) + 1 else 1
    new AddMsg(e, dt)
  }
  def addDownstream(e: V, dt: Int): MWSSet[V] = {
    val count = this.elements.getOrElse(e, 0)
    new MWSSet(this.elements.add(e, count + dt))
  }
  def preRemove(e: V) = this.lookup(e)
  def remove(e: V): SetMsg[V] = new RmvMsg(e)
  def removeDownstream(e: V): MWSSet[V] = {
    val kPrime = ??? // undefined in Algorithm 8
    new MWSSet(this.elements.add(e, kPrime - 1))
  }
}
object MWSSet extends CmRDTProof1[SetOp,SetMsg,MWSSet]
```

Listing 6.13: Computing k' at the source.

```
def remove(e: V): Tuple[V, Int] =
  new Tuple(e, this.elements.getOrElse(e, 0))
def removeDown(tup: Tuple[V, Int]): MWSSet[V] = {
  val e = tup.fst; val kPrime = tup.snd
  new MWSSet(this.elements.add(e, kPrime - 1))
}
```

Listing 6.14: Computing k' downstream.

```
def remove(e: V): V = e
def removeDown(e: V): MWSSet[V] = {
  val kPrime = this.elements.getOrElse(e, 0)
  new MWSSet(this.elements.add(e, kPrime - 1))
}
```

The `MWSSet` class has a field, called `elements`, that maps elements to their count (Line 3). Like all op-based CRDTs, the `MWSSet` implements two phases: `prepare` and `effect`. The `prepare` method pattern matches on the operation and delegates it to the corresponding source method which prepares a `SetMsg` message to be broadcast to all replicas. The class overrides the `enabledSrc` method to implement the source precondition on the `remove` method, as defined by Algorithm 8. When replicas receive incoming messages, they are processed by the `effect` method which delegates them to the corresponding downstream method which performs the actual update. For example, the `removeDownstream` method processes incoming `RmvMsg` messages by decreasing some count k' by 1. Unfortunately, k' is undefined in Algorithm 8.

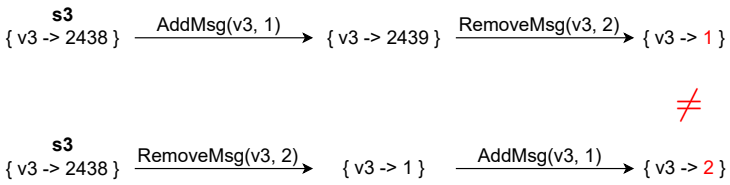
We believe that k' is either defined by the source replica and included in the propagated message (Algorithm 9), or, k' is defined as the element's count at the downstream replica (Algorithm 10). We implemented both possibilities in VeriFx (Listings 6.13 and 6.14) and verified them to find out which one, if any, is correct. To this end, the companion object of the `MWSSet` class (cf. Line 33 in Listing 6.12) extends the `CmRDTProof1` trait (cf. Section 6.4.1.2), passing along three type arguments: the type of operations `SetOp`, the type of messages being exchanged `SetMsg`, and the CRDT type constructor `MWSSet`. The object extends `CmRDTProof1` as the `MWSSet` class is polymorphic and expects one type argument. When executing the proof inherited by the companion object, VeriFx automatically proves that the possibility implemented by Listing 6.14 is correct and that the one of Listing 6.13 is wrong.

Figure 6.8 shows the counterexample returned by VeriFx for the implementation from Listing 6.13 together with a visualization of the divergence problem. This problem manifests when replicas concurrently add and remove the same element. Recall that a counterexample is a mapping from variables (defined by the proof) to values that break the proof. Since the MWS Set is an operation-based CRDT we used the `CmRDTProof1` trait (cf. Section 6.4.1.2). This proof defines three variables `s1`, `s2`, and `s3` that represent the state of the replicas, and two variables `x` and `y` that represent concurrent operations generated by replicas `s1` and `s2` respectively.

The counterexample found by VeriFx is shown in Fig. 6.8a. It consists of a replica `s1` that generates an add operation `x = Add(v3)` and prepares a message `AddMsg(v3, 1)` to be broadcast. Concurrently, replica

```
// Definition of the type parameter S
enum S { v0 | v1 | v2 | v3 | v4 }
// Variable assignments that cause divergence
val s1 = MWSSet(Map(v3 -> 8856, ...))
val s2 = MWSSet(Map(v3 -> 2, ...))
val s3 = MWSSet(Map(v3 -> 2438, ...))
val x = Add(v3) // operation generated by replica s1
// The prepare phase constructs the following message:
// s1.prepare(x) = AddMsg(v3, 1)
val y = Remove(v3) // operation generated by replica s2
// s2.prepare(y) = RemoveMsg(v3, 2)
```

(a) Counterexample returned by VeriFx.



(b) Visualization of the counterexample returned by VeriFx.

Figure 6.8: Counterexample for the MWS Set, found by VeriFx.

`s2` generates a remove operation `y = Remove(v3)` and prepares a message `RemoveMsg(v3, 2)` to be broadcast. The `RemoveMsg` contains 2 because that is the local count that `s2` stores for key `v3`. Eventually, every replica receives the broadcasted messages, possibly in a different order, and processes them. Figure 6.8b shows that depending on the order in which `s3` processes the messages, the outcome is different. The reason for this divergence problem is that `add` increments the local count by a given value (cf. Listing 6.12) but `remove` overrides the local count with the incoming count minus one (cf. Listing 6.13). Thus, depending on the order in which those operations are applied the elements will have a different count.

Listing 6.14 shows a modified implementation of the `remove` operation which decrements the local count instead of overriding the local count with the incoming count minus one. VeriFx proves that this implementation guarantees convergence. We thus successfully completed the MWS Set implementation thanks to VeriFx's integrated verification capabilities. The counterexample returned by VeriFx was crucial to understand the problem and find an appropriate solution.

6.5.2.2 A Buggy Map CRDT

Kleppmann [Kle22] describes the specification of an operation-based Map CRDT which he believed to be "obviously correct" only to find out it contains a bug that causes divergence after spending hours trying to verify it. He then tweeted the buggy pseudo code of the Map CRDT and challenged his 29400 followers (mainly software engineers) to find the bug. Although I came close, I could not identify the precise timestamp conditions under which the bug would occur. Only Sreeja Nair (which at the time was also a PhD student doing research on RDTs) was able to manually identify the bug. Kleppmann later tweeted a variation on the algorithm: "Here is a variant of the algorithm that is correct (I believe)".

Fortunately, we now have VeriFx to help us *automatically* verify both the buggy map CRDT and the corrected map CRDT, which until now had not been verified. We implemented and verified both map CRDTs in an afternoon. The complete implementation and verification of the buggy map CRDT is explained in Appendix I. We now present the key takeaways from our experience implementing and verifying these map CRDTs.

Implementation. The implementation of the map CRDTs mainly consisted of translating the mathematical specifications to VeriFx. We introduced slight changes to the design to improve efficiency. For example, the specification keeps a set of triples where each triple holds a key, a value, and a timestamp. Since every key appears at most in one triple, our implementation uses a dictionary to efficiently map keys to their value and timestamp.

Verification. After implementing the buggy map CRDT, we proceeded to its automated verification but VeriFx generated invalid counterexamples. For instance, one in which two distinct replicas generated the same timestamp. This is not possible because the design assumes that replicas have unique IDs and combine them with Lamport clocks [Lam78] to generate unique timestamps. However, VeriFx does not know this assumption nor does it know the relation between a replica's clock and the values it observed. In practice, many CRDTs make similar *implicit* assumptions which is the reason they are complex and difficult to get right.

Thus, in order to verify the buggy map CRDT we had to encode all assumptions *explicitly* such that VeriFx does not analyze impossible cases. Naturally, we were not able to distill all assumptions from the first time. Instead, VeriFx kept returning invalid counterexamples which helped us

find and formulate the missing assumptions. Listing I.2 in Appendix I.3 shows the encoding of these assumptions.

Counterexample. After defining all assumptions, VeriFx found a valid counterexample for the buggy map CRDT that is equivalent to the one found manually by Nair [Kle22]. It consists of a corner case in which the `Put` and `Delete` operations do not commute and thus may cause replicas to diverge. We explain the complete counterexample in Appendix I.3.

Corrected Map CRDT. After finding the counterexample for the buggy map CRDT, we also verified the corrected map CRDT from [Kle22]. This did not require additional efforts since we already distilled all assumptions for the buggy map CRDT. VeriFx automatically proved that the corrected design indeed guarantees convergence, which to the best of our knowledge, is the first mechanical proof of correctness for this CRDT.

As shown in Table 6.1, the verification times for the buggy and corrected map CRDTs are slightly higher compared to the other CRDTs we verified, but are still very fast for a fully automated verification approach. The higher verification times come from the fact that these map CRDTs are too complex to directly prove convergence of all operation pairs. Hence, we introduce a subproof for every operation pair. The total verification time is the sum of the verification times of the subproofs.

6.5.2.3 Conclusion

We verified a diverse and extensive portfolio of CRDTs, including optimized designs that are representative of real-world CRDTs used in industrial-strength databases. For example, the Last-Writer-Wins Register, Multi-Value Register, PN-Counter, Enable-Wins and Disable-Wins Flags, Grow-Only Map, Grow-Only Set, Observed-Removed Set, and Remove-Wins Set are used in AntidoteDB[5]. The Dynamic Positive-Negative Counter supports a dynamic number of replicas and is similar to the one that is implemented in Akka's distributed key-value store. The Riak key-value database[6] implements several CRDTs (counters, registers, sets, etc.) based on the specifications of Shapiro et al. [Sha+11a] which we also implemented and verified. Thus, we conclude that VeriFx is suited to verify CRDTs since all implementations were verified in a matter of seconds (cf. Table 6.1).

[5] `https://www.antidotedb.eu/`

[6] `https://riak.com/products/riak-kv`

6.5.3 Verifying Operational Transformation

We now show that VeriFx is general enough to verify other RDT families such as Operational Transformation (OT) [EG89]. We implemented all transformation functions for collaborative text editing described by Imine et al. [Imi+03] and verified TP_1 and TP_2 in VeriFx.

Transformation Function	**LoC**	**Properties**		**Time**	
		TP_1	TP_2	TP_1	TP_2
Ellis and Gibbs [EG89]	84	✗	✗	115 s	29 s
Ressel et al. [RNG96]	78	✓	✗	68 s	30 s
Sun et al.[Sun+98]	68	✗	✗	321 s	13 s
Suleiman et al. [SCF97]	85	✗	✗	34 s	40 s
Imine et al. [Imi+03]	83	✓	✗	61 s	17 s
$\text{Register}_{\text{v1}}$ [Imi22]	6	✗	✓	3 s	3 s
$\text{Register}_{\text{v2}}$ [Imi22]	6	✓	✗	3 s	3 s
$\text{Register}_{\text{v3}}$ [Imi22]	7	✓	✓	3 s	3 s
Stack [Imi22]	47	✗	✓	5 s	5 s

Table 6.2: Verification results of OT functions in VeriFx.

Table 6.2 summarizes the verification results. For each transformation function, the table shows the code size, whether or not it satisfies TP_1 and TP_2, and the average verification time. As can be observed from the table, the functions proposed by Ellis and Gibbs [EG89], Sun et al. [Sun+98], and Suleiman et al. [SCF98] do not satisfy TP_1 nor TP_2. Ressel et al.'s functions [RNG96] satisfy TP_1 but not TP_2. The transformation functions proposed by Imine et al. [Imi+03] also do not satisfy TP_2, which confirms the findings of Li and Li [LL04] and Oster et al. [Ost+06]. In addition, in a private communication [Imi22], Imine asked us to verify (unpublished) OT designs for replicated registers and stacks. Out of the three register designs verified in VeriFx, only one was correct for both TP_1 and TP_2. Regarding the stack design, it guarantees TP_2 but not TP_1. VeriFx provided meaningful counterexamples for each incorrect design.

To exemplify our approach to verifying OT, the remainder of this section reports on the implementation and verification of Imine et al.'s transformation functions [Imi+03] in VeriFx.

6.5.3.1 The IMOR Transformation Functions

Listing 6.15: Excerpt from the implementation of Imine et al.'s transformation functions [Imi+03] in VeriFx.

```
1  enum Op { Ins(p: Int, ip: Int, c: Int) | Del(p: Int) | Id() }
2  object Imine extends ListOT[Int, Op] {
3    def Tii(x: Ins, y: Ins) = {
4      val p1 = x.p; val ip1 = x.ip; val c1 = x.c
5      val p2 = y.p; val ip2 = y.ip; val c2 = y.c
6      if (p1 < p2) x
7      else if (p1 > p2) new Ins(p1 + 1, ip1, c1)
8      else if (ip1 < ip2) x
9      else if (ip1 > ip2) new Ins(p1+1, ip1, c1)
10     else if (c1 < c2) x
11     else if (c1 > c2) new Ins(p1+1, ip1, c1)
12     else new Id()
13   }
14   def Tid(x: Ins, y: Del) =
15     if (x.p > y.p) new Ins(x.p - 1, x.ip, x.c)
16     else x
17   def Tdi(x: Del, y: Ins) =
18     if (x.p < y.p) x else new Del(x.p + 1)
19   def Tdd(x: Del, y: Del) =
20     if (x.p < y.p) x
21     else if (x.p > y.p) new Del(x.p - 1)
22     else new Id()
23 }
```

Listing 6.15 shows the implementation of Imine et al.'s transformation functions [Imi+03] in VeriFx. These transformation functions are often referred to as "IMOR" based on the authors' initials. The enumeration `Op` on Line 1 defines the three supported operations:

- `Ins(p, ip, c)` represents the insertion of character `c` at position `p`. Initially, the character[7] was inserted at position `ip`. Transformations may change `p` but leave `ip` untouched.
- `Del(p)` represents the deletion of the character at position `p`.
- `Id()` acts as a no-op. This operation is never issued by users directly but operations may be transformed to a no-op.

The object `Imine` extends the `ListOT` trait and implements the four transformation functions (`Tii`, `Tid`, `Tdi`, `Tdd`) that are required for collabora-

[7] We represent characters using integers that correspond to their ASCII code.

tive text editing (cf. Section 6.4.2). The implementation of these transformation functions is a straightforward translation from their description by Imine et al. [Imi+03]. The resulting object inherits automated proofs for TP_1 and TP_2. When running these proofs, VeriFx reports that the transformation functions guarantee TP_1 but not TP_2.

6.5.3.2 Conclusion

Based on our experience verifying OT functions and the results shown in Table 6.2, we conclude that VeriFx is suited to verify other RDT families such as OT. Due to the number of cases that have to be considered, the verification times are longer than for CRDTs but are still acceptable for static verification [Cal+15].

6.6 Notes on Related Work

Program verification is a vast area of research. We structure this discussion on related work in four parts. First, we discuss related work on verification languages. Second, we focus on the verification of CRDTs. Third, we review approaches to verify invariants in distributed systems. Finally, we discuss related work on the verification of OT.

6.6.1 Verification Languages

We reviewed existing verification languages in Section 2.3.1 and classified them into three categories: interactive, auto-active, and automated verification languages [LM10]. We now compare VeriFx to the most relevant works in these categories of verification languages.

Vazou et al. [Vaz+17] introduce the idea of refinement reflection in Liquid Haskell, where user-defined functions are reflected in a decidable fragment of SMT logic and can be used in refinement types to express correctness properties. VeriFx leverages a similar idea of reflection where *every* construct of the base language and its collections are reflected in SMT logic such that arbitrary VeriFx programs can completely be reflected in the logic. The main difference is that Liquid Haskell requires programmers to express correctness properties using refinement types and *manually* write proofs as Haskell functions, whereas, VeriFx targets *automated* verification of user-defined correctness properties (expressed with

the proof construct) by leveraging specialized encodings of the language's constructs in SMT logic. Moreover, VeriFx targets an iterative programming style where incorrect designs are improved based on the returned counterexamples, whereas Liquid Haskell only raises a type error without providing additional information.

Auto-active verification languages like Dafny [Lei10] and Spec# [BLS05] verify programs for runtime errors and user-defined invariants based on annotations provided by the programmer (e.g. preconditions, postconditions, loop invariants, etc.). IVLs like Boogie [Bar+06] and Why3 [FP13] automate the proof task by generating VCs from source code and discharging them using one or more SMT solvers but are not used directly by programmers. While the aforementioned approaches aim to be general such that they can be used to prove any property of a program, VeriFx was designed to be a high-level programming language capable of verifying RDTs *fully automatically*.

Also related to VeriFx is the work by Kaki and Jagannathan [KJ14] which consists of an automated verification framework integrated in a refinement type system. Programmers write relational specifications that define structural relations (using relational algebra) for the data type at hand and express correctness properties as refinement types atop operations. However, writing relational specifications for advanced data types is non-trivial and can be rather verbose, as noted by the authors themselves. In contrast, VeriFx allows programmers to write custom correctness properties directly as proofs in the language and thus does not require separate specifications, thereby, avoiding mismatches between the implementation and the verification, and simplifying software evolution.

6.6.2 Verifying Conflict-free Replicated Data Types

As mentioned in the introduction of this chapter, Gomes et al. [Gom+17] and Zeller et al. [ZBP14] propose formal frameworks to mechanically verify SEC for CRDT implementations but these are not automated and require significant efforts from the programmer.

Liu et al. [Liu+20] extend Liquid Haskell with typeclass refinements and use them to prove SEC for some of their own CRDTs. While simple proofs can be discharged automatically by the underlying SMT solver, advanced CRDTs also require significant proof efforts (as discussed in Section 6.1). In contrast, we fully automatically verified over 30 well-known

CRDTs in VeriFx. Automated verification is enabled by efficiently encoding all functional collections and their operations using the combinatory array logic for SMT solvers, whereas the reflection of those Liquid Haskell functions in SMT logic is recursive which breaks automated verification.

Liang and Feng [LF21] propose a new correctness criterion for CRDTs that extends SEC with functional correctness. While their focus is on *manual* verification of functional correctness, VeriFx focuses on *automated* verification of SEC.

Wang et al. [Wan+19] propose replication-aware linearizability, a criterion that enables sequential reasoning to verify CRDT implementations. The authors manually encoded the CRDTs in Boogie [Bar+06] to prove correctness. Those encodings are non-trivial and differ from real-world CRDT implementations. In contrast, VeriFx verifies high-level CRDT implementations directly.

Nagar and Jagannathan [NJ19] developed a proof rule that is parametrized by the consistency model and automatically checks convergence for CRDTs. Unfortunately, their framework introduces imprecisions which may lead to correct CRDTs being rejected. Moreover, their framework requires a first-order logic specification of the CRDT which is cumbersome and error-prone. The resulting proofs thus verify the specification instead of a concrete implementation. In contrast, VeriFx verifies CRDT implementations directly, thereby avoiding mismatches between the specification and the implementation and fostering code evolution.

6.6.3 Verifying Invariants of Replicated Data

Maintaining application-specific invariants under weak consistency is difficult. As explained in Section 2.3.3, invariant confluent operations [Bai+14] maintain application invariants without coordination. Follow-up work [WH18] devised an automated decision procedure to verify if operations are invariant confluent.

Some works have focused on verifying program invariants for existing RDTs. Soteria [NPS20] verifies program invariants for state-based replicated objects based on the invariant confluence criterion. Repliss [ZBP20] verifies program invariants for applications that are built on top of their CRDT library. CISE [Got+16] proposes a proof rule to check that a particular choice of consistency for the operations preserves the application's invariants. IPA [Bal+18] detects invariant-breaking operations and

proposes modifications to the operations in order to preserve application-specific invariants. Unfortunately, these approaches assume that the underlying RDT is correct. VeriFx enables programmers to verify that this is the case. In this dissertation, we did not verify application invariants and leave it as future work.

6.6.4 Verifying Operational Transformation

Ellis and Gibbs [EG89] first proposed an algorithm for operational transformation together with a set of transformation functions. Several works [SCF98; Sun+98] showed that integration algorithms like adOPTed [RNG96], SOCT2 [SCF98], and GOTO [SE98] guarantee convergence if the transformation functions satisfy the TP_1 and TP_2 properties. Unfortunately, Ellis and Gibbs' transformation functions [EG89] do not satisfy these properties [Sun+98; RNG96; SCF98].

Over the years, several transformation functions were proposed [RNG96; Sun+98; SCF97]. Imine et al. [Imi+03] used SPIKE, an automated theorem prover, to verify the correctness of these transformation functions and found counterexamples for all of them, except for Suleiman et al.'s transformation functions [SCF97] (which later were also shown to be wrong by Oster et al. [Ost+06]). As shown in Section 6.5.3, we were able to reproduce Imine et al.'s findings [Imi+03] using VeriFx and generate similar counterexamples. Imine et al. [Imi+03] also proposed a simpler set of transformation functions which later was found to also violate TP_2 [LL04; Ost+06]. VeriFx also found this counterexample.

6.7 Conclusion

Replicated Data Types (RDTs) are widespread among highly available distributed systems but verifying them remains complex, even for experts. Recently, automated verification efforts have been proposed [Liu+20; NJ19] but these cannot yet produce complete correctness proofs from high-level implementations.

In this chapter, we proposed VeriFx, a functional object-oriented programming language that features a novel proof construct to express correctness properties that are verified automatically. We leverage the proof construct to build libraries for implementing and verifying two well-known

families of RDTs: CRDTs [Sha+11b] and OT [EG89]. Programmers can also implement custom libraries to verify other approaches. Verified RDT implementations can be transpiled to mainstream languages, currently Scala and JavaScript. VeriFx's modular architecture allows programmers to add support for other languages.

By integrating automated verification in a high-level programming language, VeriFx empowers programmers to implement RDTs and seamlessly verify them. VeriFx detects divergence problems and returns high-level counterexamples which help correct the implementation. Since the verification process is automated and integrated into the language itself, programmers do not need expertise in formal verification. We thus successfully solved the verification problem outlined in Section 1.1.3.

Chapter 7

Conclusion

Throughout this dissertation we proposed principled approaches and developed novel programming languages for the development and verification of new and existing RDTs. This final chapter concludes this dissertation by summarizing our work. First, we revisit the problem statement for the development of RDTs. Afterward, we provide an overview of our approach and restate our contributions. We then identify promising avenues for future research and conclude this dissertation with some final remarks.

7.1 Programming Replicated Data Types

Distributed systems replicate data for good reasons, but keeping replicas consistent when facing network partitions is complicated since programmers face a trade-off between availability and consistency, as identified by the CAP theorem [Bre00; Bre12]. To further complicate matters, today's systems experience huge workloads and users expect applications to work even when they are on the move and have only reduced connectivity. As a result, high availability, low latencies, and good scalability are often more important than strong consistency. For this reason, distributed systems increasingly shift toward weakly consistent data replication. However, ensuring convergence and data integrity under weak consistency is hard.

To relieve programmers from the complexity of programming with weak consistency, researchers propose RDTs that resemble sequential data types but internally embed nifty, often ad-hoc, design tricks to ensure state convergence. However, designing dedicated RDTs for every possible data

type does not scale. Several composition techniques exist [MV15; KB17; WMM20] but none allow for arbitrary compositions of all CRDTs.

We identified three main problems of RDTs, in Section 1.1.3, that hinder their integration in mainstream distributed applications:

Non-customizable semantics. Existing RDTs exhibit hardcoded concurrency semantics but real-world applications require custom semantics. As a result, programmers often find themselves building new RDTs *from scratch* for their specific use case. This is problematic because it exposes programmers to ad-hoc conflict resolution which is hard, even for experts.

Limited support for application invariants. Applications often involve business-specific data integrity invariants that should not be violated. However, traditional RDTs do not support application-specific invariants out-of-the-box. Some approaches extend RDTs with invariants but these require programmers to provide separate (often formal) specifications, e.g. in first-order logic. We cannot expect regular software engineers to write formal specifications as these are highly complex. Moreover, they hamper software evolution as the specifications must evolve along with the code.

Complexity of verification. RDTs are built around subtle design choices and assumptions (e.g. causal delivery) that are easy to break. Therefore, it is crucial to verify RDT implementations before deploying them in production. However, verification of RDTs is mostly done *manually* based on formal specifications instead of actual implementations. As a result, formal verification of RDTs is currently reserved for experts in distributed systems and verification.

7.2 Overview of our Approach

To address the three aforementioned problems of RDTs we devised a principled approach for the development of application-specific RDTs and developed a novel programming language to verify RDT implementations automatically without requiring expertise in formal verification.

Our approach is built around three fundamental principles, initially defined in Section 1.2:

Don't Design for Replication, Replicate your Design. According to this principle, programmers should not manually deal with conflicts but instead replicate existing data types and declaratively specify the desired concurrency semantics.

Correct Replicated Data Types Out-of-the-Box. This principle requires RDTs to be correct out-of-the-box such that programmers do not need to manually verify RDT implementations.

Programming Language Support. This principle requires techniques for building RDTs to feature appropriate language support.

We now provide an overview of our approach that summarizes each chapter and explains how they solve the aforementioned problems (cf. Section 7.1) according to our vision.

We started in Chapter 3 with the SECRO approach which extends sequential data types with application-specific preconditions and invariants that are used at runtime to compute valid executions of the operations. However, computing serializations at runtime proved to be inefficient and does not scale to the workloads experienced by modern applications.

To address the performance issue of SECROs, Chapter 4 proposes the ECRO approach which statically analyzes RDT specifications to detect conflicts and identify solutions beforehand. Although the resulting RDTs exhibit good performance and outperform related hybrid approaches, programmers need to define separate specifications which are cumbersome and error-prone. Subtle mistakes in the RDT specification may lead to runtime anomalies which violates our second principle. Furthermore, programmers now also need to maintain separate specifications which considerably complicates software evolution and thus does not completely meet our third principle.

Chapter 5 introduces EFx, a novel programming language that simplifies the development of RDTs by extending sequential data types with concurrency contracts. These contracts consist of high-level preconditions and invariants that are integrated in the language. We built EFx such that the entire language and its collections have efficient SMT encodings. As a result, EFx can analyze arbitrary data types and their contracts in order to synthesize correct ECROs automatically. Thus, programmers no longer have to write separate specifications in low-level specification languages. This solves the problem of disconnected specifications in ECROs.

The development of RDTs in EFx meets our three principles: RDTs are synthesized from sequential data types and their concurrency contract (principle 1). The resulting RDTs leverage the ECRO protocol to guarantee state convergence and maintain application-specific invariants out-of-the-box (principle 2). The entire approach is integrated in the EFx language which fulfills the principle of programming language support (principle 3).

While EFx proposes a principled approach for the development of custom RDTs, many researchers and practitioners build ad-hoc RDTs. Since ad-hoc approaches are brittle, it is important to provide appropriate verification tools. Chapter 6 proposes VeriFx, a variation on EFx that derives automated correctness proofs for high-level RDT implementations. VeriFx returns concrete counterexamples for incorrect RDTs. This is essential to iteratively improve RDT implementations until they are correct.

We conclude that this dissertation successfully tackles the problems of non-customizable semantics and limited support for invariants by enabling programmers to build custom RDTs with application-specific invariants in EFx. We also addressed the problem that RDTs are difficult to get right by enabling programmers to verify real-world RDT implementations *automatically* in VeriFx without requiring verification-specific code.

7.3 Reviewing the Contributions

Throughout this dissertation, we realized our vision for the development and verification of RDTs, outlined in the introduction (cf. Section 1.2). This led to three main contributions which we now review with hindsight.

The ECRO family of RDTs. We presented a new family of RDTs that are built by augmenting sequential data types with a declarative specification of the desired concurrency semantics. These specifications enable programmers to customize the conflict resolution policy of RDTs and express application-specific invariants that must be maintained. The first incarnation of this approach was the SECRO data type, however, it did not meet the performance requirements of modern applications. We then proposed improved approaches, ECRO and EFx, that statically analyze the specifications to detect conflicts and find coordination-free solutions beforehand. The resulting ECRO RDTs use this information to efficiently serialize

operations in order to guarantee convergence and preserve application invariants. We conducted a geo-distributed RUBiS benchmark which showed that even though replicas occasionally need to reorder operations, this is often more efficient than coordinating the operations (i.e. executing them under strong consistency).

Synthesizing RDT specifications. We proposed EFx, a novel programming language that simplifies the development of RDTs using the ECRO approach. EFx integrates a contract system that allows programmers to attach high-level preconditions and invariants to a data type's operations. These preconditions and invariants are expressed in high-level EFx code, instead of first-order logic as was the case for ECROs. We carefully designed EFx such that any EFx program can be readily analyzed using SMT solving. As a result, EFx can automatically analyze RDT implementations and their contract to synthesize an appropriate distributed specification and generate a correct ECRO for the given data type. We validated our approach by implementing an extensive portfolio of RDTs and several real-world applications including a flight reservation system, an auction system, and a banking application.

Automated verification of RDTs. We developed VeriFx, a high-level programming language with integrated verification capabilities for the development of RDTs. VeriFx allows programmers to implement RDTs using collections and operations from functional programming. The resulting RDTs are automatically verified for the necessary correctness properties. If the RDT is not correct, VeriFx returns a concrete counterexample which enables programmers to iteratively improve the implementation until it is correct. We validated VeriFx by implementing and verifying 37 CRDTs - many of which are used in industrial databases - and reproducing a study on the correctness of OT functions. This work accounts for the most extensive portfolio of verified RDTs to date. Our results show that it is possible to integrate automated verification in a high-level programming language for the development of RDTs, which frees programmers from an additional and complicated verification step.

7.4 Avenues for Future Research

Before we conclude this dissertation, we identify potential avenues for future research that seek to address some of the current limitations of ECROs, EFx, and VeriFx. We also identify other properties of RDTs that could be verified using VeriFx.

7.4.1 Multi-Object Invariants

The ECRO family of RDTs lets programmers turn sequential data types into RDTs that converge and respect application-specific invariants. However, invariants are confined to a single RDT and cannot span several instances of the same or different RDTs.

An interesting research avenue consists in adding support for *multi-object invariants*, i.e. invariants that span several objects. For example, a banking system may use an RDT per account and require clients to keep a positive balance across accounts (i.e. the sum of a client's account balances should be non-negative).

Adding support for multi-object invariants to the ECRO approach will require modifications to the ECRO analyses since all pairs of operations of these objects must be analyzed. It remains to be seen if this is feasible for invariants spanning many objects since the number of operation pairs that need to be analyzed grows significantly with every object that must be considered. In addition, the ECRO protocol will need to be adapted in order to respect invariants even when operations are invoked concurrently on different objects. This will require the protocol to safely serialize the operations of all objects related by an invariant. For instance, the protocol could keep a DAG of operations per group of objects that are related by one or more invariants. It remains to be seen which impact this will have on performance and if more efficient strategies exist.

7.4.2 Improving Automated Verification

The EFx and VeriFx languages currently have a number of limitations that follow directly from their design (cf. Section 5.7). We now discuss potential avenues for future research that aim to lift these limitations.

7.4.2.1 Inductive Proofs

Recall that EFx and VeriFx provide extensive functional collections with higher-order operations. These are encoded using the Combinatory Array Logic (CAL) [MB09] which allows for efficient and automated verification. However, the collections do not provide aggregation methods because those cannot be encoded using CAL. Instead, they need to be encoded recursively, and this requires inductive reasoning, which the underlying SMT solver (Z3 [MB08]) does not support out-of-the-box.

A potential avenue for future research is to explore the integration of inductive proofs in EFx and VeriFx, which would pave the way for adding aggregation methods. The difficulty consists in finding a suitable encoding for these inductive proofs such that SMT solvers can automate the verification process. While VeriFx currently leverages the Z3 solver [MB08], other solvers may be more appropriate. For example, Reynolds and Kuncak [RK15] have integrated inductive reasoning in CVC4[1].

7.4.2.2 Proof Composition

VeriFx provides a novel proof construct for programmers to specify application-specific correctness properties and discharges those properties automatically using the Z3 SMT solver [MB08].

Some proofs may depend on other proofs, but this cannot be expressed in VeriFx. For example, Listing 7.1 shows the implementation of a state-based Positive-Negative Counter CRDT as the composition of two Grow-Only Counter CRDTs [Sha+11a]. The `merge` method merges two `PNCounter`s by merging their respective `GCounter`s.

Listing 7.1: Implementation of a PN-Counter CRDT in VeriFx by composing two G-Counter CRDTs.

```
1 class PNCounter(p:GCounter,n:GCounter) extends CvRDT[PNCounter]{
2   // ...
3   def merge(that: PNCounter) =
4     new PNCounter(this.p.merge(that.p), this.n.merge(that.n))
5 }
```

[1] `https://cvc4.github.io/`

Currently, to prove that the `PNCounter` is correct, VeriFx needs to prove that the `GCounter` is a CRDT and also that the composition of those two `GCounter`s into a `PNCounter` is a CRDT. Thus the proof of the `PNCounter` depends on the proof of the `GCounter`. A more efficient approach would be to first verify the `GCounter` and then verify the `PNCounter` given the correctness proof for the `GCounter`, i.e. by explicitly encoding the fact that `GCounter` is a CRDT as an assumption of the proof. As such, VeriFx only needs to reason about the composition and not about the underlying `GCounter`s. This strategy was employed in [BFP] to verify invariants of composed CRDTs in Antidote SQL [Lop+19] with VeriFx.

Proof composition could also simplify the verification of nested RDTs. For instance, some map RDTs [Pre18] allow the values to again be RDTs. Thus, programmers should be able to prove that the map is correct given that the underlying RDTs are also correct. For a concrete map, say `Map[ORSet[Int]]`, the correctness proof of the map depends on the proof of the underlying `ORSet` CRDT.

It remains to be seen if proof composition is suited to automatically verify complex RDTs formed by arbitrary compositions and nestings.

7.4.3 Going Further with Automated Verification

In this dissertation, we successfully used VeriFx to verify the consistency guarantees of CRDTs [Sha+11b] and OT [EG89]. We now identify other properties of RDTs that could be verified with VeriFx.

7.4.3.1 Verifying Functional Correctness of RDTs

Most RDTs presented in the literature are replicated variants of well-known sequential data structures. It is well-known that RDTs can have concurrency semantics that cannot be explained by a sequential execution of the operations [Pre18]. However, any sequential execution on an RDT should be identical to that execution on the corresponding sequential data type. We say that the RDT must preserve the data type's sequential semantics.

It would be interesting to verify functional correctness for well-known RDTs using VeriFx. However, research is needed to determine which conditions are sufficient to prove functional correctness. Is it enough to prove that all pairs of sequential operations lead to equivalent states when

applied on the RDT and the sequential data type? Or do we also need to prove other properties?

7.4.3.2 Verifying Invariants

Real-world applications involve business-specific invariants that must be respected at all times. Although operations uphold invariants locally, they may break invariants when executed concurrently with other operations. For example, in a courseware system, students may enroll for courses and enrollments must be linked to existing students. This is a widespread invariant known as referential integrity. Now, a student may enroll for a course and *concurrently* that course could be deleted from the system. Although both operations maintain referential integrity when executed locally, the resulting state violates the invariant because the student is enrolled in a course that no longer exists.

To uphold application-specific invariants programmers sometimes resort to ad-hoc RDT approaches. For example, programmers can compose RDTs in a specific way to achieve the desired semantics. It would be interesting to verify that those RDTs indeed preserve the application's invariants.

7.5 Closing Remarks

Our journey toward a simple and efficient approach to build application-specific RDTs resulted in the EFx language, a minimalist object-oriented programming language for the development of RDTs by means of simple concurrency contracts that describe the desired semantics and invariants. EFx synthesizes ECROs from sequential data types and their concurrency contracts. The resulting ECROs guarantee convergence and invariant-preservation out-of-the-box.

We then took our approach a step further and developed VeriFx, a variation on EFx that leverages similar SMT encodings to verify real-world RDTs. This approach proved fruitful as we were able to automatically verify 37 well-known CRDTs and 9 OT designs.

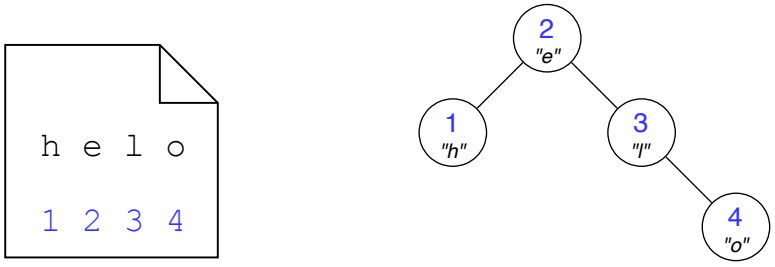
In conclusion, EFx and VeriFx present significant improvements over existing approaches for the development and verification of RDTs. We

believe that the integration of SMT solving in high-level programming languages is key to help programmers develop bulletproof software.

Appendix A

Tree Organization of a Text Document

In order to implement an efficient text editor application, we organize the text document as a balanced tree of characters. Informally, given a node N, nodes in its left subtree must occur before N in the document, whereas nodes in the right subtree must occur after N in the document.



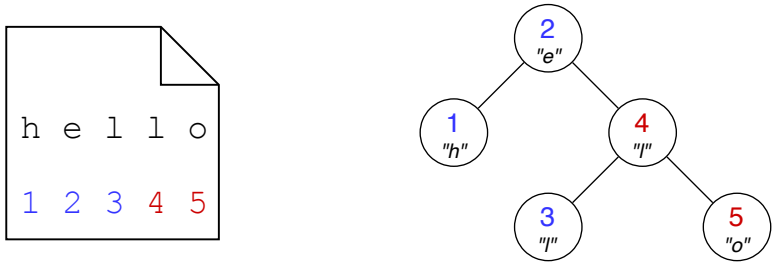
(a) A simple text document.

(b) The document's tree representation.

Figure A.1: A text document and its tree representation. Numbers indicate the characters' indices.

Although the index of a character reflects its position within the document, the text editor cannot organize the tree according to absolute character indices, as they are not stable, i.e. indices may change over time as characters are inserted and deleted. To illustrate the problem, Fig. A.1 shows a text document and its tree representation. In Fig. A.2,

we insert the character “l” after “e” in the document, which affects the index of all succeeding characters (red indices). Hence, in order to remain correct, every affected node of the tree is updated accordingly, thereby making insertions and deletions linear operations.



(a) Modified text document.

(b) Modified tree representation.

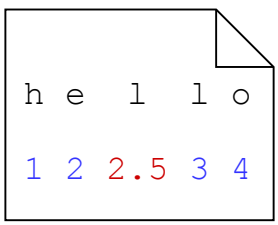
Figure A.2: A text document and its tree representation. Red numbers indicate index changes compared to Fig. A.1.

If character indices are used as is, they reflect the order of the characters but are not stable. On the other hand, the unique IDs of characters are stable but do not reflect their order. A solution to this problem is to generate identifiers such that they **a)** reflect the position of the characters, and **b)** are stable (i.e. a character’s identifier does not change over time) Generating stable identifiers is based on the previous and next characters:

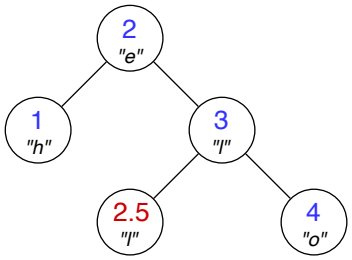
$$\text{generate\_id(prev, next)} = \begin{cases} 1, & \text{if } \neg\text{prev} \wedge \neg\text{next} \\ \frac{\text{id(next)}}{2}, & \text{if } \neg\text{prev} \\ \text{id(prev)} + 1, & \text{if } \neg\text{next} \\ \frac{\text{id(prev)}+\text{id(next)}}{2}, & \text{otherwise} \end{cases}$$

When inserting a character, its identifier is the average of the identifiers of the previous and next characters (last case). The first three cases are corner cases, which arise when the document is empty (1^{st} case), when prepending a character to the document (2^{nd} case) and when appending a character to the document (3^{rd} case). Notice that this scheme reflects the order of the characters since the generated identifier is smaller than the next identifier and bigger than the previous identifier (by definition of the average).

Assume the same text document and identifiers as in Fig. A.1. If a user inserts the character “l” after “e”, this results in the text document and



(a) Modified text document.



(b) Modified tree representation.

Figure A.3: A text document and its tree representation. Red number is the identifier of the newly added character.

tree shown in Fig. A.3. Using the aforementioned scheme, the identifiers of the other characters remain unchanged.

Appendix B

Formal Definition of the Transitive Closure of Concurrent Operations

The SECRO algorithm discussed in Section 3.2.2.1 checks preconditions and postconditions based on the transitive closure of concurrent operations. This appendix formally defines the transitive closure of concurrent operations.

Definition 10: Happens before relation

An operation $m_1 = (o_1, p_1, a_1, c_1, id_1)$ happened before an operation $m_2 = (o_2, p_2, a_2, c_2, id_2)$ iff the logical timestamp of m_1 happened before the logical timestamp of m_2: $m_1 \prec m_2 \iff c_1 \prec c_2$.

Definition 11: Concurrency relation

Two operations m_1 and m_2 are concurrent iff neither one happened before the other [Lam94]: $m_1 \parallel m_2 \iff m_1 \not\prec m_2 \wedge m_2 \not\prec m_1$.

Definition 12: Transitive closure of concurrency relation

We define $\parallel^+$ as the transitive closure of $\parallel$.

Definition 13: Transitive closure of concurrent operations

The set of all operations that are transitively concurrent to an operation m with respect to a history h is defined as: $TC(m, h) = \{m' \mid m' \in h \wedge m' \parallel^{+} m\}$.

Appendix C

Scala DSL for First-Order Logic

The ECRO approach augments sequential data types with a distributed specification. These specifications describe operations and application invariants using first-order logic. As described in Section 4.2.2, we designed an embedded domain-specific language (DSL) for programming first-order logic formulas in Scala, which are then translated to SMT formulas. We now briefly discuss the different components of the language.

Types, values, and operators. The DSL features three primitive types (booleans, integers, and strings) and supports custom types that can be used to represent complex types such as user-defined classes (cf. Table C.1). Primitive values are represented by the `BoolValue`, `IntValue`, and `StringValue` wrappers. We also provide the traditional numeric operators, boolean operators, and comparison operators and provide convenient infix notations for them (cf. Table C.2).

Type	*Type Representation*	*Value Representation*
Boolean	`case object Bool extends Type`	`case class BoolValue(value: Boolean)`
Integer	`case object Integer extends Type`	`case class IntValue(value: Int)`
String	`case object Stringg extends Type`	`case class StringValue(value: String)`
<Custom>	`case class CustomType(name: String) extends Type`	/

Table C.1: Types supported by the DSL.

Description	*Representation*	*Infix Notation*
Equals	`case class Equals(lhs: Any, rhs: Any)`	`lhs === rhs`
Not Equals	`case class NotEquals(lhs: Any, rhs: Any)`	`lhs <> rhs`
Boolean and	`case class And(lhs: Any, rhs: Any)`	`lhs /\ rhs`
Boolean or	`case class Or(lhs: Any, rhs: Any)`	`lhs \/ rhs`
Negation	`case class Not(stat: Any)`	/
Plus	`case class Plus(lhs: Any, rhs: Any)`	`lhs |+| rhs`
Minus	`case class Minus(lhs: Any, rhs: Any)`	`lhs |-| rhs`
Multiplication	`case class Times(lhs: Any, rhs: Any)`	`lhs |*| rhs`
Division	`case class Divide(lhs: Any, rhs: Any)`	`lhs |/| rhs`
Smaller than	`case class SmallerThan(lhs: Any, rhs: Any)`	`lhs « rhs`
Smaller than or equal to	`case class SmallerThanOrEq(lhs: Any, rhs: Any)`	`lhs «= rhs`
Bigger than	`case class BiggerThan(lhs: Any, rhs: Any)`	`lhs » rhs`
Bigger than or equal to	`case class BiggerThanOrEq(lhs: Any, rhs: Any)`	`lhs »= rhs`

Table C.2: List of operators provided by the DSL.

Variables and identifiers. Programmers can declare *free variables* by providing a name and type for them and refer to them using *identifiers* (cf. Table C.3). Note that declarations do not assign a value to the variable, i.e. they may hold any value of the given type. In order to "assign" a value to a variable, one can state that the variable equals the desired value. For example, `Identifier("age") === IntValue(25)` "assigns" the value 25 to an existing variable "age".

Relations and states. Relations constrain the state of an object. The DSL provides two special state types: `OldState` and `NewState` (cf. Table C.3). The former represents the state of the object prior to applying an operation, whereas the latter represents the state after applying the operation. This enables programmers to express the effects of an operation. For instance, the value of a counter can be represented with a

Description	*Representation*	*Notation*
Identifier	`case class Identifier(name: String)`	/
Variable	`case class Var(name: String, tpe: Type)`	`Identifier(name) :: tpe`
Relation	`case class Relation(name: String, vars: Var*)(ret: Type)`	/
First-Order Logic Formula	`case class RelationInstance(name: String, args: Any*)`	`Relation(name, _)(args)`
State	`sealed trait State`	/
Old State	`class OldState extends State`	/
New State	`class NewState extends State`	/
Current State	`class CurrentState extends State`	/
Universal Quantifier	`case class Forall(vars: Set[Var], body: Formula)`	`forall(vars) :- body`
Existential Quantifier	`case class Exists(vars: Set[Var], body: Formula)`	`exists(vars) :- body`
Logical Implication	`case class Implication(ante: Formula, conse: Formula)`	`ante ==> conse`

Table C.3: Logic building blocks provided by the DSL.

Listing C.1: Overview of the interface of the `Relation` class.

```
case class Relation(name: String, vars: Var*)(ret: Type) {
  def instance(args: Any*): RelationInstance
  def apply(args: Any*) = instance(args:_*)
  def copy(fromTo: (State, State)): Formula
  def copyExcept(fromTo: (State, State), condition: Formula):
    Formula
  def copyWhen(fromTo: (State, State), condition: Formula):
    Formula
  // `key`-`v` must be unique
  def unique(key: Var, v: Var, state: State): Formula
  def assertion(cond: Formula, state: State): Formula
}
```

relation $value :: State \to Integer$. Incrementing the counter can then be expressed as `value(newState) === value(oldState) |+| 1`.

Custom relations, such as the aforementioned `value` example, are defined by instantiating the `Relation` class shown in listing C.1. More concretely, relations are instatiated with a name, a variable number of typed arguments, and a return type. As shown in listing C.1, relations provide methods to copy facts from one state to another (`copy`, `copyExcept`, and `copyWhen`), express uniqueness constraints (`unique`), or assert any other condition (`assertion`). Relations are instantiated by applying them to some arguments and yield a first-order logic formula, e.g. `value(newState)`.

Quantifiers and implications. The DSL provides universal and existential quantifiers (`forall` and `exists` functions, cf. Table C.3) which take one or more variables and a boolean formula that specifies a property about these variables. Logical implication can be expressed using the `==>` infix notation which expects two boolean formulas: the antecedent and the consequent.

C.1 Complete Set Specification

We now present the complete specification of the Set ECROs discussed in Section 4.2 using our DSL.

Listing C.2: Distributed specification of the Add-Wins Set.

```
1  case class AWSet[V](set: Set[V]) extends ESet[V]
2  object AWSet extends DistributedSpec {
3    // Declarations
4    val V = CustomType("V"); val elem = "elem"
5    val contains = Relation("contains", Var(elem, V))(Bool)
6    val x = Identifier("x")
7    // Specs
8    val aws = classOf[AWSet[_]]
9    val add = aws.getDeclaredMethod("add", classOf[Object])
10   val remove = aws.getDeclaredMethod("remove", classOf[Object])
11   val relations = Set(contains)
12   val operations: Map[Method, Mutator] = Map(
13     add -> Mutator(
14       post = (old: OldState, res: NewState) => {
15         contains(res, x) /\
16         contains.copyExcept(old -> res, elem === x)
17       },
18       // add wins invariant
19       inv = (_: OldState, res: NewState) => contains(res, x)),
20     remove -> Mutator(
21       post = (old: OldState, res: NewState) => {
22         not (contains(res, x)) /\
23         contains.copyExcept(old -> res, elem === x)
24       })) }
```

Listing C.2 shows the distributed specification of the Add-Wins Set ECRO. To represent elements contained by the set we will need a predicate[1] $contains :: V \times State \rightarrow Boolean$. To this end, line 4 defines a custom type `V` which is the abstract type of the elements that are contained by the set (i.e. it corresponds to the type parameter `V` in `AWSet[V]`). Line 5 then defines the `contains` relation which takes one argument `elem` and is true if `elem` is contained by the set, false otherwise. Note that we do not define a "set" argument explicitly because every relation is defined over a state, hence, the DSL adds a state argument (representing the object) behind the scenes.

Now that we defined the `contains` predicate, we can implement the actual specification of the operations. First, we inform the DSL about all relations we will use, by providing a set containing the relations (see the `relations` field on line 11). Then, we provide the DSL with an `operations` field that maps each method to its specification (lines 12 to 24). As explained in Section 4.2, the postconditions of `add` and `remove`

[1] A predicate is a relation that returns a boolean.

state that the added element `x`[2] is present/absent in the resulting set and use the `copyExcept` method defined on relations to copy all the other elements from the `old` set to the `res` set (lines 16 and 23). The invariant on `add` states that the added element must occur in the resulting state and thus guarantees add-wins semantics. The Remove-Wins Set is similar, except that it puts an invariant on `remove` such that the removed element is not present in the resulting state (cf. Listing 4.2 in Section 4.2.2).

C.2 RUBiS Specification

We now present the complete specification of the RUBiS ECRO discussed in Section 4.2.3 using our DSL. More concretely, we provide the complete implementation of the `placeBid` and `closeAuction` operations for the RUBiS application. Listing C.3 shows the sequential implementation of the RUBiS data type. It keeps a set of users and a map from auction IDs to auctions (line 32). Auctions consist of a set of bids, a status (open or closed), and optionally a winner (line 19). Method `placeBid` (line 40) retrieves the auction and places the bid on the auction. `closeAuction` (line 46) retrieves the auction and puts its status on closed.

To turn this sequential RUBiS data type into an ECRO, we augment it with a distributed specification, shown in Listing C.4. First, we declare three first-order logic predicates to represent auctions, users, and bids on auctions: `auction(id, status)`, `user(name)`, and `bid(auction, user, amount)` (line 12 to 14). Then, we use these predicates to describe the `placeBid` and `closeAuction` operations, as explained in Section 4.2.3. The precondition of `placeBid` (line 34 to 38) requires the auction to be open, the user to exist, the price to be bigger than zero, and every auction to be well-formed (i.e. either open or closed but not both). The postcondition of `placeBid` (line 39 to 42) adds the bid and copies all the existing bids from the old state to the new state. The precondition of `closeAuction` (line 44) states that auctions must be well-formed. Its postcondition (line 45 to 49) closes the auction, states that the auction can no longer be open, and copies all other auctions from the old state to the new state.

[2] `x` is defined on line 6 and corresponds to the parameter of the `add` and `remove` operations.

Listing C.3: Sequential RUBiS implementation.

```
import scala.collection.SortedSet

type User = String
type AID = String
sealed trait Status
case object Open extends Status
case object Closed extends Status

case class Bid(userId: User, bid: Int) extends Ordered[Bid] {
  def compare(that: Bid): Int = bid.compareTo(that.bid)
}

val bidOrdering =
  Ordering.by[Bid, Bid](
    b => b.copy(bid = b.bid * -1)) // big to small

case class Auction(
    bids: SortedSet[Bid] = SortedSet.empty[Bid](bidOrdering),
    status: Status = Open, winner: Option[User] = None) {
  def bid(userId: User, price: Int) =
    copy(bids = bids + Bid(userId, price))

  def close() = {
    val highestBid: Option[Bid] = bids.headOption
    val winner = highestBid.map(_.userId)
    copy(status = Closed, winner = winner)
  }
}

case class Rubis(
    users: Set[User] = Set(),
    auctions: Map[AID, Auction] = Map()) extends ECRO {
  private def getAuction(auctionId: AID) = {
    auctions.get(auctionId) match {
      case Some(auction) => auction
      case None => throw AuctionNotFound(auctionId)
    }
  }

  def placeBid(auctionId:AID, userId:User, price:Int): Rubis = {
    val auction = getAuction(auctionId)
    val updatedAuction = auction.bid(userId, price)
    copy(auctions = auctions.updated(auctionId, updatedAuction))
  }

  def closeAuction(auctionId: AID): Rubis = {
    val auction = getAuction(auctionId)
    copy(auctions = auctions.updated(auctionId, auction.close))
  }
}
```

Listing C.4: Distributed specification for RUBiS ECRO.

```
object Rubis extends DistributedSpec {
  // Declarations
  val id = "id"
  val idVar = Variable(id, Stringg)
  val statusV = Variable("status", Bool)
  val auctionVar = Variable("auction", Stringg)
  val userVar = Variable("user", Stringg)
  val amountVar = Variable("amount", Integer)
  val Open = True; val Closed = False

  // Relations
  val auction = Relation("auction", idVar, statusV)(Bool)
  val user = Relation("user", userVar)(Bool)
  val bid = Relation("bid",auctionVar,userVar,amountVar)(Bool)

  val auctionId = Identifier("auctionId")
  val price = Identifier("price")
  val userId = Identifier("userId")

  val rbs = classOf[Rubis]
  val str = classOf[String]; val i = classOf[Int]
  val placeBid = rbs.getDeclaredMethod("placeBid", str, str, i)
  val closeAuction = rbs.getDeclaredMethod("closeAuction", str)

  // Specs
  val relations = Set(auction, user, bid)

  // auctions are either open or closed but not both
  def auctionsOpenOrClose(state: State) =
    auction.unique(idVar, statusVar, state)

  val operations: Map[Method, Mutator] = Map(
    placeBid -> Mutator(
      pre = (state: CurrentState) => {
        auction(auctionId, Open, state) /\
        user(userId, state) /\ (price >> 0) /\
        auctionsOpenOrClose(state)
      }
      post = (old: OldState, res: NewState) => {
        old + bid(auctionId, userId, price, newState) /\
        bid.copy(old -> res)
      }),
    closeAuction -> Mutator(
      pre = (state: CurrentState) => auctionsOpenOrClose(state)
      post = (old: OldState, res: NewState) => {
        old + auction(auctionId, Closed, newState) /\
        not (auction(auctionId, Open, newState)) /\
        auction.copyExcept(old -> res, id === auctionId)
      }))
}
```

Appendix D

Cycle Detection and Resolution in the ECRO Protocol

We now explain how ECRO's replication protocol keeps the execution graph acyclic. Algorithm 11 extends the replication algorithm presented in Section 4.4.1 with a deterministic approach to detect and solve cycles. While adding new edges, the algorithm continuously checks for cycles (line 15). If a newly added edge $c_1 \rightarrow c_2$ causes a cycle, at least one path exists from c_2 to c_1. To solve the cycle, the algorithm computes all paths from c_2 to c_1 (line 38) and breaks them one by one by removing one `ao`-edge on each path (line 41). These edges can be removed without putting at risk convergence since they impose an artificial ordering between non-commutative operations (say $c_i \xrightarrow{ao} c_j$) but we know that they are already ordered by one or more paths (from c_j to c_i) between them (otherwise they would not be part of the cycle). As a result, we solved the cycle while ensuring that all non-commutative operations remain ordered. Sometimes it is not possible to break each path only by removing `ao`-edges. In that case the cycle is caused by a combination of `hb`-edges and `co`-edges. These cannot be removed as this would violate either convergence or safety. Instead, the algorithm deterministically discards a call that breaks the cycle (line 18). Information about discarded `ao`-edges and discarded calls is propagated between the replicas to ensure that all replicas eliminate the same `ao`-edges and/or calls and thus still converge. Since

the set of discarded edges and the set of discarded calls grow monotonically and Algorithm 11 is deterministic, all replicas converge to the same execution graph and hence to equivalent states as proven in Section 4.4.

Algorithm 11 Detecting and solving cycles in the ECRO replication protocol.

```
 1: ⟨Σ, σ0, M, G, t, F⟩, with G = (C, E)                    ▷ ECRO's internal state
 2: σ: Σ                                                    ▷ object current state σ
 3: discarded                                               ▷ set of discarded edges
 4: function EXECUTE_REMOTE(c)                              ▷ execution of call c at remote replica
 5:     new_edges ← ∅                                       ▷ initialise set to keep edges related to call c
 6:     new_discarded ← ∅                                   ▷ initialise set to keep discarded edges related to call c
 7:     C ← C ∪ { c }                                       ▷ update graph vertices
 8:     E ← E \ discarded                                   ▷ remove discard edges from the following analysis
 9:     for v ∈ C ∧ v ≠ c do                                ▷ determine hb and co-edges involving call c
10:         if v ≺ c ∧ not seqCommutative(c, v) then
11:             edge ← ⟨v, hb, c⟩                           ▷ add hb-edge from call v to call c
12:         else if v ∥ c then                              ▷ call v is concurrent with c
13:             if resolution(c, v) = < then edge ← ⟨c, co, v⟩   ▷ order c before v
14:             else if resolution(c, v) = > then edge ← ⟨v, co, c⟩ ▷ order v before c
15:         if causesCycle(edge) then                       ▷ does this edge cause a cycle?
16:             new_discarded ← resolveCycle(edge)          ▷ try discarding ao-edges
17:             if hasNoSolution() then                     ▷ cycle caused by hb and co-edges
18:                 C ← C \ { c }                           ▷ discard call c
19:                 E ← E \ new_edges                       ▷ discard edges related to call c
20:                 propagateDiscardedCall(c)               ▷ inform replicas about discarded call
21:                 return                                  ▷ function terminates
22:         new_edges ← new_edges ∪ { edge }                ▷ collect edges related to call c
23:     for v ∈ C ∧ v ∥ c do                                ▷ determine ao-edges between existing calls and call c
24:         if resolution(c, v) = ⊤ ∧ not commutative(c, v) then
25:             if Id(c) < Id(v) then edge ← ⟨v, ao, c⟩     ▷ impose deterministic order
26:             else edge ← ⟨v, ao, c⟩
27:             if causesCycle(edge) then
28:                 new_discarded ← new_discarded ∪ { edge }    ▷ discard the edge
29:             else new_edges ← new_edges ∪ { edge }       ▷ collect edges related to c
30:     discarded ← discarded ∪ new_discarded               ▷ update discarded edges
31:     propagateDiscardedEdges(discarded)                  ▷ tell replicas to discard these edges
32:     E ← (E ∪ new_edges) \ new_discarded                 ▷ update graph edges
33:     t ← dynamicTopologicalSort(new_edges)
34:     commit()                                            ▷ Commit causally stable operations
35:     σ ← apply(σ0, t)                                    ▷ execute the sequence of calls on the initial state σ0
36: function RESOLVECYCLE(⟨c1, rel, c2⟩)
37:     new_discarded ← ∅                                   ▷ initialize set to keep discarded edges to solve the cycle
38:     paths ← allPaths(c2, c1, G)                         ▷ determine all paths that close the cycle
39:     for p ∈ paths do
40:         if existsArbitrationOrderEdge(p) then           ▷ search ao-edges unique to path p
41:             d ← removeEdge(p)                           ▷ remove the ao-edge that has a minimal id
42:             new_discarded ← new_discarded ∪ { d }       ▷ update discarded edges
43:         else return NO_SOLUTION                         ▷ cycle is caused by an hb or co-edge
44:     return new_discarded
```

Appendix E

Geo-Distributed RUBiS Benchmark on a Read-Mostly Workload

Section 4.6.4 presented a geo-distributed benchmark for the RUBiS application. The benchmark was executed by measuring the latency of operations at DC Paris while the other DCs execute an update-heavy workload consisting of 100 operations per second with 50% reads and 50% writes. We now perform the same experiment with a read-mostly workload consisting of 1000 operations per second with 95% reads and 5% writes.

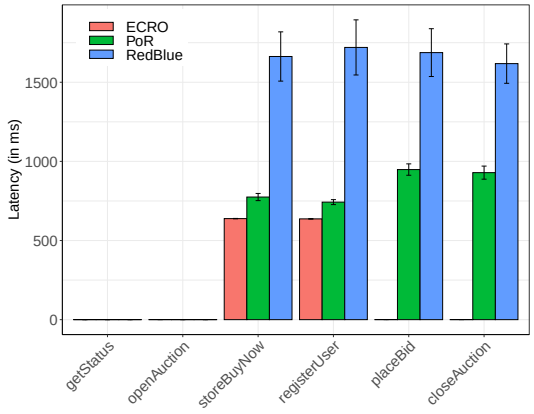


Figure E.1: Average latency of RUBiS operations as observed by users at DC Paris. Error bars represent the 99.9% confidence interval.

Figure E.1 shows the average latency of RUBiS operations under this read-mostly workload. The figures are similar to those depicted in Section 4.6.4. The `getStatus` and `openAuction` operations are safe, hence, they are not coordinated, resulting in low latencies. The `storeBuyNow` and `registerUser` operations are unsafe and require coordination in all implementations (see Table 4.3), and thus have high latencies. The `placeBid` and `closeAuction` operations are unsafe and require coordination in both PoR and RedBlue (see Table 4.3). ECROs do not coordinate these operations because Ordana found a solution to the conflict, which consists of locally ordering `placeBid` operations before `closeAuction` operations when they affect the same auction concurrently (see Table 4.2 in Section 4.5.1). As a result, ECROs achieve low latency (less than 1ms) while PoR and RedBlue exhibit high latencies (more than 900ms).

Appendix F

EFx's Type System

We now present EFx's type system which resembles that of Featherweight Generic Java [IPW01]. An environment Γ is a partial and finite mapping from variables to types. A type environment Δ is a finite set of type variables. EFx's type system consists of a judgment for type well-formedness $\Delta \vdash T$ ok which says that type T is well-formed in context Δ, and a judgment for typing $\Delta; \Gamma \vdash e : T$ which says that in context Δ and environment Γ, the expression e is of type T. We abbreviate $\Delta \vdash T_1$ ok, $\ldots$, $\Delta \vdash T_n$ ok to $\Delta \vdash \overline{T}$ ok, and $\Delta; \Gamma \vdash e_1 : T_1, \ldots, \Delta; \Gamma \vdash e_n : T_n$ to $\Delta; \Gamma \vdash \overline{e} : \overline{T}$.

$$\frac{}{\Delta \vdash \texttt{string}\ \text{ok}}\ (\textit{WF-String}) \qquad \frac{}{\Delta \vdash \texttt{bool}\ \text{ok}}\ (\textit{WF-Bool}) \qquad \frac{}{\Delta \vdash \texttt{int}\ \text{ok}}\ (\textit{WF-Int})$$

$$\frac{X \in \Delta}{\Delta \vdash X\ \text{ok}}\ (\textit{WF-TVar})$$

$$\frac{\begin{array}{c}\Delta \vdash \overline{T}\ \text{ok} \\ \texttt{class}\ \overline{A}\langle C\rangle\,(\overline{X})\,\{\ldots\}\ldots \\ \textit{or}\ \ \texttt{class}\ \overline{A}\langle C\rangle\,(\overline{X})\ \texttt{extends}\ \ldots\{\,I\langle\ldots\rangle\,\}\ldots\end{array}}{\Delta \vdash C\langle\overline{T}\rangle\ \text{ok}}\ (\textit{WF-Class})$$

$$\frac{\begin{array}{c}\Delta \vdash \overline{T}\ \text{ok} \qquad \overline{T} <: \overline{P} \\ \texttt{trait}\ I\,\langle\overline{X} <: \overline{P}\rangle\,\{\ldots\} \\ \textit{or}\ \ \texttt{trait}\ I\,\langle\overline{X} <: \overline{P}\rangle\ \texttt{extends}\ I\langle\ldots\rangle\,\{\ldots\}\end{array}}{\Delta \vdash I\langle\overline{T}\rangle\ \text{ok}}\ (\textit{WF-Trait})$$

Figure F.1: Judgments for type well-formedness in EFx.

Figure F.1 defines well-formed types. Primitive types are always well-formed. A type variable X is valid if it is in scope: $X \in \Delta$, i.e. the surrounding method or class defined the type parameter. Class types and trait types are valid if a corresponding class or trait definition exists and all type arguments are well-formed.

We now define a few auxiliary definitions which are needed for the typing rules. The *fields* function takes a class type and returns its fields and their types:

$$\frac{\texttt{class}\ \overline{A}\,\langle C\rangle\,(\overline{X})\,\{\,\overline{v} : \overline{T}\,\}\overline{M}\ \ \textit{or}\ \ \texttt{class}\ \overline{A}\,\langle C\rangle\,(\overline{X})\ \texttt{extends}\ \overline{v} : \overline{T}\{\,I\langle\overline{Q}\rangle\,\}\overline{M}}{\textit{fields}(C\langle\overline{P}\rangle) = [\overline{P}/\overline{X}]\ \overline{v} : \overline{T}}\ (\text{F-CLASS})$$

The *mtype* function takes the name of a method and the type of a class, and returns the actual type signature of the method. If the method is not found in the class (MT-CLASS-REC rule) it is looked up in the hierarchy of super traits by the MT-TRAIT and MT-TRAIT-REC rules. For polymorphic methods, the returned type signature is polymorphic:

$$\frac{\texttt{class}\ \overline{A}\,\langle C\rangle\,(\overline{X})\,\{\ldots\}\overline{M}\ \ \textit{or}\ \ \texttt{class}\ \overline{A}\,\langle C\rangle\,(\overline{X})\ \texttt{extends}\ \ldots\{\,I\langle\overline{Q}\rangle\,\}\overline{M} \quad \texttt{def}\ m\,\langle\overline{Y}\rangle\,(\overline{x} : \overline{T}) : T = e \in \overline{M}}{\textit{mtype}(m, C\langle\overline{P}\rangle) = [\overline{P}/\overline{X}]\ (\langle\overline{Y}\rangle\overline{T}\ \rightarrow\ T)}\ (\text{MT-CLASS})$$

$$\frac{\texttt{class}\ \overline{A}\,\langle C\rangle\,(\overline{X})\ \texttt{extends}\ \ldots\{\,I\langle\overline{Q}\rangle\,\}\overline{M} \quad \texttt{def}\ m\,\langle\overline{Y}\rangle\,(\overline{x} : \overline{T}) : T = e \notin \overline{M}}{\textit{mtype}(m, C\langle\overline{P}\rangle) = \textit{mtype}(m, I\langle\overline{Q}\rangle)}\ (\text{MT-CLASS-REC})$$

$$\frac{\texttt{trait}\ I\,\langle\overline{X} <: T'\rangle\,\{\,\overline{M}\,\}\ \ \textit{or}\ \ \texttt{trait}\ I\,\langle\overline{X} <: T'\rangle\ \texttt{extends}\ I'\langle\ldots\rangle\,\{\,\overline{M}\,\} \quad \texttt{def}\ m\,\langle\overline{Y}\rangle\,(\overline{x} : \overline{T}) : T = e \in \overline{M}}{\textit{mtype}(m, I\langle\overline{P}\rangle) = [\overline{P}/\overline{X}]\ (\langle\overline{Y}\rangle\overline{T}\ \rightarrow\ T)}\ (\text{MT-TRAIT})$$

$$\frac{\texttt{trait}\ I\,\langle\overline{X} <: T'\rangle\,\{\,\overline{M}\,\}\ \ \textit{or}\ \ \texttt{trait}\ I\,\langle\overline{X} <: T'\rangle\ \texttt{extends}\ I'\langle\overline{P}\rangle\,\{\,\overline{M}\,\} \quad \texttt{def}\ m\,\langle\overline{Y}\rangle\,(\overline{x} : \overline{T}) : T = e \notin \overline{M}}{\textit{mtype}(m, I\langle\overline{P}\rangle) = \textit{mtype}(m, I'\langle\overline{P}\rangle)}\ (\text{MT-TRAIT-REC})$$

Similarly, we assume that there are functions $valNames(I\langle\overline{P}\rangle)$ and $declaredMethods(I\langle\overline{P}\rangle)$ that return all fields, respectively all methods, declared by a trait (and its super traits).

Figure F.2 introduces judgments for the well-formedness of classes and traits. Classes are well-formed if the types of the fields are well-formed and all its methods, preconditions, and invariants are well-formed (T-CLASS1 rule). If the class extends a trait, it must also implement all

$$\frac{\begin{array}{c}\Delta = \overline{X} \quad \Delta \vdash \overline{T}\ \text{ok} \\ \overline{D} = \overline{M} \cup \overline{Pre} \cup \overline{Inv} \qquad \overline{M}\ \mathit{OK\ IN}\ C\langle\overline{X}\rangle \\ \overline{Pre}\ \mathit{OK\ IN}\ C\langle\overline{X}\rangle \qquad \overline{Inv}\ \mathit{OK\ IN}\ C\langle\overline{X}\rangle\end{array}}{\texttt{class}\ \overline{A}\,\langle C\rangle\,(\overline{X})\,\{\,\overline{v} : \overline{T}\,\}\overline{D}\ \mathit{OK}}\ \text{(T-CLASS1)}$$

$$\frac{\begin{array}{c}\Delta = \overline{X} \qquad \Delta \vdash \overline{T}\ \text{ok} \qquad \Delta \vdash I\langle\overline{P}\rangle\ \text{ok} \\ \texttt{trait}\ I\,\langle\ldots\rangle\,\{\,B\,\}\ \mathit{or}\ \texttt{trait}\ I\,\langle\ldots\rangle\ \texttt{extends}\ \ldots\,\{\,B\,\} \\ \overline{D} = \overline{M} \cup \overline{Pre} \cup \overline{Inv} \qquad \overline{M}\ \mathit{OK\ IN}\ C\langle\overline{X}\rangle \\ \overline{Pre}\ \mathit{OK\ IN}\ C\langle\overline{X}\rangle \qquad \overline{Inv}\ \mathit{OK\ IN}\ C\langle\overline{X}\rangle \\ \mathit{valNames}(I\langle\overline{P}\rangle) \subset \overline{v} \qquad \mathit{declaredMethods}(I\langle\overline{P}\rangle) \subset \overline{M}\end{array}}{\texttt{class}\ \overline{A}\,\langle C\rangle\,(\overline{X})\ \texttt{extends}\ \overline{v} : \overline{T}\{\,I\langle\overline{P}\rangle\,\}\overline{D}\ \mathit{OK}}\ \text{(T-CLASS2)}$$

$$\frac{\begin{array}{c}\Delta = \overline{X} \qquad \Delta \vdash \overline{T}\ \text{ok} \qquad \Delta \vdash I'\langle\overline{P}\rangle\ \text{ok} \\ \texttt{trait}\ I'\,\langle\ldots\rangle\,\{\ldots\}\ \mathit{or}\ \texttt{trait}\ I'\,\langle\ldots\rangle\ \texttt{extends}\ \ldots\,\{\ldots\} \\ B = \overline{\mathit{valDecl}} \cup \overline{\mathit{methodDecl}} \cup \overline{M} \qquad \overline{M}\ \mathit{OK\ IN}\ I\langle\overline{X}\rangle \\ \mathit{valNames}(I'\langle\overline{P}\rangle) \subset \overline{\mathit{valDecl}} \\ \mathit{declaredMethods}(I'\langle\overline{P}\rangle) \subset (\overline{\mathit{methodDecl}} \cup \overline{M})\end{array}}{\texttt{trait}\ I\,\langle\overline{X} <: \overline{T}\rangle\ \texttt{extends}\ I'\,\langle\,\overline{P}\,\rangle\,\{\,\overline{B}\,\}\ \mathit{OK}}\ \text{(T-TRAIT)}$$

$$\frac{\begin{array}{c}\Delta = \overline{X}, \overline{Y} \quad \Delta \vdash \overline{T}, T\ \text{ok} \\ \texttt{class}\ \overline{A}\,\langle C\rangle\,(\overline{X})\,\{\ldots\}\ldots \\ \mathit{or}\ \texttt{trait}\ C\,\langle\overline{X} <: \overline{Q}\rangle\,\{\ldots\} \\ \mathit{or}\ \texttt{trait}\ C\,\langle\overline{X} <: \overline{Q}\rangle\ \texttt{extends}\ \ldots\,\{\ldots\} \\ \Delta; \overline{x} : \overline{T}, \mathit{this} : C\langle\overline{X}\rangle \vdash e : T\end{array}}{\texttt{def}\ m\,\langle\overline{Y}\rangle\,(\overline{x} : \overline{T}) : T = e\ \mathit{OK\ IN}\ C\langle\overline{X}\rangle}\ \text{(T-METHOD)}$$

$$\frac{\begin{array}{c}\Delta = \overline{X}, \overline{Y} \quad \Delta \vdash \overline{T}, T\ \text{ok} \\ \texttt{class}\ @\mathit{replicated}\,\langle C\rangle\,(\overline{X})\,\{\ldots\}\overline{D} \\ \Delta; \overline{x} : \overline{T}, \mathit{this} : C\langle\overline{X}\rangle \vdash e : \mathsf{bool} \\ \texttt{def}\ m\,\langle\overline{Y}\rangle\,(\overline{x} : \overline{T}) : T = e' \in \overline{D}\end{array}}{\texttt{pre}\,m\,\langle\overline{Y}\rangle\,(\overline{x} : \overline{T})\,\{\,e\,\}\ \mathit{OK\ IN}\ C\langle\overline{X}\rangle}\ \text{(T-PRE)}$$

$$\frac{\begin{array}{c}\Delta = \overline{X}, \overline{Y} \quad \Delta \vdash \overline{T}, T\ \text{ok} \\ \texttt{class}\ @\mathit{replicated}\,\langle C\rangle\,(\overline{X})\,\{\ldots\}\overline{D} \\ \Delta; \overline{x} : \overline{T}, \mathit{this} : C\langle\overline{X}\rangle, \mathit{old} : C\langle\overline{X}\rangle \vdash e : \mathsf{bool} \\ \texttt{def}\ m\,\langle\overline{Y}\rangle\,(\overline{x} : \overline{T}) : T = e' \in \overline{D}\end{array}}{\texttt{inv}\,m\,\langle\overline{Y}\rangle\,(\overline{x} : \overline{T})\,\{\,e\,\}\ \mathit{OK\ IN}\ C\langle\overline{X}\rangle}\ \text{(T-INV)}$$

Figure F.2: Judgments for well-formedness of classes and traits in EFx.

fields and methods declared by the hierarchy of super traits (T-CLASS2 rule). The judgment for the well-formedness of traits is defined similarly. Preconditions and invariants are well-formed if they are associated with an existing method of the replicated class (i.e. a class that is annotated with `@replicated`) and their body returns a boolean (T-PRE and T-INV rules). Preconditions and invariants can use `this` to refer to the current instance of the object. Additionally, invariants can also use `old` to refer to the old instance of the object (i.e. the object as it was before executing the method call).

Figure F.3 shows the typing rules for expressions. Most rules are a simplification of Featherweight Generic Java [IPW01] without subtyping. Literal values and arithmetic and boolean operations are straightforward to type. Types of variables (T-VAR rule) are looked up in the environment Γ. If statements (T-IF rule) are of type T if the condition is of type boolean and both branches are of type T. Value statements (T-VAL rule) introduce a variable in the environment and have the same type as their body. Anonymous functions (T-ABSTRACTION) are of a function type where the domain corresponds to the type of the parameters and the codomain corresponds to the type of the body. The type of a function invocation is the type of the function's codomain (T-CALL rule). The T-NEW and T-FIELD rules are simple rules to type class instantiations and field accesses respectively. To type (polymorphic) method invocations (T-INVOKE rule) we fetch the method's type (which is a polymorphic function type) and substitute all occurrences of the method's type parameters with the actual type arguments.

$$\frac{}{\Delta;\Gamma \vdash \texttt{num} : \texttt{int}}\ (\text{T-NUM}) \qquad \frac{}{\Delta;\Gamma \vdash \texttt{str} : \texttt{string}}\ (\text{T-STR})$$

$$\frac{}{\Delta;\Gamma \vdash \texttt{true} : \texttt{bool}}\ (\text{T-TRUE}) \qquad \frac{}{\Delta;\Gamma \vdash \texttt{false} : \texttt{bool}}\ (\text{T-FALSE})$$

$$\frac{x \in dom(\Gamma)}{\Delta;\Gamma \vdash x : \Gamma(x)}\ (\text{T-VAR}) \qquad \frac{\Delta;\Gamma \vdash e : \texttt{bool}}{\Delta;\Gamma \vdash !e : \texttt{bool}}\ (\text{T-NEG})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \texttt{int} \quad \Delta;\Gamma \vdash e_2 : \texttt{int}}{\Delta;\Gamma \vdash e_1 \oplus e_2 : \texttt{int}}\ (\text{T-OP1}) \qquad \frac{\Delta;\Gamma \vdash e_1 : \texttt{bool} \quad \Delta;\Gamma \vdash e_2 : \texttt{bool}}{\Delta;\Gamma \vdash e_1 \otimes e_2 : \texttt{bool}}\ (\text{T-OP2})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \texttt{bool} \quad \Delta;\Gamma \vdash e_2 : T \quad \Delta;\Gamma \vdash e_3 : T}{\Delta;\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T}\ (\text{T-IF}) \qquad \frac{\Delta \vdash T_1\ \text{ok} \quad \Delta;\Gamma \vdash e_1 : T_1 \quad \Delta;\Gamma, x : T_1 \vdash e_2 : T_2}{\Delta;\Gamma \vdash \texttt{val}\ x : T_1 = e_1\ \texttt{in}\ e_2 : T_2}\ (\text{T-VAL})$$

$$\frac{\Delta \vdash \vec{T}\ \text{ok} \quad \Delta;\Gamma, \vec{x} : \vec{T} \vdash e : T}{\Delta;\Gamma \vdash (\vec{x} : \vec{T}) \Rightarrow e : \vec{T} \to T}\ (\text{T-ABSTRACTION}) \qquad \frac{\Delta;\Gamma \vdash e_1 : \vec{T} \to T \quad \Delta;\Gamma \vdash \vec{e_2} : \vec{T}}{\Delta;\Gamma \vdash e_1(\vec{e_2}) : T}\ (\text{T-CALL})$$

$$\frac{fields(C\langle\overline{P}\rangle) = \overline{v} : \overline{T} \quad \Delta \vdash C\langle\overline{P}\rangle\ \text{ok} \quad \Delta;\Gamma \vdash \overline{e} : \overline{T}}{\Delta;\Gamma \vdash \texttt{new}\ C\langle\overline{P}\rangle(\,\overline{e}\,) : C\langle\overline{P}\rangle}\ (\text{T-NEW}) \qquad \frac{\Delta;\Gamma \vdash e : T_o \quad fields(T_o) = \overline{v} : \overline{T}}{\Delta;\Gamma \vdash e.v_i : T_i}\ (\text{T-FIELD})$$

$$\frac{\Delta;\Gamma \vdash e_o : T_o \quad \Delta \vdash \overline{P}\ \text{ok} \quad mtype(m, T_o) = \langle\overline{X}\rangle\,\overline{T} \to T \quad \Delta;\Gamma \vdash \overline{e} : [\overline{P}/\overline{X}]\overline{T}}{\Delta;\Gamma \vdash e_o.m\ \langle\overline{P}\rangle\,(\,\overline{e}\,) : [\overline{P}/\overline{X}]\,T}\ (\text{T-INVOKE})$$

Figure F.3: EFx's type system.

Appendix G

Core SMT Expressions

We now discuss the expressions that Core SMT supports. Those expressions are common to most SMT solvers, except lambdas which, as mentioned before, are described by the preliminary proposal for SMT-LIB v3.0 and are only implemented by some SMT solvers such as Z3 [MB08].

Figure G.1 provides an overview of all Core SMT expressions. The simplest expressions are literal values representing integers, strings, and booleans. Core SMT supports the typical arithmetic operators ($+$, $-$, $*$, $/$) and boolean operators ($\wedge$, $\vee$, and negation $\neg$) as well as universal and existential quantification, and logical implication. Let bindings define immutable variables. Pattern matching is supported, but the cases must be exhaustive. For example, when pattern matching against an ADT every constructor must be handled. Core SMT supports two types of patterns: constructor patterns $n(\overline{n})$ that match a specific ADT constructor n and binds names to its fields $\overline{n}$, and wildcard patterns that match anything and give it a name n. References v refer to variables that are in scope, e.g. function parameters or variables introduced by a let binding or pattern matching. If statements are supported but an else branch is mandatory, and both branches must type to the same sort. Functions can be called, and type arguments can be provided explicitly to disambiguate polymorphic functions. For example, we defined an ADT $\mathsf{Option}\langle T\rangle$ with two constructors Some and None. When calling the None constructor, we need to explicitly provide a type argument since it cannot be inferred from the call, e.g. $\mathsf{None}\langle\mathsf{int}\rangle()$. Finally, fields of an ADT can be accessed by their name. Arrays and lambdas were already discussed in Section 5.3.1.

$$
\begin{array}{rll}
e ::= & \mathsf{num} \mid \mathsf{str} \mid \mathsf{true} \mid \mathsf{false} & (\textit{primitive values}) \\
\mid & e[\,\vec{e}\,] \mid e[\,\vec{e}\,] := e \mid \lambda(\vec{x} : \vec{\overline{T}}).e & \\
\mid & x \mid e \oplus e \mid e \otimes e \mid \neg e & \\
\mid & \mathsf{match}(e,\ \mathsf{case}(\mathit{ptn},\ e)) & (\textit{pattern matching}) \\
\mid & \mathsf{let}\, x = e\, \mathsf{in}\, e & (\textit{let expression}) \\
\mid & \mathsf{if}(e,\ e,\ e) & (\textit{conditional expression}) \\
\mid & e(e) & (\textit{function call}) \\
\mid & f\langle\overline{T}\rangle(e) & (\textit{function call with explicit type arguments}) \\
\mid & e.v & (\textit{field access}) \\
\mid & \forall(\vec{x} : \vec{\overline{T}}).e \mid \exists(\vec{x} : \vec{\overline{T}}).e & (\textit{quantified formulas}) \\
\mid & e \implies e & (\textit{logical implication}) \\
\mathit{ptn} ::= & K(\overline{x}) \mid x & (\textit{patterns})
\end{array}
$$

Figure G.1: All Core SMT expressions.

Appendix H

EFx's Complete Map Semantics

Section 5.3.3.2 explained how to encode maps in Core SMT using arrays and how to efficiently encode the basic map operations as well as some advanced map operations. This appendix defines the compilation rules for the remaining map operations.

$$\begin{aligned}&[\![e_m.\mathit{bijective}()]\!] = \forall(k_1 : [\![K]\!]_t,\, k_2 : [\![K]\!]_t).\\ &\quad (k_1 \neq k_2 \wedge [\![e_m]\!][\ k_1\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \wedge [\![e_m]\!][\ k_2\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle())\\ &\qquad \implies [\![e_m]\!][\ k_1\ ] \neq [\![e_m]\!][\ k_2\ ] \quad \text{where } \mathit{typeof}(e_m) = \mathtt{Map}\langle K, V\rangle\end{aligned}$$

The `bijective` method checks if the mapping of keys to values is one-to-one. Calls to `bijective` are compiled to a universally quantified formula that checks that every two distinct keys that are present in the map are associated with different values.

$$\begin{aligned}&[\![e_m.\mathit{forall}(e_p)]\!] =\\ &\quad \forall(x : [\![K]\!]_t).[\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \implies [\![e_p]\!][\ x, [\![e_m]\!][\ x\ ].\mathit{value}\ ]\\ &\qquad \text{where } \mathit{typeof}(e_m) = \mathtt{Map}\langle K, V\rangle \text{ and } \mathit{typeof}(e_p) = (K, V) \to \mathtt{bool}\\ &[\![e_m.\mathit{exists}(e_p)]\!] =\\ &\quad \exists(x : [\![K]\!]_t).[\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \wedge [\![e_p]\!][\ x, [\![e_m]\!][\ x\ ].\mathit{value}\ ]\\ &\quad \text{where } \mathit{typeof}(e_m) = \mathtt{Map}\langle K, V\rangle \text{ and } \mathit{typeof}(e_p) = (K, V) \to \mathtt{bool}\end{aligned}$$

When calling `forall` with a predicate e_p of type $(K, V) \to \mathtt{bool}$ on a map e_m of type $\mathtt{Map}\,\langle \mathtt{K}, \mathtt{V}\rangle$, the method checks that the predicate holds for all elements of the map. Similarly, the `exists` method checks that the predicate holds for at least one element of the map. Thus, at least one

key k must exist that is present in the map and whose associated value v fulfills the predicate.

$$
\begin{array}{l}
[\![e_m.\mathit{filter}(e_p)]\!] = \\
\quad \lambda(x : [\![K]\!]_t).\mathsf{if}([\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \ \wedge\ [\![e_p]\!][\ x, [\![e_m]\!][\ x\ ].\mathit{value}\ ], \\
\qquad\qquad\qquad \mathsf{Some}([\![e_m]\!][\ x\ ].\mathit{value}), \\
\qquad\qquad\qquad \mathsf{None}\langle[\![V]\!]_t\rangle()) \\
\quad \text{where } \mathit{typeof}(e_m) = \mathtt{Map}\langle K, V\rangle \text{ and } \mathit{typeof}(e_p) = (K, V) \to \mathtt{bool}
\end{array}
$$

The `filter` method takes a predicate e_p and returns a map containing only the key-value pairs that fulfill the predicate. Calls to `filter` are encoded as a lambda that defines an array containing only the key-value pairs that are in the compiled map ($[\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle()$) and fulfill the predicate ($[\![e_p]\!][\ x, [\![e_m]\!][\ x\ ].\mathit{value}\ ]$).

$$
\begin{array}{l}
[\![e_{m_1}.\mathit{zip}(e_{m_2})]\!] = \\
\quad \lambda(x : [\![K]\!]_t).\mathsf{if}([\![e_{m_1}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \ \wedge\ [\![e_{m_2}]\!][\ x\ ] \neq \mathsf{None}\langle[\![W]\!]_t\rangle(), \\
\qquad\qquad\qquad \mathsf{Some}(\mathit{Tuple\_ctor}([\![e_{m_1}]\!][\ x\ ].\mathit{value}, [\![e_{m_2}]\!][\ x\ ].\mathit{value})), \\
\qquad\qquad\qquad \mathsf{None}\langle[\![\mathtt{Tuple}\langle V, W\rangle]\!]_t\rangle()) \\
\quad \text{where } \mathit{typeof}(e_{m_1}) = \mathtt{Map}\langle K, V\rangle \text{ and } \mathit{typeof}(e_{m_2}) = \mathtt{Map}\langle K, W\rangle
\end{array}
$$

When calling `zip` on a map e_{m_1} of type $\mathtt{Map}\langle K, V\rangle$ with a map e_{m_2} of type $\mathtt{Map}\langle K, W\rangle$, the method returns a map of type $\mathtt{Map}\langle K, \mathtt{Tuple}\langle V, W\rangle\rangle$ that contains only the keys that are present in both maps, i.e. $[\![e_{m_1}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \ \wedge\ [\![e_{m_2}]\!][\ x\ ] \neq \mathsf{None}\langle[\![W]\!]_t\rangle()$, and holds their values in a tuple, i.e. $\mathit{Tuple\_ctor}([\![e_{m_1}]\!][\ x\ ].\mathit{value}, [\![e_{m_2}]\!][\ x\ ].\mathit{value})$.

$$
\begin{array}{l}
[\![e_{m_1}.\mathit{combine}(e_{m_2}, e_f)]\!] = \\
\quad \lambda(x : [\![K]\!]_t).\mathsf{if}([\![e_{m_1}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \ \wedge\ [\![e_{m_2}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle(), \\
\qquad\qquad\qquad \mathsf{Some}([\![e_f]\!][\ [\![e_{m_1}]\!][\ x\ ].\mathit{value}, [\![e_{m_2}]\!][\ x\ ].\mathit{value}\ ]), \\
\qquad\qquad\qquad \mathsf{if}([\![e_{m_1}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle(), \\
\qquad\qquad\qquad\quad [\![e_{m_1}]\!][\ x\ ], \\
\qquad\qquad\qquad\quad \mathsf{if}([\![e_{m_2}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle(), \\
\qquad\qquad\qquad\qquad [\![e_{m_2}]\!][\ x\ ], \\
\qquad\qquad\qquad\qquad \mathsf{None}\langle[\![V]\!]_t\rangle()))) \\
\quad \text{where } \mathit{typeof}(e_{m_1}) = \mathtt{Map}\langle K, V\rangle \text{ and } \mathit{typeof}(e_{m_2}) = \mathtt{Map}\langle K, V\rangle \\
\quad \text{and } \mathit{typeof}(e_f) = (V, V) \to V
\end{array}
$$

The `combine` method combines two maps e_{m_1} and e_{m_2} using a user-provided function e_f. To this end, calls to `combine` are compiled to a lambda that defines an array containing all the keys from e_{m_1} and e_{m_2}. If

a key is present in both maps their values are combined using the provided function e_f. If a key-value pair is present in only one of the maps it is copied to the resulting map. If a key is not present in e_{m_1} nor in e_{m_2} then it is not present in the resulting map.

$$
\begin{array}{l}
[\![e_m.toSet()]\!] = \\
\quad \lambda(x : \mathsf{Tuple}\langle [\![K]\!]_t, [\![V]\!]_t \rangle). \\
\quad\quad [\![e_m]\!][\ x.fst\ ] = \mathsf{Some}(x.snd) \\
\quad\quad\quad \text{where } \mathit{typeof}(e_m) = \mathtt{Map}\langle K, V \rangle
\end{array}
$$

Finally, the `toSet` method turns a map e_m of type $\mathtt{Map}\,\langle \mathtt{K}, \mathtt{V} \rangle$ into a set of type $\mathtt{Set}\,\langle \mathtt{Tuple}\langle \mathtt{K}, \mathtt{V} \rangle \rangle$ where each key-value pair is represented as a tuple. Calls to `toSet` are compiled to a lambda that checks that the first element of the tuple (i.e. the key) is associated with the second element of the tuple (i.e. the value) in the map e_m.

Appendix I

Implementation and Verification of the Buggy Map CRDT

Section 6.5.2.2 reported on our experience implementing and verifying the buggy and corrected map CRDTs proposed by Kleppmann [Kle22]. In this appendix, we explain the implementation and verification of the *buggy* map CRDT in detail using code examples. We also discuss the counterexample found by VeriFx.

I.1 Original Specification

The buggy map CRDT is a replicated dictionary storing key-value pairs where the values are regular values (i.e. no nested CRDTs). Algorithm 12 shows the specification of the buggy map CRDT. It defines a `read` operation to fetch the value associated to a certain key, and two update operations: `set` and `delete` which assign a value to a key, respectively, delete a certain key. Every operation consists of two parts, a prepare phase (denoted "on request") that prepares a message to be broadcast to every replica (including itself), and an effect phase (denoted "on delivering") that applies the incoming message. We briefly explain both update operations:

`set`(k, v). When preparing a `set` operation that assigns a value v to a key k, the replica generates a new and globally unique timestamp t and

Algorithm 12 The buggy map CRDT algorithm, taken from [Kle22].

on initialisation **do**
    $values := \{\}$
end on

on request to read value for key k **do**
    if $\exists t, v.(t, k, v) \in values$ **then return** v **else return** null
end on

on request to set key k to value v **do**
    $t :=$ newTimestamp() ▷ globally unique, e.g. Lamport timestamp
    broadcast $(\texttt{set}, t, k, v)$ by causal broadcast (including to self)
end on

on delivering $(\texttt{set}, t, k, v)$ by causal broadcast **do**
    $previous := \{(t', k', v') \in values \mid k' = k\}$
    if $previous = \{\} \vee \forall (t', k', v') \in previous.\, t' < t$ **then**
        $values := (values \setminus previous) \cup \{(t, k, v)\}$
    end if
end on

on request to delete key k **do**
    if $\exists t, v.\, (t, k, v) \in values$ **then**
        broadcast $(\texttt{delete}, t)$ by causal broadcast (including to self)
    end if
end on

on delivering $(\texttt{delete}, t)$ by causal broadcast **do**
    $values := \{(t', k', v') \in values \mid t' \neq t\}$
end on

Listing I.1: Excerpt from the implementation of the buggy map CRDT in VeriFx.

```
enum MapOp[K, V] { Put(k: K, v: V) | Delete(k: K) }
enum MapMsg[K, V] {
  PutMsg(t: Clock, k: K, v: V) |
  DeleteMsg(t: Clock, k: K) |
  NopMsg()
}
class KMap[K, V](clock: Clock, values: Map[K, Tuple[Clock, V]])
    extends CmRDT[MapOp[K, V], MapMsg[K, V], KMap[K, V]] {
  def contains(k: K): Boolean = this.values.contains(k)
  def get(k: K): V = this.values.get(k).snd

  // Prepare phase for the "put" operation
  // "put" corresponds to the "set" operation in the
    specification
  def preparePut(k: K, v: V) = {
    val t = this.clock
    new PutMsg(t, k, v)
  }
  // Effect phase for incoming "put" messages
  def put(t: Clock, k: K, v: V) = {
    val newClock = this.clock.sync(t)
    if (!this.values.contains(k) ||
        this.values.get(k).fst.smaller(t))
      new KMap(newClock, this.values.add(k, new Tuple(t, v)))
    else
      new KMap(newClock, this.values)
  }

  // Prepare phase for the "delete" operation
  def prepareDelete(k: K) = {
    if (this.values.contains(k)) {
      val t = this.values.get(k).fst
      new DeleteMsg[K, V](t, k)
    }
    else
      new NopMsg[K, V]()
  }
  // Effect phase for incoming "delete" messages
  def delete(t: Clock, k: K) = {
    if (this.values.contains(k) && this.values.get(k).fst == t)
      new KMap(this.clock, this.values.remove(k))
    else
      new KMap(this.clock, this.values)
  }

  override def equals(that: KMap[K, V]) =
    this.values == that.values
}
```

broadcasts a $(\texttt{set}, t, k, v)$ message. When receiving such a message, the replica checks if it already stores a value for this key. If this is not the case, or if the previous value has a smaller timestamp $t' < t$, then it assigns the incoming value v to the key k, thereby, overriding any previous value. On the other hand, if the previous value has a bigger timestamp, then the incoming `set` message is ignored and the previous value is kept.

`delete`(k). When preparing a `delete` operation that deletes a key k, the replica fetches the timestamp t at which that key was inserted and broadcasts a $(\texttt{delete}, t)$ message. Note that the key itself is not added to the message because `set` always inserts a single key with a unique timestamp, hence, the timestamp t uniquely identifies the key. When receiving a $(\texttt{delete}, t)$ message, the replica removes the key that was inserted at timestamp t (if it is still present).

I.2 Implementation in VeriFx

Listing I.1 shows the implementation of the buggy map CRDT in VeriFx. Every replica (i.e. every instance of the `KMap` class) maintains a local Lamport clock (consisting of a counter and a replica identifier) and keeps a dictionary that maps keys to timestamped values (i.e. a tuple containing a timestamp and a value). This implementation strategy is slightly different from Algorithm 12 but more efficient because a dictionary allows for constant-time lookup, insertion, and deletion. We also extended the `DeleteMsg` such that it not only contains the timestamp t but also the key to be deleted (Line 4). This allows for an efficient implementation of `delete` since the replica knows which key to delete and does not have to loop over the map to find the key whose value has timestamp t.

We override equality - which by default is structural equality - because replicas have different Lamport clocks [Lam78] as our implementation of the clocks keeps a unique replica identifier. Hence, two replicas are considered equal if they have the same values, independent of their clocks. We also renamed the `set` operation to `put`. The remainder of the implementation is a straightforward translation from the specification.

Listing I.2: Encoding the assumptions of the Map CRDT in VeriFx.

```
override def reachable(): Boolean = {
  // every value must have a unique timestamp
  !(exists(k1: K, k2: K) {
    k1 != k2 &&
    this.values.get(k1).fst == this.values.get(k2).fst
  }) &&
  // All the values in the map must have a timestamp < than our
    local clock (since we sync our clock on incoming updates)
  this.values.values().forall((entry: Tuple[Clock, V]) =>
    entry.fst.counter < this.clock.counter)
}
private def noValueFromFuture(r1: KMap[K, V], r2: KMap[K, V]) {
  r1.values.values().forall((entry: Tuple[Clock, V]) => {
    val t = entry.fst
    (t.replica == r2.clock.replica) =>:
      (t.counter < r2.clock.counter)
  })
}
override def compatible(that: KMap[K, V]) = {
  // replicas have unique IDs
  (this.clock.replica != that.clock.replica) &&
  // we have no value from the future of the other replica
  this.noValueFromFuture(this, that) &&
  // the other did not observe a value from our future
  this.noValueFromFuture(that, this) &&
  // unique timestamps
  !(exists(k1: K, k2: K) {
    k1 != k2 && this.values.get(k1).fst ==
    that.values.get(k2).fst
  }) &&
  // replicas cannot store different values for the same key
    and timestamp
  !(exists(k: K) {
    val thisTuple = this.values.get(k)
    val thisTimestamp = thisTuple.fst
    val thisValue = thisTuple.snd
    val thatTuple = that.values.get(k)
    val thatTimestamp = thatTuple.fst
    val thatValue = thatTuple.snd
    (thisTimestamp == thatTimestamp) && (thisValue != thatValue)
  })
}
```

I.3 Verification in VeriFx

After implementing the buggy map CRDT in VeriFx we proceeded to the verification of the map. As explained in Section 6.5.2.2, VeriFx returned invalid counterexamples because it is not aware of the CRDT's assumptions which are *implicit* in the design. For instance, VeriFx does not know that replicas have unique IDs nor does it know the relation between a replica's clock and the values it observed. We need to encode these assumptions explicitly such that VeriFx does not consider cases that cannot occur in practice. To this end, we override the `reachable` and `compatible` predicates (cf. Section 6.4.1.2). The former defines which states are reachable (i.e. valid), while the latter defines which replicas are compatible.

Listing I.2 shows the implementation of the `reachable` and `compatible` predicates. First, we define a state to be reachable iff every value has a unique timestamp (Line 3 to 6) and all values have a timestamp whose count is smaller than the replica's local clock (Line 8). The latter property follows from the fact that the dictionary is constructed by successive insertions and every insertion synchronizes the replica's clock with the timestamp of the inserted element.

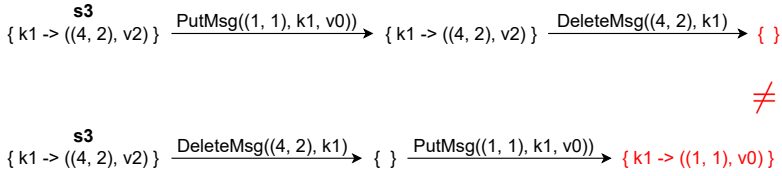
Second, we define two replicas to be compatible iff:

- they have unique IDs (Line 19),
- they did not observe values with a timestamp that is bigger than the current clock of the replica that inserted that value (Line 21 to 23) because that would mean that some replica observed a value from the future of the origin replica which is not possible,
- they do not have the same timestamp for different keys (Line 25 to 27) because every insertion inserts a single key with a unique timestamp,
- for every key k for which they store the same timestamp t they also store the same value v (Line 29 to 37) because every timestamp uniquely identifies one insertion: `PutMsg(t, k, v)`.

Clearly, the above assumptions are not straightforward and are in fact implicit in the original specification, but are nevertheless vital to the correctness of the algorithm. In practice, many CRDTs make similar implicit assumptions which is the reason they are complex and difficult to get right.

```
enum V { v0 | v2 }
enum K { k1 }
val s1 = KMap(Clock(1, 1), Map())
val s2 = KMap(Clock(2, 9), Map(k1 -> (Clock(4, 2), v2)))
val s3 = KMap(Clock(3, 3), Map(k1 -> (Clock(4, 2), v2)))
val x = Put(k1, v0) // operation generated by s1
// The prepare phase will broadcast the following message:
// s1.preparePut(k1, v0) = PutMsg(Clock(1, 1), k1, v0))
val y = Delete(k1) // operation generated by s2
// s2.prepareDelete(k1) = DeleteMsg(Clock(4, 2), k1)
```

(a) Simplified counterexample returned by VeriFx.



(b) Visualization of the counterexample returned by VeriFx.

Figure I.1: Counterexample for the buggy Map CRDT, found by VeriFx.

Counterexample. After defining all assumptions described above, VeriFx found a valid counterexample which is shown in Fig. I.1a. We simplified the counterexample by renaming the keys and values and removing those that do not affect the outcome. The counterexample is equivalent to the one that was found manually by Nair (cf. [Kle22]). It consists of a corner case in which the `Put` and `Delete` operations do not commute and thus may cause replicas to diverge.

Recall that a counterexample is a mapping from variables (defined by the proof) to values that break the proof. In this case, the `CmRDTProof2` trait (cf. Section 6.4.1.2) that was used to check commutativity of the operations, defines three variables `s1`, `s2`, and `s3` representing the state of the replicas, and two variables `x = Put(k1, v0)` and `y = Delete(k1)` representing concurrent operations that were generated by replica `s1` and `s2` respectively. These replicas first prepare a message for the operations (respectively, `PutMsg(Clock(1, 1), k1, v0))` and `DeleteMsg(Clock(4, 2), k1)`) and broadcast those messages to every replica. Every replica receives these messages, possibly in a different order, and applies them.

Depending on the order in which replica `s3` applies the operations, the outcome is different. This is visualized in Fig. I.1b. If `s3` first processes the `DeleteMsg(Clock(4, 2), k1)` message then key `k1` is gone because the stored timestamp matches the timestamp that was requested to delete. Afterwards, when processing the `PutMsg(Clock(1, 1), k1, v0)` message, the replica will add key `k1` with value `v0`. When applying the operations the other way around, the outcome is different because the `PutMsg(Clock(1, 1), k1, v0)` message is ignored since its timestamp is smaller than the timestamp `s3` currently stores for that key: `Clock(1, 1) < Clock(4, 2)`. Later, when processing the `DeleteMsg(Clock(4, 2), k1)` message, `s3` effectively deletes key `k1` because the timestamp matches the one that is stored. Thus, after the first execution, the resulting state contains key `k1`, whereas, after the second execution, `k1` is not present in the map. This explains the divergence bug.

Bibliography

[Aha+95] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. "Causal memory: Definitions, implementation, and programming". In: *Distributed Computing* 9.1 (1995), pp. 37–49. DOI: 10.1007/BF01784241.

[Akk] Akka. *Implementation of a Positive-Negative Counter CRDT in Akka.* https://github.com/akka/akka/blob/main/akka-distributed-data/src/main/scala/akka/cluster/ddata/PNCounter.scala. Accessed: 04-07-2022.

[Alo+08] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. "The Cost of Serializability on Platforms That Use Snapshot Isolation". In: *2008 IEEE 24th International Conference on Data Engineering*. 2008, pp. 576–585. DOI: 10.1109/ICDE.2008.4497466.

[Anta] AntidoteDB. *Implementation of a Disable-Wins Flag CRDT in AntidoteDB.* https://github.com/AntidoteDB/antidote/blob/master/apps/antidote_crdt/src/antidote_crdt_flag_dw.erl. Accessed: 19-07-2022.

[Antb] AntidoteDB. *Implementation of an Enable-Wins Flag CRDT in AntidoteDB.* https://github.com/AntidoteDB/antidote/blob/master/apps/antidote_crdt/src/antidote_crdt_flag_ew.erl. Accessed: 19-07-2022.

[ASB15] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. "Efficient State-based CRDTs by Delta-Mutation". In: *Int. Conference on Networked Systems* (May 13–15, 2015). Ed. by Ahmed Bouajjani and Hugues Fauconnier. Springer-Verslag. Agadir, Morocco, 2015, pp. 62–76. DOI: 10.1007/978-3-319-26850-7_5.

[Att+16] Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. "Specification and Complexity of Collaborative Text Editing". In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. PODC '16. Chicago, Illinois, USA: Association for Computing Machinery, 2016, pp. 259–268. ISBN: 9781450339643. DOI: 10.1145/2933057.2933090.

[Bai] Peter Bailis. *Stickiness and Client-Server Session Guarantees*. http://www.bailis.org/blog/stickiness-and-client-server-session-guarantees/. Accessed: 23-08-2022.

[Bai+13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. "Highly Available Transactions: Virtues and Limitations". In: *Proc. VLDB Endow.* 7.3 (Nov. 2013), pp. 181–192. ISSN: 2150-8097. DOI: 10.14778/2732232.2732237.

[Bai+14] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. "Coordination Avoidance in Database Systems". In: *Proc. VLDB Endow.* 8.3 (Nov. 2014), pp. 185–196. ISSN: 2150-8097. DOI: 10.14778/2735508.2735509.

[Bal+15] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. "Putting Consistency Back into Eventual Consistency". In: *10th European Conference on Computer Systems*. EuroSys '15. Bordeaux, France, 2015, 6:1–6:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741972.

[Bal+18] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. "IPA: Invariant-preserving Applications for Weakly Consistent Replicated Databases". In: *Proc. VLDB Endow.* 12.4 (Dec. 2018), pp. 404–418. ISSN: 2150-8097. DOI: 10.14778/3297753.3297760.

[Bar+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer,

Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387. DOI: 10.1007/11804192_17.

[BAS17] Carlos Baquero, Paulo S. Almeida, and Ali Shoker. "Pure Operation-Based Replicated Data Types". In: *CoRR* abs/1710.04469 (2017). eprint: 1710.04469.

[BFP] Dina Borrego, Carla Ferreira, and Nuno Preguiça. "Verificação e Reforço de Invariantes Aplicacionais no Antidote SQL". In: *INForum 2022*, to appear.

[BG13] Peter Bailis and Ali Ghodsi. "Eventual Consistency Today: Limitations, Extensions, and Beyond: How Can Applications Be Built on Eventually Consistent Infrastructure given No Guarantee of Safety?" In: *Queue* 11.3 (Mar. 2013), pp. 20–32. ISSN: 1542-7730. DOI: 10.1145/2460276.2462076.

[BHG87] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading, 1987.

[Bie+12] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. *An optimized conflict-free replicated set*. 2012. DOI: 10.48550/ARXIV.1210.3368.

[Bir+82] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. "Grapevine: An Exercise in Distributed Computing". In: *Commun. ACM* 25.4 (Apr. 1982), pp. 260–274. ISSN: 0001-0782. DOI: 10.1145/358468.358487.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. "The Spec# Programming System: An Overview". In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Ed. by Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 49–69. ISBN: 978-3-540-30569-9. DOI: 10.1007/978-3-540-30569-9_3.

[BN10] Jasmin Christian Blanchette and Tobias Nipkow. "Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder". In: *Interactive Theorem Proving*.

Ed. by Matt Kaufmann and Lawrence C. Paulson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 131–146. ISBN: 978-3-642-14052-5.

[Böh+11] Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. "Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL". In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 183–198. ISBN: 978-3-642-25379-9. DOI: 10.1007/978-3-642-25379-9_15.

[BR95] Adel Bouhoula and Michaël Rusinowitch. "Implicit induction in conditional theories". In: *Journal of automated reasoning* 14.2 (1995), pp. 189–235. DOI: 10.1007/BF00881856.

[Bre00] Eric A. Brewer. "Towards Robust Distributed Systems (Abstract)". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: Association for Computing Machinery, 2000, p. 7. ISBN: 1581131836. DOI: 10.1145/343477.343502.

[Bre12] Eric Brewer. "CAP Twelve years later: How the "Rules" have Changed". In: *Computer* 45 (Feb. 2012), pp. 23–29. DOI: 10.1109/MC.2012.37.

[BSW04] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. "From session causality to causal consistency". In: *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. 2004, pp. 152–158. DOI: 10.1109/EMPDP.2004.1271440.

[Bur+12] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. "Cloud Types for Eventual Consistency". In: *26th European Conference on Object-Oriented Programming*. ECOOP'12. Beijing, China: Springer-Verlag, 2012, pp. 283–307. ISBN: 978-3-642-31056-0. DOI: 10.1007/978-3-642-31057-7_14.

[Bur+14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. "Replicated Data Types: Specification, Verification, Optimality". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: As-

sociation for Computing Machinery, 2014, pp. 271–284. ISBN: 9781450325448. DOI: 10.1145/2535838.2535848.

[BW10] Sascha Böhme and Tjark Weber. "Fast LCF-style proof reconstruction for Z3". In: *International Conference on Interactive Theorem Proving*. Springer. 2010, pp. 179–194. DOI: 10.1007/978-3-642-14052-5_14.

[Cal+15] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving Fast with Software Verification". In: *NASA Formal Methods*. Ed. by Klaus Havelund, Gerard Holzmann, and Rajeev Joshi. Cham: Springer International Publishing, 2015, pp. 3–11. ISBN: 978-3-319-17524-9. DOI: 10.1007/978-3-319-17524-9_1.

[Cet+14] Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, Kristin Tufte, Hao Wang, and Stanley Zdonik. "S-Store: A Streaming NewSQL System for Big Velocity Applications". In: *Proc. VLDB Endow.* 7.13 (Aug. 2014), pp. 1633–1636. ISSN: 2150-8097. DOI: 10.14778/2733004.2733048.

[De +19a] Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. "A Generic Replicated Data Type for Strong Eventual Consistency". In: *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362764. DOI: 10.1145/3301419.3323974.

[De +19b] Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. "Putting Order in Strong Eventual Consistency". In: *Distributed Applications and Interoperable Systems*. Ed. by José Pereira and Laura Ricci. Cham: Springer International Publishing, 2019, pp. 36–56. ISBN: 978-3-030-22496-7. DOI: 10.1007/978-3-030-22496-7_3.

[De +20] Kevin De Porre, Florian Myter, Christophe Scholliers, and Elisa Gonzalez Boix. "CScript: A distributed programming language for building mixed-consistency applications". In: *J. Parallel Distributed Comput.* 144 (2020), pp. 109–123. DOI: 10.1016/j.jpdc.2020.05.010.

[De +21] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. "ECROs: Building Global Scale Systems from Sequential Code". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Nov. 2021). DOI: 10.1145/3485484.

[DeC+07] 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: *21st ACM SIGOPS Symp. on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1323293.1294281.

[DFGed] Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. "VeriFx: Correct Replicated Data Types for the Masses". In: *Proc. ACM Program. Lang.* OOPSLA (Submitted).

[DG19] Kevin De Porre and Elisa Gonzalez Boix. "Squirrel: An Extensible Distributed Key-Value Store". In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*. META 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 21–30. ISBN: 9781450369855. DOI: 10.1145/3358502.3361271.

[Dim+14] Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and Eric Koskinen. "Commutativity Race Detection". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 305–315. ISBN: 9781450327848. DOI: 10.1145/2594291.2594322.

[EG89] C. A. Ellis and S. J. Gibbs. "Concurrency Control in Groupware Systems". In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. SIGMOD '89. Portland, Oregon, USA: Association for Computing Machinery, 1989, pp. 399–407. ISBN: 0897913175. DOI: 10.1145/67544.66963.

[EJ09] Cecchet Emmanuel and Marguerite Julie. *RUBiS: Rice University Bidding System*. http://rubis.ow2.org/. 2009.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 — Where Programs Meet Provers". In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–128. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_8.

[GL02] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601.

[Gom+17] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. "Verifying Strong Eventual Consistency in Distributed Systems". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 109:1–109:28. ISSN: 2475-1421. DOI: 10.1145/3133933.

[Goo] Google. *Protocol Buffers*. https://developers.google.com/protocol-buffers. Accessed: 10-10-2022.

[Got+16] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. "'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 371–384. ISBN: 9781450335492.

[GPS16] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. "Incremental Consistency Guarantees for Replicated Objects". In: *12th USENIX Symposium on Operat-*

ing Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. USENIX Association, 2016, pp. 169–184. ISBN: 978-1-931971-33-1. DOI: 10.5555/3026877.3026891.

[Hel15] Pat Helland. "Immutability Changes Everything: We Need It, We Can Afford It, and the Time is Now." In: *Queue* 13.9 (Nov. 2015), pp. 101–125. ISSN: 1542-7730. DOI: 10.1145/2857274.2884038.

[HL19] Farzin Houshmand and Mohsen Lesani. "Hamsaz: Replication Coordination Analysis and Synthesis". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290387.

[Hol+16] Brandon Holt, James Bornholt, Irene Zhang, Dan R. K. Ports, Mark Oskin, and Luis Ceze. "Disciplined Inconsistency with Consistency Types". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*. ACM, 2016, pp. 279–293. DOI: 10.1145/2987550.2987559.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972.

[Imi+03] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. "Proving Correctness of Transformation Functions in Real-Time Groupware". In: *Proceedings of the Eighth Conference on European Conference on Computer Supported Cooperative Work*. ECSCW'03. Helsinki, Finland: Kluwer Academic Publishers, 2003, pp. 277–293. DOI: 10.5555/1241889.1241904.

[Imi22] Abdessamad Imine. *Exchange of mails regarding OT, and unpublished register and stack designs.* personal communication. Mar. 10, 2022.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 2001), pp. 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505.

[Jep] Jepsen, LLC. *Linearizability*. `https://jepsen.io/consistency/models/linearizable`. Accessed: 2-12-2022.

[JR18] Radha Jagadeesan and James Riely. "Eventual Consistency for CRDTs". In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 968–995. ISBN: 978-3-319-89884-1. DOI: 10.1007/978-3-319-89884-1_34.

[Jua+16] Rubén de Juan-Marín, Hendrik Decker, José Enrique Armendáriz-Íñigo, José M Bernabéu-Aubán, and Francesc D Muñoz-Escoí. "Scalability approaches for causal multicast: a survey". In: *Computing* 98.9 (2016), pp. 923–947. DOI: 10.1007/s00607-015-0479-0.

[Kak+18] Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. "Safe Replication through Bounded Concurrency Verification". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276534.

[Kak+19] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. "Mergeable Replicated Data Types". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360580.

[KB17] Martin Kleppmann and Alastair R Beresford. "A Conflict-Free Replicated JSON Datatype". In: *IEEE Trans. on Parallel and Distributed Systems*. TPDS'17 28.10 (2017), pp. 2733–2746. DOI: 10.1109/TPDS.2017.2697382.

[Ker+01] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. "The IceCube Approach to the Reconciliation of Divergent Replicas". In: *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*. PODC '01. Newport, Rhode Island, USA: Association for Computing Machinery, 2001, pp. 210–218. ISBN: 1581133839. DOI: 10.1145/383962.384020.

[KJ14] Gowtham Kaki and Suresh Jagannathan. "A Relational Framework for Higher-Order Shape Analysis". In: *SIGPLAN Not.* 49.9 (Aug. 2014), pp. 311–324. ISSN: 0362-1340. DOI: 10.1145/2692915.2628159.

[Kle15] Martin Kleppmann. *A Critique of the CAP Theorem*. 2015. DOI: 10.48550/ARXIV.1509.05393.

[Kle22] Martin Kleppmann. *Assessing the understandability of a distributed algorithm by tweeting buggy pseudocode*. Tech. rep. UCAM-CL-TR-969. University of Cambridge, Computer Laboratory, May 2022. DOI: 10.48456/tr-969. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-969.pdf.

[Köh+20] Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. "Rethinking Safe Consistency in Distributed Object-Oriented Programming". In: *Proc. ACM Program. Lang.* OOPSLA (2020). DOI: 10.1145/3428256.

[Kul+11] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. "Exploiting the Commutativity Lattice". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 542–555. ISBN: 9781450306638. DOI: 10.1145/1993498.1993562.

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563.

[Lam94] Leslie Lamport. "The Temporal Logic of Actions". In: *ACM Trans. Program. Lang. Syst.* 16.3 (May 1994), pp. 872–923. ISSN: 0164-0925. DOI: 10.1145/177492.177726.

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4. DOI: 10.1007/978-3-642-17511-4_20.

[LF21] Hongjin Liang and Xinyu Feng. "Abstraction for Conflict-Free Replicated Data Types". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021,

pp. 636–650. ISBN: 9781450383912. DOI: 10.1145/3453483.3454067.

[LHL20] Xiao Li, Farzin Houshmand, and Mohsen Lesani. "Hampa: Solver-Aided Recency-Aware Replication". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 324–349. DOI: 10.1007/978-3-030-53288-8_16.

[Li+12] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. "Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 265–278. ISBN: 9781931971966. DOI: 10.5555/2387880.2387906.

[Li+14] 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. Philadelphia, PA: USENIX Association, 2014, pp. 281–292. ISBN: 978-1-931971-10-2. DOI: 10.5555/2643634.2643664.

[Lig] Lightbend Inc. *Serialization*. https://doc.akka.io/docs/akka/current/serialization.html. Accessed: 10-10-2022.

[Lin] Greg Linden. *Slides from my talk at Stanford*. http://glinden.blogspot.com/2006/12/slides-from-my-talk-at-stanford.html. Accessed: 14-10-2022.

[Liu+20] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. "Verifying Replicated Data Types with Typeclass Refinements in Liquid Haskell". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428284.

[LL04] Du Li and Rui Li. "Preserving Operation Effects Relation in Group Editors". In: *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*. CSCW '04.

Chicago, Illinois, USA: Association for Computing Machinery, 2004, pp. 457–466. ISBN: 1581138105. DOI: 10.1145/1031607.1031683.

[LM10] K. Rustan M. Leino and Michał Moskal. "Usable Auto-Active Verification". In: *Usable Verification Workshop*. 2010. URL: http://fm.csl.sri.com/UV10/.

[Lop+19] Pedro Lopes, João Sousa, Valter Balegas, Carla Ferreira, Ségio Duarte, Annette Bieniusa, Rodrigo Rodrigues, and Nuno Preguiça. *Antidote SQL: Relaxed When Possible, Strict When Necessary*. 2019. DOI: 10.48550/ARXIV.1902.03576.

[LPR18] Cheng Li, Nuno Preguiça, and Rodrigo Rodrigues. "Fine-grained consistency for geo-replicated systems". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 359–372. ISBN: 978-1-931971-44-7. DOI: 10.5555/3277355.3277391.

[mac] macrotrends. *Amazon Revenue 2010-2022*. https://www.macrotrends.net/stocks/charts/AMZN/amazon/revenue. Accessed: 14-10-2022.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[MB09] Leonardo de Moura and Nikolaj Bjørner. "Generalized, efficient array decision procedures". In: *2009 Formal Methods in Computer-Aided Design*. 2009, pp. 45–52. DOI: 10.1109/FMCAD.2009.5351142.

[MM18] Matthew Milano and Andrew C. Myers. "MixT: A Language for Mixing Consistency in Geodistributed Transactions". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 226–241. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192375.

[MSD18] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. "A CAPable Distributed Programming Model". In: *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2018. Boston, MA, USA: ACM, 2018, pp. 88–98. ISBN: 978-1-4503-6031-9. DOI: 10.1145/3276954.3276957.

[MV15] Christopher Meiklejohn and Peter Van Roy. "Lasp: A Language for Distributed, Coordination-free Programming". In: *17th Int. Symp. on Principles and Practice of Declarative Programming*. PPDP '15. Siena, Italy, 2015, pp. 184–195. ISBN: 978-1-4503-3516-4. DOI: 10.1145/2790449.2790525.

[Néd+13] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. "LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing". In: *Proc. of the 2013 ACM Symposium on Document Engineering*. DocEng '13. Florence, Italy, Sept. 2013, pp. 37–46. ISBN: 978-1-4503-1789-4. DOI: 10.1145/2494266.2494278.

[Nie+22] Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. "Modular Verification of Op-Based CRDTs in Separation Logic". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563351. URL: https://doi.org/10.1145/3563351.

[NJ19] Kartik Nagar and Suresh Jagannathan. "Automated Parameterized Verification of CRDTs". In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Cham: Springer International Publishing, 2019, pp. 459–477. ISBN: 978-3-030-25543-5. DOI: 10.1007/978-3-030-25543-5_26.

[NPS20] Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. "Proving the Safety of Highly-Available Distributed Objects". In: *Programming Languages and Systems*. Ed. by Peter Müller. Cham: Springer International Publishing, 2020, pp. 544–571. ISBN: 978-3-030-44914-8. DOI: 10.1007/978-3-030-44914-8_20.

[OHe18] Peter W. O'Hearn. "Continuous Reasoning: Scaling the Impact of Formal Methods". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*.

LICS '18. Oxford, United Kingdom: Association for Computing Machinery, 2018, pp. 13–25. ISBN: 9781450355834. DOI: 10.1145/3209108.3209109.

[Ope] OpenJDK. *jmh - OpenJDK*. https://openjdk.java.net/projects/code-tools/jmh/. Accessed: 13-05-2020.

[Ost+06] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. "Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems". In: *2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 2006, pp. 1–10. DOI: 10.1109/COLCOM.2006.361867.

[PK07] David J. Pearce and Paul H. J. Kelly. "A Dynamic Topological Sort Algorithm for Directed Acyclic Graphs". In: *J. Exp. Algorithmics* 11 (Feb. 2007), 1.7–es. ISSN: 1084-6654. DOI: 10.1145/1187436.1210590.

[Pon14] Julien Ponge. *Avoiding Benchmarking Pitfalls on the JVM*. https://www.oracle.com/technical-resources/articles/java/architect-benchmarking.html. Accessed: 13-05-2020. July 2014.

[Pre18] Nuno Preguiça. "Conflict-free Replicated Data Types: An Overview". In: (2018). DOI: 10.48550/ARXIV.1806.10254.

[Ran+13] Aurel Randolph, Hanifa Boucheneb, Abdessamad Imine, and Alejandro Quintero. "On Consistency of Operational Transformation Approach". In: *Electronic Proceedings in Theoretical Computer Science* 107 (Feb. 2013), pp. 45–59. DOI: 10.4204/eptcs.107.5.

[RK15] Andrew Reynolds and Viktor Kuncak. "Induction for SMT Solvers". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 80–98. ISBN: 978-3-662-46081-8. DOI: 10.1007/978-3-662-46081-8_5.

[RNG96] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. "An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors". In:

Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work. CSCW '96. Boston, Massachusetts, USA: Association for Computing Machinery, 1996, pp. 288–297. ISBN: 0897917650. DOI: 10.1145/240080.240305.

[SCF97] Maher Suleiman, Michèle Cart, and Jean Ferrié. "Serialization of Concurrent Operations in a Distributed Collaborative Environment". In: *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge*. GROUP '97. Phoenix, Arizona, USA: Association for Computing Machinery, 1997, pp. 435–445. ISBN: 0897918975. DOI: 10.1145/266838.267369.

[SCF98] Maher Suleiman, Michèle Cart, and Jean Ferrié. "Concurrent Operations in a Distributed and Mobile Collaborative Environment". In: *Proceedings of the Fourteenth International Conference on Data Engineering*. ICDE '98. USA: IEEE Computer Society, 1998, pp. 36–45. ISBN: 0818682892. DOI: 10.5555/645483.656223.

[SE98] Chengzheng Sun and Clarence Ellis. "Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements". In: *Proc. of the 1998 ACM Conference on Computer Supported Cooperative Work*. CSCW '98. Seattle, Washington, USA, 1998, pp. 59–68. ISBN: 1-58113-009-0. DOI: 10.1145/289444.289469.

[Sha+11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50.

[Sha+11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-free Replicated Data Types". In: *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. SSS'11. Springer-Verslag. Grenoble, France, 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29.

[Sha17] Marc Shapiro. "Replicated Data Types". In: *Encyclopedia Of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu.

Vol. Replicated Data Types. Springer-Verlag, July 2017, pp. 1–5. DOI: 10.1007/978-1-4899-7993-3_80813-1.

[SKJ15] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. "Declarative Programming over Eventually Consistent Data Stores". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 413–424. ISBN: 9781450334686. DOI: 10.1145/2737924.2737981.

[Sun+98] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. "Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems". In: *ACM Trans. Comput.-Hum. Interact.* 5.1 (Mar. 1998), pp. 63–108. ISSN: 1073-0516. DOI: 10.1145/274444.274447.

[Ter+13] 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: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 309–324. ISBN: 9781450323888. DOI: 10.1145/2517349.2522731.

[Ter+94] 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: *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems*. PDIS '94. Autin, Texas, USA: IEEE Computer Society Press, 1994, pp. 140–150. ISBN: 0818664010. DOI: 10.5555/381992.383631.

[Ter+95] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In: *15th ACM Symp. on Operating Systems Principles*. Ed. by M. B. Jones. SOSP '95. Copper Mountain, Colorado,

USA, 1995, pp. 172–182. ISBN: 0-89791-715-4. DOI: 10.1145/224056.224070.

[TS] Doug Tollefson and Andrew Spyker. *Acme Air*. https://github.com/acmeair/acmeair. Accessed: 27-04-2022.

[TV07] Andrew S Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. 2nd ed. Upper Saddle River, New Jersey, USA: Prentice-Hall, 2007.

[Vaz+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. "Refinement Types for Haskell". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 269–282. ISBN: 9781450328739. DOI: 10.1145/2628136.2628161.

[Vaz+17] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. "Refinement Reflection: Complete Verification with SMT". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158141.

[Vog09] Werner Vogels. "Eventually Consistent". In: *Communications of the ACM* 52.1 (2009), pp. 40–44. DOI: 10.1145/1435417.1435432.

[VV16] Paolo Viotti and Marko Vukoliundefined. "Consistency in Non-Transactional Distributed Storage Systems". In: *ACM Comput. Surv.* 49.1 (June 2016). ISSN: 0360-0300. DOI: 10.1145/2926965.

[Wal+97] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. "A note on distributed computing". In: *Mobile Object Systems Towards the Programmable Internet*. Ed. by Jan Vitek and Christian Tschudin. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 49–64. ISBN: 978-3-540-68705-4. DOI: 10.1007/3-540-62852-5_6.

[Wan+19] Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. "Replication-Aware Linearizability". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019.

Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 980–993. ISBN: 9781450367127. DOI: 10.1145/3314221.3314617.

[Web08] Tjark Weber. "Sat-based finite model generation for higher-order logic". PhD thesis. Technische Universität München, 2008.

[WH18] Michael Whittaker and Joseph M Hellerstein. "Interactive checks for coordination avoidance". In: *Proceedings of the VLDB Endowment* 12.1 (2018), pp. 14–27. DOI: 10.14778/3275536.3275538.

[WMM20] Matthew Weidner, Heather Miller, and Christopher Meiklejohn. "Composing and Decomposing Op-Based CRDTs with Semidirect Products". In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3408976.

[ZBP14] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. "Formal Specification and Verification of CRDTs". In: *Formal Techniques for Distributed Objects, Components, and Systems*. Ed. by Erika Ábrahám and Catuscia Palamidessi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 33–48. ISBN: 978-3-662-43613-4. DOI: 10.1007/978-3-662-43613-4_3.

[ZBP20] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. "Combining State- and Event-Based Semantics to Verify Highly Available Programs". In: *Formal Aspects of Component Software*. Ed. by Farhad Arbab and Sung-Shik Jongmans. Cham: Springer International Publishing, 2020, pp. 213–232. ISBN: 978-3-030-40914-2. DOI: 10.1007/978-3-030-40914-2_11.

[ZH18] Xin Zhao and Philipp Haller. "Observable atomic consistency for CvRDTs". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 2018, pp. 23–32. DOI: 10.1145/3281366.3281372.

[ZH20] Xin Zhao and Philipp Haller. "Replicated data types that unify eventual consistency and observable atomic consistency". In: *Journal of Logical and Algebraic Methods in Pro-*

gramming 114 (2020), p. 100561. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2020.100561.

[ZN16] Nosheen Zaza and Nathaniel Nystrom. "Data-centric Consistency Policies: A Programming Model for Distributed Applications with Tunable Consistency". In: *First Workshop on Programming Models and Languages for Distributed Computing, PMLDC@ECOOP 2016, Rome, Italy, July 17, 2016*. ACM, 2016, p. 3. DOI: 10.1145/2957319.2957377.