

# **Banking on Domain Knowledge for Faster Transactions**

**LEVERAGING MODELS TO AVOID COORDINATION**

Tim Soethout



# **Banking on Domain Knowledge for Faster Transactions**

**Leveraging Models to Avoid Coordination**

Tim Soethout



# **Banking on Domain Knowledge for Faster Transactions**

Leveraging Models to Avoid Coordination

## **PROEFSCHRIFT**

ter verkrijging van de graad van doctor aan de  
Technische Universiteit Eindhoven, op gezag van de  
rector magnificus, prof. dr. ir. F.P.T. Baaijens, voor een  
commissie aangewezen door het College voor  
Promoties, in het openbaar te verdedigen  
op maandag 27 juni 2022 om 11:00 uur

door

**Timotheus Martinianus Soethout**

geboren te Nijmegen

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de **promotiecommissie** is als volgt:

voorzitter: prof. dr. J.J. Lukkien  
promotores: prof. dr. J.J. Vinju (CWI – Technische Universiteit Eindhoven)  
prof. dr. T. van der Storm (CWI – Rijksuniversiteit Groningen)  
leden: dr. N. Crooks (UC Berkeley)  
prof. dr. W.J. Fokkink  
prof. dr. G.H.L. Fletcher  
prof. dr. A. Iosup (Vrije Universiteit Amsterdam)  
adviseur: drs. J. de Vos (ING Bank Nederland)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.



Centrum Wiskunde & Informatica



The work in this dissertation has been carried out at Centrum Wiskunde & Informatica (CWI), in collaboration with the ING Bank, under the auspices of the research school Institute for Programming research and Algorithmics (IPA).

A catalogue record is available from the Eindhoven University of Technology Library ISBN: 978-90-386-5523-9



# Table of Contents

<b>Acknowledgements</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background	3
1.1.1 Consistency and Isolation	6
1.1.2 Coordination Avoidance	9
1.2 Coordination in Distributed Systems	9
1.2.1 Coordination Avoidance by Example	12
1.3 Context within ING Bank	13
1.4 Approach	15
1.4.1 Research Questions	15
1.4.2 Local-Coordination Avoidance	17
1.5 Origins of the Chapters	19
<b>2 Path-Sensitive Atomic Commit: Local Coordination Avoidance for Distributed Transactions</b>	<b>23</b>
2.1 Introduction	24
2.2 Background: Distributed Transactions	26
2.3 Path-Sensitive Atomic Commit (PSAC)	28
2.3.1 PSAC in action	30
2.3.2 PSAC Algorithm	32
2.4 Implementation: Rebel and Akka	35
2.4.1 Rebel: a DSL for Financial Products	35
2.4.2 Executing Rebel on Akka	37
2.5 Performance Evaluation	40
2.5.1 Research Objectives	40
2.5.2 Deployment Setup	41
2.5.3 Baseline Experiments: Akka Scalability	43
2.5.4 Synchronization Experiments: PSAC vs 2PL/2PC	44
2.6 Discussion	49
2.6.1 Threats to Validity	49
2.6.2 Limitations	51
2.6.3 Evaluation	53
2.7 Related work	53
2.8 Further Directions	55

## Table of Contents

2.9	Conclusion . . . . .	57
<b>3</b>	<b>Static Local Coordination Avoidance for Distributed Objects</b>	<b>59</b>
3.1	Introduction . . . . .	59
3.2	Independent Events . . . . .	61
3.2.1	Bank Account Example . . . . .	61
3.2.2	Independent Events . . . . .	63
3.2.3	Statically Independent Events . . . . .	63
3.2.4	Computing <i>SIE</i> . . . . .	64
3.2.5	Always Accept or Always Reject? . . . . .	65
3.3	Local Coordination Avoidance (LoCA) . . . . .	66
3.3.1	Static LoCA . . . . .	67
3.4	Evaluation . . . . .	69
3.4.1	Independence in Realistic Scenarios (RQ 1) . . . . .	69
3.4.2	Throughput and Latency (RQ 2) . . . . .	70
3.5	Discussion . . . . .	76
3.6	Related Work . . . . .	77
3.7	Future Work . . . . .	80
3.8	Conclusion . . . . .	81
<b>4</b>	<b>Automated Validation of State-Based Client-Centric Isolation with TLA<sup>+</sup></b>	<b>83</b>
4.1	Introduction . . . . .	84
4.2	Background: State-Based Client-Centric Consistency . . . . .	85
4.3	Formalizing CI in TLA <sup>+</sup> . . . . .	87
4.4	CI examples . . . . .	89
4.5	Model Checking Algorithms Using CI . . . . .	92
4.5.1	Formalizing 2PL/2PC . . . . .	92
4.5.2	Model Checking 2PL/2PC . . . . .	95
4.5.3	2PL/2PC Bug Seeding . . . . .	96
4.6	Discussion and Future Work . . . . .	98
4.7	Conclusion . . . . .	99
<b>5</b>	<b>Safely Exploiting Contract-Based Return-Value Commutativity for Faster Serializable Transactions</b>	<b>101</b>
5.1	Introduction . . . . .	102
5.2	Background: State-Dependent Commutativity and Return-Value Commutativity . . . . .	104

5.3	Contract-Based Commutativity: actionable SDC and RVC . . . .	I06
5.3.1	Computing CBC at Run Time . . . . .	I07
5.3.2	CBC for Multiple In-progress Operations . . . . .	I10
5.4	Return-Value Serializability . . . . .	I11
5.5	Local Coordination Avoidance (LoCA) . . . . .	I13
5.5.1	LoCA with Independent Events . . . . .	I14
5.6	Model Checking LoCA and RV-SER . . . . .	I15
5.7	Initial Validation . . . . .	I18
5.8	Discussion . . . . .	I19
5.8.1	Threats to Validity . . . . .	I21
5.9	Related Work . . . . .	I22
5.10	Conclusion . . . . .	I23
<b>6</b>	<b>Design and Architecture</b>	<b>125</b>
6.1	Introduction . . . . .	I25
6.2	Distributed Actors in rebel-runtime-lib . . . . .	I26
6.3	Experiment Runner . . . . .	I32
6.4	rebel-conflictors: rebel-sie and rebel-cbc . . . . .	I33
6.5	Verifying Isolation in TLA <sup>+</sup> with isolation-specs . . . . .	I34
6.6	Summary . . . . .	I35
<b>7</b>	<b>Conclusion</b>	<b>137</b>
7.1	Research Questions . . . . .	I37
7.1.1	RQ 1: Local Coordination Avoidance with Independent Operations . . . . .	I38
7.1.2	RQ 2: Local Coordination Avoidance at Run Time . . . .	I39
7.1.3	RQ 3: Local Coordination Avoidance at Compile Time .	I40
7.1.4	RQ 4: Performance Benefits in High Contention Scenarios	I40
7.1.5	RQ 5: Isolation Guarantees . . . . .	I41
7.2	Discussion and Further Directions . . . . .	I42
7.2.1	Implications for Research . . . . .	I42
7.2.2	Implications for Practitioners . . . . .	I43
7.2.3	Further Directions . . . . .	I44
	<b>Bibliography</b>	<b>147</b>
<b>A</b>	<b>Path-Sensitive Atomic Commit</b>	<b>161</b>
A.1	Example 2PL/2PC and PSAC diagrams with ABORT . . . . .	I61

## Table of Contents

A.2 Actor class definition . . . . .	161
A.3 Detailed Rebel implementation using Akka . . . . .	164
<b>B Posters</b>	<b>167</b>
<b>Executive Summary</b>	<b>173</b>

# Acknowledgements

A major thanks to all the people involved directly and indirectly. Without you this PhD project would never have happened. All discussions before and during helped me a lot. A PhD is a very personal journey towards technical, but also personal deepening. Both because of the challenges encountered in the research itself, but also due to events happening in the span of 6 years.

First, I would like to thank my supervisors Jurgen Vinju and Tijs van der Storm. They learned me almost everything I know about research. Jurgen was hugely influential by using positive affirmation when research and life were harder. His endless knowledge, positivity and understanding is truly inspirational. Tijs was already a great and inspiring group member at CWI, with whom I shared some great and fruity late nights. Tijs also became my promoter later in the project, which changed the supervision style, but that was definitely not a bad thing. Looking at the subject and research in a different way, made the quality and insight deeper.

This research project would not have been possible without my wife Saskia Ubbink at my side. Thank you for your realistic view, and your support at all times. Your unending patience and endless interest broke a lot of deadlocks during various parts of research, paper writing and thesis writing. Also, the births of our children, Stern and Aster, were monumental in creating real strict deadlines for a paper and the final thesis.

A big thanks to my always supporting parents, Thecla and Luc Soethout. During my upbringing you already guided my interest to computers and programming. Instead of only exposing me to games, you showed me programming in SuperLogo and C and with that, computational thinking. Thanks to my siblings, Stijn, Floor and Gijs, and brother-in-law Martijn, for always being their jovial selves. A special mention is for my grandparents, Joke, Wiel, Mia and Carel, and parents-in-law, Marianne and René, who are, were or would have been very proud.

Major thanks to ING bank for giving me the opportunity to take on this challenge and for generously funding this research collaboration. Thanks to: Jannes Smit for sponsoring and supporting; Joost Bosman for setting up connections with academia, and being ING's patron of research; Jordi de Vos for supervision and support, and for asking the right reflection questions at the right moments; and other ING colleagues for ongoing emotional and substantive

## Acknowledgements

support: Alessandro Vermeulen, Kevin van der Vlist, Robbert van Dalen and Luna Luo. Also thanks to colleagues on the side line who supported and helped me during various parts of the research: Ana, Anton, Bertjan, Daniele, Effi, Joris, Jorryt-Jan, Miguel, Rene, Sebastian, Stefan, Viet, and many more.

Of course the whole SWAT research group was a warm bath. Everyone was really welcoming and kind, also to unexperienced researchers. Thanks to Aiko, Ali, Anastasia, Bert, Davy, Jouke, Jurgen, Kai, Lina, Mauricio, Michael, Nikolaos, Pablo, Paul, Riemer, Rodin, Thomas, Thomas, Tijs, Ulyana, Yanja for the great coffee breaks, fruit breaks, and (Duvel) drinks at Praethuys and/or Polder. A special mention goes to Jouke, for your interesting work to build upon and for your friendship and great commute car rides. Thanks Riemer, my office mate for the whole period. We had many good and motivational discussions on everything and then some, including the special L222 office humor.

Marijn, one of the *paranimfs*, is a great friend and also unexpectedly motivated to understand the research, even though his expertise lies in a different area. Thanks to Jochem for all discussions on the squash court including my research, academia in general, life, programming and everything else. A special thanks goes to the TanCKI, who made sure that enough social and explicitly non-PhD events and relaxation took place.

I thank all the anonymous reviewers for their thorough reviews and feedback, and the defense committee for evaluating and accepting my thesis.

Live long and prosper. 🙌



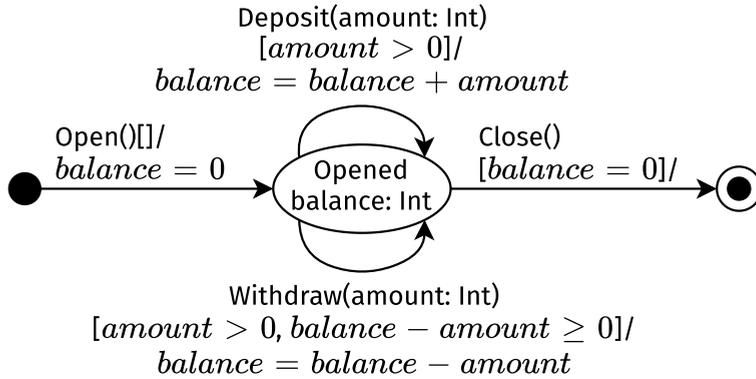
# Introduction

In large-scale enterprise software systems, performance and data consistency are paramount. Internal and external clients should receive timely and correct responses from web applications, also when these applications are under high load. For example, customer information needs to be up-to-date and correct, and bank accounts' money should not get lost. This all needs to happen performantly under high transaction loads. Complex IT landscapes with multiple distributed applications in heterogeneous technologies are complicated and hard to maintain over time. All these applications communicate and synchronize with one another and many store data. For these applications maintaining high throughput and enabling scalability is very important.

An important bottleneck for scalability is coordination. Different applications or parts of applications, potentially running distributed in a data center or geo-replicated, synchronize over updates on data. For example, transferring money from a bank account to another bank account should only happen when the money is respectively withdrawn and deposited on both. These kinds of atomic updates are expensive to implement and can potentially slow all connected applications down. Connectivity between applications is more expensive in terms of delay when distances between hosting locations increase, and even more when multiple back-and-forths are necessary. In the worst case new operation requests have to wait on all coordination to finish before starting.

Coordination needs to be avoided as much as possible. However, one-size-fits-all approaches, such as databases or generic middleware, stay on the safe side by possibly doing more coordination than necessary. Determining when coordination can be avoided is non-trivial and often dependent on the specific application logic. This dissertation therefore proposes approaches to leverage

## Chapter 1 Introduction



**Figure 1.1** A simple bank account state chart with contracts on operations. Operations follow state chart notation: Event(fields)[guard]/effect

(domain) knowledge of applications to reduce coordination. Contracts on objects and operations enable detection of necessity of coordination.

This doctoral project is born from an ongoing research collaboration between the Dutch national research institute for mathematics and computer science CWI and ING Bank on enterprise software engineering. Its main goal is to devise approaches to manage the complexity and evolution of large enterprise IT systems. This is explored in the context of a state-machine based domain specific language for financial products, Rebel [108]. It generalizes to other models, as long as a similar contract is available.

**Examples** A running example throughout the dissertation is bank accounts. These accounts do not allow overdraft, so their balances should never go below zero. Operations defined on bank accounts are the deposit and withdrawal of money. This is always done in the context of a transfer from one account to another. Some chapters consider a slightly more involved version where accounts first have to be opened, and can be closed.

Figure 1.1 shows a state chart of a bank account that can be opened, closed and have money withdrawn and deposited via operations. These operations are enabled in specific states when guards or preconditions on its local state (the account balance) hold. The effect of the operations is always local and updates the internal state.

A synchronized operation, such as a money transfer, requires all involved objects (accounts) to have the grouped operations enabled. For a money transfer the withdraw and deposit operations act on respectively the *from* account and

the *receiving* account. Only if both accounts enable the operations, the transfer happens. This is an atomic step for multiple objects. An account only accesses its local state and operations, so it knows that the local deposit is valid, but not yet if the withdraw on the contra-account is. For this communication and coordination is necessary. Also locally for the account, a local deposit can be needed for the next local withdrawal to be allowed or not.

The definitions of the objects and its operations, guards and effects is called the *contract*. One example of leveraging these contracts to reduce coordination is detecting that multiple deposits for an account can safely run concurrently without coordination, because their guards are always true. Even if the first deposit is aborted due to the contra-account not having enough balance, the second deposit can happen, since the abortion has no influence on its enabledness. The deposit's guard-contract allows multiple deposits in parallel, because independent of the other in-progress deposit the operations can make progress, without leading to a different output.

## 1.1 Background

**Distributed Systems** A large interconnected IT landscape is a distributed system. A distributed system is a collection of servers or applications that communicate via messages [33]. For resiliency purposes the functioning of the system should not be impacted by for example failing hardware. A distributed program is deployed on the different servers as separate components. The main benefit is that the program can continue to run if separate components fail and that its application can scale beyond the single machine. The main issue is that there is no longer a main process or main clock, since all components operate independently. Also, messaging over the network results in overhead in communication compared to within a single server.

This dissertation relates to the distributed systems and database field by considering distributed objects and messages between them. It embraces the latency, failure, and potential distributed location of objects. It relates closely to database transactions [9, 42] and distributed data structures [85].

**Reactive Programming** The reactive manifesto [16] describes principles that lead to resilient and responsive applications, based on asynchronous message passing. Instead of trying to abstract away the properties of implementation

## Chapter 1 Introduction

layers, such as message delivery failure, delays, and latency, it becomes part of the programming model. Abstractions of synchronous methods and non-parallelism, while implemented on top of non-perfect networks and distributed objects, can lead to unexpected and complex corner cases. Embracing failure of the underlying implementation and hardware is done by including this in the programming abstractions.

Messages are also the base of our approach: Distributed objects or actors that communicate via messaging. The modeling of the objects and architecture of the implementation approach is done in a domain driven design (DDD) [17, 26, 31] fashion. The essential complexity of message passing and making sure the correct messages are handled, lead to explicitly stating requirements and consistency for applications.

Distributed Objects [110], such as actors [53], that communicate with each other need ways to synchronize operations, in order to fulfill business requirements, e.g. money transferred from one bank account to another should not get lost. Abstractions such as guaranteed delivery are created on top of message passing. Virtual Actors are created to make sure they are restarted without data loss on another server node when crashed or unavailable. Part of this is also the requirement of synchronization, certain operations are only allowed when other operations also happen or only if and only if conditions on other distributed objects hold. This transactionality inherently requires coordination, but sometimes coordination can be deferred, for example if operations do not change the internal state, or can be reordered without this influencing decisions on other objects.

Keeping decisions local [64] reduces potential bottlenecks and other challenges with cloud computing. Reaction speed of interaction is fast, ownership of data stays with the user, applications keep functioning when offline, all while maintaining most functionality that cloud services provide. A change from the cloud and centralized computing model, which is currently the major part of the online landscape, required a change of view. The contributions in this dissertation help towards such a local-first approach. It supplies opportunity and direction on where decisions can be kept local. Objects which do not require coordination, can be implemented in a local-first fashion.

**Model Driven Engineering** Model driven engineering (MDE) [24, 91] starts with models of some kind. Models describe for example business objects and rules. Often Domain Specific Languages (DSLs) [74] or visual representations are used. Model driven engineering raises the level of abstraction by using

the domain-specific solution space. DSLs express domain specific models and constraints. This allows for specialized tooling with targeted semantical checks, such as type and constraints checkers. Models can be automatically translated to other formalisms, such as executable implementations. Both MDE and DSLs abstract away from low-level implementation details and allow reasoning on the domain level.

The holy grail for MDE is fully generating the implementation from these often-at-higher-level models. MDE is an important part of the *low-code* and *no-code* movement [90]. These low-code development platforms provide an often visual development environment in order to quickly produce applications based on models, where the platform takes care of implementation details. Many vendors promise applications using only models without requiring specialized personnel. The resulting application should be correct, fast and scalable with customer demands. The algorithms and model analysis methods presented in this dissertation are compatible with and fit into these kinds of platforms.

However, generating such an implementation is far from trivial, especially when non-functional requirements are taken into account, e.g. performance (throughput, latency), security, audit trails, etc. The main advantages are that models are often at a higher level and allow reasoning without being bothered by implementation details, and the code generator can be reused [14]. This means the far-from-trivial design is reusable. This also means that, since business requirements often change less often than new implementation strategies and hypes appear, these models are reusable in different iterations for the applications, e.g. different used frameworks, programming paradigms and changes in external technical connections. Different implementation targets can be generated and large parts of an application can be generated using the source models. This also leads to ample opportunity to optimize: specialized implementations can be used that provably implement the model, but do not resort to ad-hoc optimizations.

Many generative programming approaches use general purpose building blocks to implement the models. Since these platforms need to work in all situations, these general purpose building blocks, such as databases, stay on the safe side on the kinds of interaction they allow. For example, relational databases lock rows and columns based on low-level reads and writes to rule out theoretical errors that might never occur for the specific application or model. The general purpose code has stricter constraints than the actual model might require for functionally correct behavior.

## Chapter 1 Introduction

This work focuses on constructively and deliberately leveraging the semantically higher-level knowledge from models to increase performance in specific cases without violating business and data consistency requirements. This is a form of domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) [74]. The actual application logic can be more lenient than the generic building blocks based on lower-level operations. By using the higher-level semantic information available, these cases can be found out and used.

**System model** The system model is state charts with operations [21, 47]. On top of this multiple state charts take part in (distributed) transactions.

Grouped operations are transactions that by default run under serializable isolation [42]. This means that externally all operations run in a serial order. Depending on the level of abstraction, operations can be reads and writes of database fields, or higher-level operations on state machines. A grouped transaction for a money transfer between bank accounts  $A$  and  $B$ , can for example exist of two operations: a withdrawal on  $A$  and a deposit on  $B$ ; or when looking at a lower level, a read and write the balance of  $A$  and a read and write of the balance of  $B$ . A schedule is a specific ordering of operations. An isolation level can be determined for a schedule, meaning that the observed execution as defined by the schedule could have occurred under that isolation level [23, 115]. Adya's generalized isolation levels [1] define a direct serialization graph based on conflicts between different transactions and determines via cycles in this graph if an isolation level is upheld. For example, strict serializability is the highest achievable isolation level, and relaxing conflict relations result in lower, less stricter isolation levels, such as, in order of strict to looser guarantees [9]: snapshot isolation, repeatable read, read committed and read uncommitted. Crooks *et al.*'s client-centric isolation model [23] looks at observed return values to define the different isolation levels, and proofs them to be equivalent to Adya's model.

### 1.1.1 Consistency and Isolation

The terms *consistency* and *isolation* are often used vaguely and interchangeably [49]. In different research communities they have different meanings. It is beneficial to define what these terms mean in the context of this dissertation.

**Consistency** Consistency has many meanings in different fields. It often is a mix of semantic program consistency and guarantees on groups and orders of operations. In this dissertation consistency is used for adherence to the local contract of an objects. Consistent objects only reach states in accordance to their guards and local effects.

This is similar to the consistency *c* of ACID transactions in databases [46]: The client defines consistency by only committing the transaction when correct from a program semantics perspective. Next to that, the database preserves all the database rules, such as unique and foreign keys, and cascading updates.<sup>1</sup> The client or application controls what fields and values are relevant for the application semantics, and should be included in the transaction.

**Isolation** Next to that, isolation (the *I* of ACID) is concerned with “consistency” of operations on a group of objects or resources, where ACID consistency is only concerned with single resources. It concerns the isolation of a database transaction: all operations in a transaction should not interfere with operations from other transactions.

The most well-known and easy-to-reason-about isolation level is *serializability*. All transactions must be totally isolated from another and appear in a serial order. Operations from different transactions should not read not yet committed or aborted values. The flexibility of interpretation lies in orderings that are equivalent to a serial order. But when are orderings equivalent? When looking at low-level read and written values, equivalence is based on observing the same reads and writes. If higher-level program semantics are taken in to account, the equivalence could be less strict, e.g. as long as deposits and withdrawals on a bank account do not have to be retracted they might be considered equivalent to the serial order as well [23, 115]. This flexibility is explored in chapter 5.

**Convergence** Convergence is a more concrete and better defined name for *eventual consistency*, sometimes called *eventual convergence* [49]. Convergence covers single objects, where replicas of the object eventually converge to the same state when the same operations are received in any order. This is a local

---

<sup>1</sup> This is different from *c* from the CAP theorem [37]: Consistency is *linearizability* of a single register: the entire application operates as if it is a single value. Operations happen somewhere between invocation and response from a client’s perspective and are not limited to reads and writes.

## Chapter 1 Introduction

guarantee on the component state, based on eventual state. A merge function is associative, commutative and idempotent, and handles the convergence. “A system is convergent or ‘eventually consistent’ if, when all messages have been delivered, all replicas agree on the set of stored values.” [5], which is the same as Strong Convergence from CRDTs [93].

**Confluence** Confluence [5, 50] is about input and output of operations. Its scope is a set of components, so multiple objects. It guarantees the same set of outputs for all orderings of its inputs. Confluence gives no promises of recency and it not about memory reads and writes.

In practice this means that a confluent component can receive any number of operations, and return values (outputs) directly. It never has to retract those outputs. When dealing with operations that are not confluent, the earlier confluent stream of operations can be *sealed*, for example by coordination. Non-confluent blocking operations then run to completion, before opening a new stream of confluent operations.

An interesting observation is that retraction of outputs depends on what an observer allows, depending on the use case. For example, outputting a different updated balance on a successful withdrawal from different replica’s could be equivalent from a application perspective, e.g. as long as the success or failure of said withdraw does not change.

**Confluence and Serializable Isolation** Both confluence and serializability are properties over multiple objects. Serializability is a guarantee on (equivalence to) the serial order of operations in transactions. Confluence is about the non-retraction of outputs. Confluence is a property on higher application-level operations, where serializability often looks at lower-level reads and writes. This dissertation bridges the gap between these formalisms. The goal is to reduce the required coordination, by providing serializability, while leveraging confluence where possible. When safe we give outputs earlier, by detecting that the output does not have to be retracted in the future. One of the main insights is that we look at higher-level operations, also for serializability, because the order of operations can be less strict, while maintaining application invariants and serializability on that level. Similar to sealing before non-confluent operations, all relevant in progress operations need to finish before a conflicting operation is handled.

The different chapters focus on different aspects of this journey. First, chapters 2 and 3 focus on an formalism and algorithm to enable concurrent oper-

ations in distributed objects. The focus there is to automatically detect, both online and offline, when coordination can be avoided. The point of departure is only local (object) information, since local decisions require no coordination, enable independent progress, and with that increased performance. These chapters also look at computational cost, and in which scenarios it actually improves performance. In the later chapters 4 and 5, the focus shifts to determining the adherence to serializability. These chapters provide a framework for determining isolation guarantees for higher-level operations and an alternative offline and online property that is sufficient for serializability.

### 1.1.2 Coordination Avoidance

The term *Coordination Avoidance* was coined by Bailis [8, 10]: use coordination only when necessary. Coordination avoidance focuses on lock-free algorithms in a geo-replicated setting. Large parts of real world use cases can be implemented without locks, increasing throughput. When transactions do not conflict, they run on multiple geo-located data centers without coordination. They are eventually merged in an asynchronous fashion. Bailis states: “Invariant Confluence captures a simple, informal rule: coordination can only be avoided if all local commit decisions are globally valid.” This dissertation focuses on leveraging domain semantics or models in order to avoid coordination, by determining when coordination is not needed based on the outcome of operations.

## 1.2 Coordination in Distributed Systems

In distributed systems coordination is an expensive endeavor. Especially latency<sup>2</sup> increases a lot when more coordination is required. Coordination is necessary because of data consistency requirements. One way to increase performance and reduce latency is to reduce the data consistency requirements. For example, if you do not require all group chat messages to have the same order for all participants, a lot less coordination is necessary. In the financial industry these consistency requirements are often harder to relax.

---

<sup>2</sup> Latency [34, 62] is the overhead time span between a request and a response in which no actual work is done. This is “waiting” on the result. Latency is typically larger when remote calls are involved.



**Table 1.1** An intuitive representation on how time for a computer is relative to a more human scale. Reproduced from *Systems Performance: Enterprise and the Cloud* [43].

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 second
Level 1 cache access	0.9 ns	3 seconds
Level 2 cache access	2.8 ns	9 seconds
Level 3 cache access	12.9 ns	43 seconds
Main memory access (DRAM, from CPU)	120 ns	6 minutes
Solid-state disk I/O (flash memory)	50–150 $\mu$ s	2–6 days
Rotational disk I/O	1–0 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware virtualization system reboot	40 s	4 millennia

necessary is of paramount importance. And ideally without (too much) loss of data or program consistency guarantees.

Application landscapes, such as large enterprise systems, grow over time. Their usage increases and dependency grow as well. For many operations coordination is a necessary evil, e.g. in a core banking application, you can not lose money, so every withdraw on an account should be accompanied by a deposit on another bank account. If these accounts are stored on other servers, or even on other continents, coordination is necessary, different operations of these accounts need to wait on previous ones to finish.

This dissertation focuses on reducing coordination required while maintaining the consistency and isolation guarantees. It leverages program semantics in the form of contracts for this. These contracts are defined in models, domain specific languages, or annotated in or derived from source code.

### 1.2.1 Coordination Avoidance by Example

How can these contracts be used to avoid coordination? Consider a tax office bank account, which is used in collecting taxes and paying out benefits to citizens. By regulations, both of these use cases should be run in a limited time frame. This means that this account is involved in many transactions for a short moment. This tax account is modeled as the example in figure 1.1, with a balance, and withdrawal and deposit operations.

A straightforward implementation of this example runs into problems with a tax bank account being involved in all individual transfers, creating a potential bottleneck for the legislative time frame. A solution can be approached in multiple ways. Either by functionally changing the problem to solve the inherent technical shortcomings or keeping the original modeling and solving the problem at run time. An example of a functional change is splitting the tax account in multiple separate accounts and use those to do the bookings, or by first booking the total of payed benefits from the main tax account to an in-between account (wash account), and then do the the transfers without overdraft checks on the intermediate accounts. Both proposed alternate modeling approaches reduce coordination, either by splitting coordination over multiple accounts, or taking the coordination off the critical path. An example of a run-time solution also depends on the specific modeling. In this case the tax bank account can actually run the different withdrawals for benefit payouts in parallel as long as enough balance is available for all of them. In this case extra domain knowledge about the problem reduces the required coordination. Both alternatives for this example depend on the same inherent property of the model, that enough total balance is available for all transfers: either by checking this at run time for multiple individual withdraws, or when changing the functionality by first moving the sum of withdraws to an intermediate account.

These examples show that multiple solutions are possible. The changing of the model of tax bank accounts to into multiple separate accounts, shows that many inherent technical bottlenecks are often solved and encoded in the business logic itself. On the hand, this is good practice, since performance trade-offs are inherent to the modeling strategy and should be solved explicitly on a business level. On the other hand, this lets technical implementation limitations slip into the business domain. Firstly, this dissertation suggests a feasible middle way where seemingly technical limitations are short-circuited by finding operations which can run without coordination, and with that reducing bottlenecks. Secondly, it makes clear where performance bottlenecks

could show up and points out when a business decision is necessary to make the inherent trade-offs between performance, functional modeling, and risk.

### 1.3 Context within ING Bank

This research was executed in the context of ING Bank.<sup>3</sup> ING Bank is a large multi-national bank, with head offices based in the Netherlands. It has a very complex IT landscape that grew and grows over time, consisting of micro services, mainframes and everything in between. The interconnectivity between these applications is complex. Strict data consistency requirements ensures maintained trust from customers and adherence to laws and regulations.

Some key figures on ING Bank:<sup>4</sup>

- 39.3 million customers worldwide (year-end 2020)
- 58 000+ employees worldwide
- € 2.4+ billion net profit (2020)

One of the challenges for ING is to keep track of evolution of applications and business logic over time. An approach is to employ models or domain specific languages for this purpose on different abstraction levels. The models capture the functional business logic and are not cluttered by implementation details. This allows for underlying implementations to change, or even be regenerated, while the higher level business logic stays the same.

**Modeling choices** As observed in the previous section many high-performing implementations depend on how the business logic is defined. For example when withdrawing money from an ATM, the ATM should be connected to the bank's balance servers in order to check if enough balance is available. In this case the business logic dictates that the ATM checks the balance. However, when a network disconnect occurs, multiple solutions are possible, for example not handing out any money at the disconnected ATM until connection is restored and balance can be checked, or for example still handing out smaller amounts and deducting them from the balance on reconnect. In this last case there is a business risk involved where a client withdraws money that is not available on his account. The risk depends on how often this happens, what the actual

<sup>3</sup> <https://ing.com>

<sup>4</sup> From the annual report 2020 found at <https://www.ing.com/Investor-relations/Financial-performance/Annual-reports.htm>.

## Chapter 1 Introduction

damages are and is reduced by only allowing small money amounts. On the other hand this leads to many customers still being able to withdraw money without being hindered by the disconnect and increases customer satisfaction. This example illustrates that the way the business logic is modeled has great impact on the availability and performance of an implementation. It shows coordination avoidance on the model level.

**The Bank in 30k Lines of Code** In this collaboration between CWI and ING on enterprise software engineering the main goal is to devise approaches to manage complexity and evolution of large enterprise IT systems. The Rebel domain specific language [105, 106, 107, 108] is created for this purpose. In Rebel a domain expert specifies financial products as communicating state machines, operations on the state machines are defined using pre- and post-conditions. The Rebel toolset includes a symbolic interpreter, visualizations, a testing language and prototyped connectivity to real banking systems.

The initial dissertation's research project started as an implementation approach for Rebel: a code generator implementing the Rebel specifications on top of actors and distributed databases towards an enterprise-grade correct, resilient and available implementation. Performance bottlenecks in certain kinds of specifications led to the investigation of leveraging semantics from the specifications to remove these bottlenecks.

The Rebel language and implementation generator spun off into various projects inside ING Bank: Multiple MSc graduation projects [28, 66, 67, 111, 112, 113, 116, 117], the current PhD project and a multi-million productization inside ING's innovation incubator.

**Industrial Relevance** Separating functional domain knowledge and implementation is extremely beneficial for large enterprises, where implementations change more often than core business processes. It enables individual evolution of the business and the underlying implementation. This dissertation is a step in the direction of carefree focus on the domain of the business, while letting a code generator or implementation figure out how to run it performantly. For inherent trade-offs and fundamental bottlenecks, tooling can be created to point out these bottlenecks out and suggest a different way of modeling the application, such as the risk trade-off in the ATM example, and make it an explicit business decision instead of an implementation artifact.

Algorithms, such as LOCA with different conflict relations, e.g. independent events (chapters 2 and 3) and contract-based commutativity (chapter 5), allow

this decoupling of specification and implementation. The implementation takes care of performance where possible and the specification can be analysed on potential bottlenecks. Together with usage patterns, these bottlenecks become clear to the business experts and appropriate changes in the models can be made.

### 1.4 Approach

Reducing the amount of coordination is important, because it leads to bottlenecks in performance, especially when messages delivery takes longer. Separating functional business rules and implementation details can help here. Ideally, optimized implementations automatically leverage the program semantics.

When looking at applications as communicating distributed objects, how can coordination between operations on an object basis be avoided? This leads to the following research questions:

#### 1.4.1 Research Questions

**Research Question 1 (RQ 1):**

When can coordination between two operations be avoided without violating isolation and consistency requirements?

In order to avoid coordination, we require constraints on when it is safe to run operations concurrently. This question captures what a sufficient condition is for operations to interleave arbitrarily, without violating isolation and consistency requirements.

**Research Question 2 (RQ 2):**

How can coordination avoidance between two or more operations be achieved at run time?

How can answers to RQ 1 be leveraged in an implementation at run time? This considers formal and descriptive formalisms and how to compute them at run time.

### **Research Question 3 (RQ 3):**

How can coordination avoidance between two operations be determined at compile time?

Given that coordination reduction requires more (local) computational overhead, can this overhead partially be reduced by compile-time optimizations? This considers formal and descriptive formalisms, and how to make them actionable at run time.

### **Research Question 4 (RQ 4):**

What are the performance benefits for local coordination avoidance and in which scenarios do they hold?

Given the implementation approaches from RQs 2 and 3, what performance benefits do they entail? And in which use cases can one expect these benefits? This specifically considers scalability, throughput and latency.

### **Research Question 5 (RQ 5):**

What are the isolation guarantees when using the different non-conflict relations, and how do they relate?

Isolation guarantees are the well-known levels of database isolation for (distributed) transactions. What isolation guarantees do the different approaches for avoiding coordination actually provide? How can this be determined? And is this domain specific?

The research questions are explored around three axes:

**Compile Time versus Run Time** Effectively, coordination is only reduced at run time, because that is where the actual coordination occurs. However, since our approach of reducing of coordination requires extra computation, parts of this can be determined statically at compile time. Chapter 2 describes a run time variant, which leverages run-time state (RQs 1 and 2). Chapter 3 describes a variant with reduced coordination without requiring run-time state (RQs 1 and 3). The static variant should hold for all possible run-time states.

**Formalization versus Implementation** Both the run-time and compile-time variants are described on two levels: a descriptive formalization, and an implementation making the formalization operational.

**Table 1.2** Mapping of Chapters to Research Questions

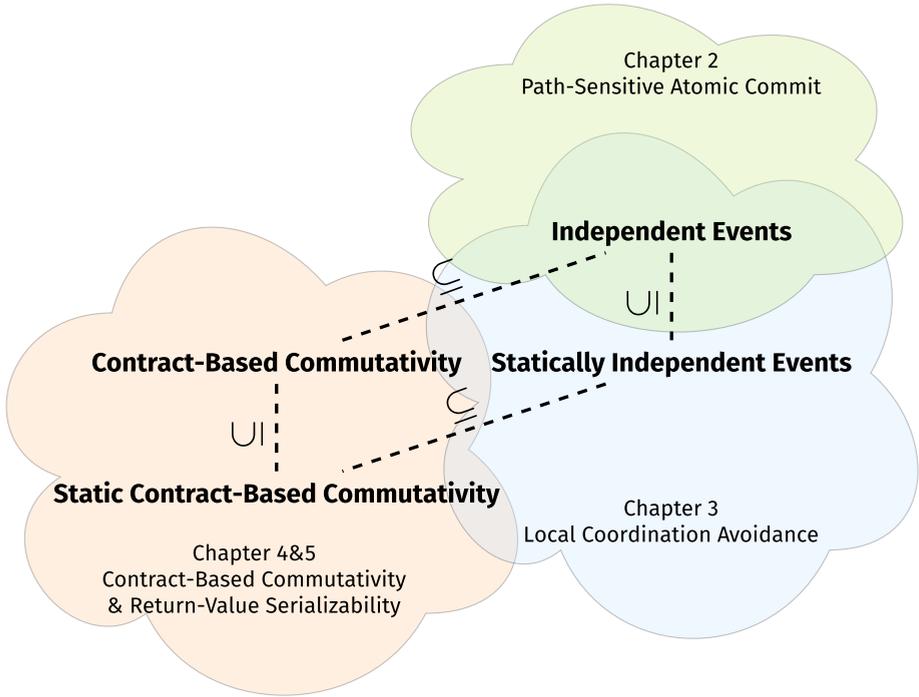
Chapter	Research Questions	run/compile time	descriptive/operational	local/global guarantees
2	RQS 1, 2 and 4	run time	operational	local
3	RQS 1 and 3	compile time	both	local
4 & 5	RQS 1 and 5	both	both	global

**Local Consistency versus Global Isolation Guarantees** The starting requirement is a correct implementation with respect to the contract. However, there are multiple ways to interpret correctness. A local perspective is that the object implements the contract, which means that local state should never be invalid, or the object should never reach an unreachable state. This is called *consistency*. Even when the individual objects never show an invalid state, and thus are *consistent*, different observable effects orders are possible. These are known as *isolation* guarantees. PSAC with independent events (chapters 2 and 3) focuses on local correctness guarantees, where LOCA with contract-based commutativity (chapter 5) also focuses on global isolation properties such as serializability.

**Mapping of Chapters to Research Questions** Table 1.2 shows an overview of the relation of chapters to the research questions and dichotomy axes.

### 1.4.2 Local-Coordination Avoidance

The Local-Coordination Avoidance (LOCA) algorithm is a connecting thread of this dissertation. A distributed object or actor, which receives method calls or messages, can run this algorithm in order to do coordinated operations. It builds on top of an atomic commit algorithm, in this case two-phase commit (2PC), but generalizes to different atomic commitment or consensus protocols. LOCA's concurrency control differs from two-phase locking (2PL) in the sense that the object only locks for conflicting operations. In between receiving a command to do an operation, and its actual execution, LOCA decides if another incoming operation can already start processing, or should wait for in-progress actions to finish. LOCA requires a conflict relation for this, which takes the current internal state of the object and two operations. Depending on the conflict relation used, its behavior is different. For example, LOCA with Contract-Based Commutativity



**Figure 1.3** Non-conflict relations with related chapters.  $\subseteq$  on the dashed edges shows the subset relation. Each of the non-conflict relations fits in this grander scheme.

(CBC) results in serializable behavior, whereas with Independent Events (*IE*) only provides local linearizability guarantees.

**Conflict relations** These conflict relations follow the interface:  $State \times Operation \times Operation \rightarrow Boolean$ . Given an object state and two operations, it defines if the operations are non-conflicting or not.

Figure 1.3 shows different conflict relations discussed in this dissertation and their internal relations. Independent Events (*IE*) defines non-conflicting operations as operations for which the enabledness of the operation are not changed if the first in-progress operation is eventually applied or not.<sup>5</sup> Statically

<sup>5</sup> Note that *event* and *operation* are used interchangeably to denote the operations defined on an object.

Independent Events (*SIE*) is the subset of *IE* which is independent of the run-time state. So for all possible run-time states the mentioned operations are Independent Events. This is a restriction, but be computed offline, without requiring the run-time state. Contract-Based Commutativity (CBC) and its static counterpart (scBC) are based on observable return values. An operation is CBC with an in-progress operations if and only if their return values do not change when swapped.

## 1.5 Origins of the Chapters

All work presented in this dissertation is related. Improvements and abstractions created later in time support and extend earlier work. Later work places earlier work into a larger frame. For example PSAC from chapter 2 fits in the independent events definition *IE* in chapter 3 and contract-based commutativity (chapter 5) is a drop-in replacement for *IE* and *SIE* (chapter 3) for the runtime LOCA algorithm. The formalization of serializability with low-level reads and writes in chapter 4 are the building blocks and frame for the formalization of return-value serializability (RV-SER) with higher-level operations in chapter 5. This results in serializable isolation guarantees.

This section lists all peer-reviewed contributions. I am the primary author of all of them. Most papers are accompanied by artifacts: (reproducible) evaluation results and/or scripts (e) and source code (s).

**Chapter 2** Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Path-Sensitive Atomic Commit - Local Coordination Avoidance for Distributed Transactions”. In: *The Art, Science, and Engineering of Programming* 5.1 (2021), page 3. DOI: 10.22152/programming-journal.org/2021/5/3

Artifacts (e) (s): Tim Soethout. *Path-Sensitive Atomic Commit: Local Coordination Avoidance for Distributed Transactions Evaluation Data*. Oct. 2019. DOI: 10.5281/zenodo.3405371

**Chapter 3** Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Static local coordination avoidance for distributed objects”. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2019*. ACM Press, 2019, pages 21–30. ISBN: 9781450369824. DOI: 10.1145/3358499.3361222

## Chapter 1 Introduction

Artifacts   : Tim Soethout. *Static Local Coordination Avoidance for Distributed Objects Artifacts*. Sept. 2019. DOI: 10.5281/zenodo.3405232

**Chapter 4** Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Automated Validation of State-Based Client-Centric Isolation with TLA<sup>+</sup>”. In: *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops - ASYDE, CIFMA, and CoSim-CPS, Amsterdam, The Netherlands, September 14-15, 2020, Revised Selected Papers*. Edited by Loek Cleophas and Mieke Massink. Volume 12524. Lecture Notes in Computer Science. Springer, 2020, pages 43–57. DOI: 10.1007/978-3-030-67220-1\_4

Artifacts   : Tim Soethout. *TimSoethout/TLA-CI: TLA<sup>+</sup> Specifications Used in “Automated Validation of State-Based Client-Centric Isolation with TLA<sup>+</sup>”*. Zenodo. July 2020. DOI: 10.5281/zenodo.3961617. URL: <https://github.com/TimSoethout/tla-ci>

**Chapter 5** Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Contract-Based Return-Value Commutativity: Safely exploiting contract-based commutativity for faster serializable transactions”. In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2021*. ACM Press, 2021. DOI: 10.1145/3486601.3486707

Artifacts   : Tim Soethout. *TimSoethout/cbc-artifacts: Artifacts for AGERE’21 paper “Contract-Based Return-Value Commutativity: Safely exploiting contract-based commutativity for faster serializable transactions”*. Zenodo. Sept. 2021. DOI: 10.5281/zenodo.5497756. URL: <https://github.com/cwi-swat/cbc-artifacts>

**SPLASH Doctoral Symposium** The earlier overall dissertation design and approach was sketched and published in the SPLASH doctoral symposium as: Tim Soethout. “Exploiting models for scalable and high throughput distributed software”. In: *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2019, Athens, Greece, October 20-25, 2019*. Edited by Yannis Smaragdakis. ACM, 2019, pages 35–37. DOI: 10.1145/3359061.3361073

**Design and Architecture of the implementations and tools** Chapter 6 features the design and architecture of the software components created to do this research. `LOCA` is implemented in the Akka actor framework, with large-scale deployment in mind. It is fully resilient, horizontally scalable, and by-default

leverages enterprise-grade database Cassandra. It also features a reproducible experiment runner for scalability experiments on cloud infrastructure, such as Amazon Web Services, which is leveraged in the experimental evaluation in chapter 2.

Next to that, the design and tool for the detection of static *IE* (chapter 3) and static CBC (chapter 5) operation pairs is presented. This chapter also discussed the  $\text{TLA}^+$  formalizations, structure and examples as used in chapters 4 and 5.



# 2

## Path-Sensitive Atomic Commit: Local Coordination Avoidance for Distributed Transactions

**Abstract** *Context* Concurrent objects with asynchronous messaging are an increasingly popular way to structure highly available, high performance, large-scale software systems. To ensure data-consistency and support synchronization between objects such systems often use distributed transactions with Two-Phase Locking (2PL) for concurrency control and Two-Phase commit (2PC) as atomic commitment protocol.

*Inquiry* In highly available, high-throughput systems, such as large banking infrastructure, however, 2PL becomes a bottleneck when objects are highly contended, when an object is queuing a lot of messages because of locking.

*Approach* In this chapter we introduce Path-Sensitive Atomic Commit (PSAC) to address this situation. We start from message handlers (or methods), which are decorated with pre- and post-conditions, describing their guards and effect.

*Knowledge* This allows the PSAC lock mechanism to check whether the effects of two incoming messages at the same time are independent, and to avoid locking if this is the case. As a result, more messages are directly accepted or rejected, and higher overall throughput is obtained.

*Grounding* We have implemented PSAC for a state machine-based DSL called Rebel, on top of a runtime based on the Akka actor framework. Our performance evaluation shows that PSAC exhibits the same scalability and latency characteristics as standard 2PL/2PC, and obtains up to 1.8 times median higher throughput in congested scenarios.

*Importance* We believe PSAC is a step towards enabling organizations to build scalable distributed applications, even if their consistency requirements are not embarrassingly parallel.

---

This chapter is previously published as: Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Path-Sensitive Atomic Commit - Local Coordination Avoidance for Distributed Transactions”. In: *The Art, Science, and Engineering of Programming* 5.1 (2021), page 3. DOI: [10.22152/programming-journal.org/2021/5/3](https://doi.org/10.22152/programming-journal.org/2021/5/3)

### 2.1 Introduction

Structuring a software system as a collection of actively communicating objects is an increasingly popular architecture for large-scale, high performance, and high availability IT infrastructure. A common challenge in systems is to maintain high availability and consistency when communicating objects need to synchronize. This is particularly challenging in the context of large, scalable, highly available enterprise software. Our experience in the context of ING Bank<sup>1</sup> is that financial institutions deal with large and complex IT landscapes, consisting of many communicating software applications and components under high request loads, which need to synchronize to keep data consistent. These systems often perform operations that span multiple different applications and server nodes with consistency and durability guarantees.

A safe and well-known distributed transaction protocol to implement these distributed transactions, is Two-Phase Locking (2PL) [84] for isolation with Two-Phase Commit (2PC) [40] for atomicity. While this approach ensures consistency and serializability, it limits throughput in high-contention objects [10, 48, 62], since transactions have to wait on locks of other transactions on the same object. High-contention objects limit the throughput and latency of other objects they communicates with. Depending on the use case, this can be a problem. Examples with high-contention and strong consistency requirements are:

- tax bank accounts, involved in many money transfers and strict regulations on turnaround time;
- a video view counter on a popular video used for advertisement income calculations;
- cases with long-running transactions, where objects stay locked for long periods.

More general, applications with a long tail usage pattern, combined with strict performance and consistency requirements, will have high-contention objects.

This chapter studies the performance of high-load strict 2PL/2PC in high- and low-contention use cases and introduces a novel concurrency mechanism named Path-Sensitive Atomic Commit (PSAC), which minimizes waiting in busy entities by exploiting high-level, functional knowledge about object behavior to reduce contention.

---

<sup>1</sup> <https://www.ing.com>

PSAC trades computing power for reduced waiting on locks, in order to achieve higher throughput than strict 2PL. By detecting whether two or more incoming requests have independent effects PSAC can start processing more requests in parallel than 2PL.

A request is independent of an already in-progress request if the acceptance or rejection of it is not influenced by whether the in-progress requests commit or abort. More details are discussed in section 2.3.

PSAC works under the assumptions that:

- all objects are state machine-based objects with clearly defined actions;
- the behavior of actions is defined by pre- and post-conditions on the local object, using first order logic with support for integer constraints, respectively describing their applicability and state effect;
- an object handles an action as an atomic step, checking the preconditions and applying its post-conditions;
- objects communicate by synchronized actions, which describe an atomic step of a group of actions on multiple objects;
- a group of actions is effectively a transaction among multiple objects.

Separating this functional specification of business objects from their implementation allows experimenting with different back-ends. In this case we have developed a code generator mapping high-level specifications, written in a state machine based domain specific language for financial products called Rebel [105, 106, 108], to an implementation based on the Akka actor framework, employing either 2PL/2PC or PSAC. The PSAC back-end then exploits the model's action pre- and post-conditions to detect independence of actions at run time.

Based on these two implementations we evaluate the performance of PSAC, and compare its performance in the same scenario to the standard of distributed transactions (ACID [46]), which is 2PL/2PC. Our results show that PSAC consistently outperforms 2PL/2PC in high-contention scenarios. Furthermore, PSAC retains the same scalability characteristics as 2PC, but does not guarantee serializability.

The contributions of this chapter are as follows:

- We introduce PSAC, a novel concurrency mechanism that exploits semantics of operations to allow transactions to proceed in parallel if it can be detected that their effects are independent (section 2.3).

- We describe the implementation of PSAC based on Rebel, targeting the Akka actor framework, which provides the basis for our experimental setup (section 2.4).
- We evaluate the performance of both 2PL/2PC and PSAC, and show that PSAC outperforms 2PL/2PC in high-contention scenarios (section 2.5).

The chapter starts with a background on distributed transactions (section 2.2) and concludes with a discussion of the evaluation (section 2.6), related work (section 2.7), further directions for research (section 2.8) and conclusion (section 2.9). Evaluation data is available on Zenodo [95].

## 2.2 Background: Distributed Transactions

Transactions are a mechanism to limit the complexities inherent to concurrent and distributed systems, such as dealing with hardware failure, application crashes, network interruptions, multiple clients writing to same resource, reading of partial updates and data and race conditions [62]. Transactions simplify solving these issues for clients. They group reads and writes together in a logical unit of work, where either all commit, or all abort, even in presence of failures. Transactions can be long running when parties take a long time to respond, for example because of waiting on user input. The safety guarantees for Transactions are ACID [46]: Atomicity, Consistency, Isolation and Durability.

Historically Isolation in ACID guarantees serializability for transactions, meaning operations take effect in a manner equivalent to some serial schedule. However, modern database systems offer a range of isolation properties weaker than serializability [9]. The reason is the trade-off between safety guarantees and performance of the database. Weaker isolation guarantees allow for optimization in performance, especially in a distributed systems setting, where coordination is expensive due to network latency.

This is related to a trade-off in the level of details in the specification of an application. The more that is explicitly known about an application's correctness criteria, the more specific the isolation guarantees can be specialized. In the general case you have to fall back to stronger isolation guarantees. PSAC should simulate the behavior of 2PL/2PC on the object level; since we assume specifications are also on the object level. Strong system-wide guarantees such as serializability are not scrutinized in the current paper, although there is a discussion in section 2.6.2.

## 2.2 Background: Distributed Transactions

Implementing distributed transactions for distributed objects is the focus of this chapter. We use the available semantic knowledge to trade some global isolation guarantees for more local performance, resulting in lower latency and higher throughput.

Distributed Transactions ensure atomicity over multiple application nodes or distributed objects. Two-Phase Locking (2PL) [13, 40, 84] is a concurrency control mechanism and makes sure that serializable isolation is maintained on the application nodes. Two-Phase Commit (2PC) [13, 40, 84] is an atomic commitment protocol that guarantees Atomicity and Durability.

**Concurrency Control and Two-Phase Locking** Consider a bank account object with withdrawal and deposit methods, where a withdrawal should never make an account balance negative. Without concurrency control, it could be the case that two withdrawal actions are simultaneously applied to the account. Both read the same balance and find that individually they do not make the balance negative and are executed, but together they do make the balance negative, violating the invariant. In a serializable situation this is not allowed, since only an outcome state equivalent to a serial execution of both actions would be valid.

Two-Phase Locking (2PL) is a concurrency control mechanism that guarantees serializable isolation (the I in ACID) for a local node or resource. 2PL uses locking to make sure no concurrent changes are made to a resource. It achieves this by using two phases, a growing phase and a shrinking phase in this strict order.

In the withdrawal example, the account resource is locked when the action starts and waits until the first withdrawal action is completed before accepting new actions.

**Atomic Commit and Two-Phase Commit** Two-Phase Commit (2PC) is an atomic commitment protocol that guarantees Atomicity and Durability (from ACID). In itself it does not guarantee Consistency and Isolation. Consistency is achieved by making sure the application invariants are maintained by all operations on the resource. The protocol consists of one Transaction Coordinator and multiple Transaction Participants per transaction. Their internal state is persisted to a durable log, and thus can be recovered in case of failure. The coordinator asks the participants to vote on the transaction. If all participants respond with YES, the coordinator tells them to commit the transactions. If any votes NO, the coordinator tells them to abort. When a participant voted YES, it promises

that it will commit when the coordinator requests it, even in case of failures. 2PC is considered blocking, because if the coordinator fails in the specific case when participants have voted YES, but not yet received a commit decision by the coordinator, the participants are blocked until the coordinator recovers.

**Distributed Transactions** 2PC and 2PL are combined to implement ACID distributed transactions. 2PL's locks are only released when the 2PC transactions are finalized.

2PL locks the resource even though a new incoming transaction might be compatible with the current in-progress transaction, and coordination between the two actions is not actually necessary. This depends on the functional application requirements, which could be less strict than serializability while still maintaining all internal consistency guarantees. The key idea of PSAC is to use available semantic knowledge to determine this, e.g. the outcome of the first withdrawal can never interfere with the acceptance of the second withdrawal when enough run-time balance is available for both. The incoming transaction can be already started, without violating consistency of the balance with respect to its specification. We explore this idea in the next section.

### 2.3 Path-Sensitive Atomic Commit (PSAC)

In this section we present Path-Sensitive Atomic Commit (PSAC), which exploits statically known preconditions and post-effects to prevent unnecessary locking at run time, and thus increases performance of the overall system in terms of throughput and availability. Intuitively PSAC, like 2PL, is a blocking access protocol between transaction and object, but instead of the opaque "locked" indicator of 2PL, PSAC filters incoming actions which would interact with concurrent actions while letting independent actions through. The strictness of the gate is determined at run time using the possible outcomes of in-progress actions determined by the post-effects, and the preconditions to validate the incoming actions against the outcomes.

Previous work [100] defines independent actions as follows:  $IE(e_1, e_2, s) = \forall s' \in State. pre(e_1, s) \wedge post(e_1, s, s') \rightarrow (pre(e_2, s) \leftrightarrow pre(e_2, s'))$ . An action  $e_2$  is independent of an in-progress action  $e_1$  in run-time state  $s$ , if and only if its preconditions check result is the same in  $s$  and in the post state  $s'$ , where  $e_1$ 's effect is applied, e.g. two withdrawals when enough balance is available

## 2.3 Path-Sensitive Atomic Commit (PSAC)

on the bank account. In order for PSAC to leverage this at run time, the pre- and post-conditions are required to be locally checkable and computable, and totally denote the actions' effects.

PSAC gives the same atomicity and linearizability guarantees as 2PL/2PC, while allowing higher throughput when no local dependency exists. Serializability is not guaranteed, which is discussed in section 2.6.2. Linearizability guarantees an atomic real-time ordering of operations on a single object, as opposed to the global, multiple object-guarantees of serializability. Functional correctness in the local participant is maintained and actions' effects are applied in the original order of arrival.

PSAC combines a variant of 2PL with locks that take the semantics of the actions into account with 2PC. Each resource can have a shared lock when it can be determined that actions are semantically independent. This includes commutative actions. However, even for non-commutative actions PSAC will potentially avoid blocking if actions are independent in the current run-time state, e.g., two withdrawal actions are non-commutative, but will run in parallel by PSAC if the run-time balance is sufficient for both because neither of them would affect the success or failure of the other one.

PSAC is faster in accepting actions and increases parallelism when possible, and falls back to the safe 2PL locking approach when not enough information is available. In practice, we limit the number of allowed in-progress actions to be sure that the system can make progress and is not overflowed with accepting new actions on objects. As a consequence, when limiting the maximum number of parallel actions to 1, PSAC degrades gracefully to standard 2PL/2PC, since new actions are delayed until the single in-progress action's lock clears.

In a scenario with many participants and many requests, but in different transactions (low contention), an application using 2PL/2PC (or PSAC) is embarrassingly parallel. This means that each of the participants can do their own computations without the need to synchronize with others. These kinds of computations are more easily spread over multiple application nodes.

PSAC's performance gain over 2PL/2PC becomes evident when multiple actions on the same participant are requested in an overlapping time span. The ability to do parallel processing when application invariants allow it, results in less waiting, and thus more throughput. It also results in processing actions that would otherwise have timed out. This benefit becomes clear at a higher request rate, especially in a higher contention use case, when a few objects are participating in many transactions.

On the other hand, there is also an upper bound to the performance improvement of PSAC over 2PL/2PC. If the servers running this application are already maxed out on one or more resources, such as CPU, memory or network bandwidth, we expect less improvement, because PSAC can no longer trade the extra CPU cycles for extra precondition computations and the extra parallel transactions. In the high-contention use-case with a high number of requests, 2PL waits most of the time on locks to clear and many resources are underused. Here lies the biggest performance gain for PSAC.

### 2.3.1 PSAC in action

Figure 2.1 and figure 2.2 visualize the general difference between 2PL/2PC and PSAC when two actions arrive at the same object in a small time frame. Both figures depict an object sequence diagram. Comment boxes show the internal state of the object, with actions in parenthesis as pending updates. Arrows denote sending and receiving of messages, with withdrawals of  $\epsilon_i$  depicted as ‘ $-\epsilon_i$ ’. “apply” and “defer” respectively denote applying of effects and deferring committed effects until later.

Figure 2.1, on the left, shows the sequence of events when using 2PL/2PC to synchronize. Consider an account object with balance  $\epsilon_{100}$  and a precondition check on the withdrawal action that prohibits a negative balance after withdrawal. When withdrawal action  $C_1$  ( $-\epsilon_{30}$ ) arrives ①, its preconditions are checked against the current balance.  $C_1$  is allowed, the resource is locked and a new 2PC-transaction starts. Even though the account allows the transaction, it is not yet known if the transaction will be committed or aborted by the coordinator, due to processing in other transaction participants. Then another withdrawal action  $C_2$  ( $-\epsilon_{50}$ ) arrives ②. Because the account object is locked, the action is delayed. When  $C_1$  commits ③, its effects are applied to the account state, resulting in a new balance of  $\epsilon_{70}$  and the object is unlocked. Now, the delayed withdrawal  $C_2$  can start, eventually it commits ④ and its effect is applied. This results in the new state of  $\epsilon_{20}$ . 2PL effectively serializes the two parallel transactions.

The amount of locking performed by 2PL/2PC can be problematic in situations where a lot of transactions happen on a single object. For instance, in the case of ING Bank, when the tax authority pays out benefits to citizens, the bank is required to handle all these transactions within a specific time frame. The tax authority’s bank account is highly contended because it is involved with all

### 2.3 Path-Sensitive Atomic Commit (PSAC)

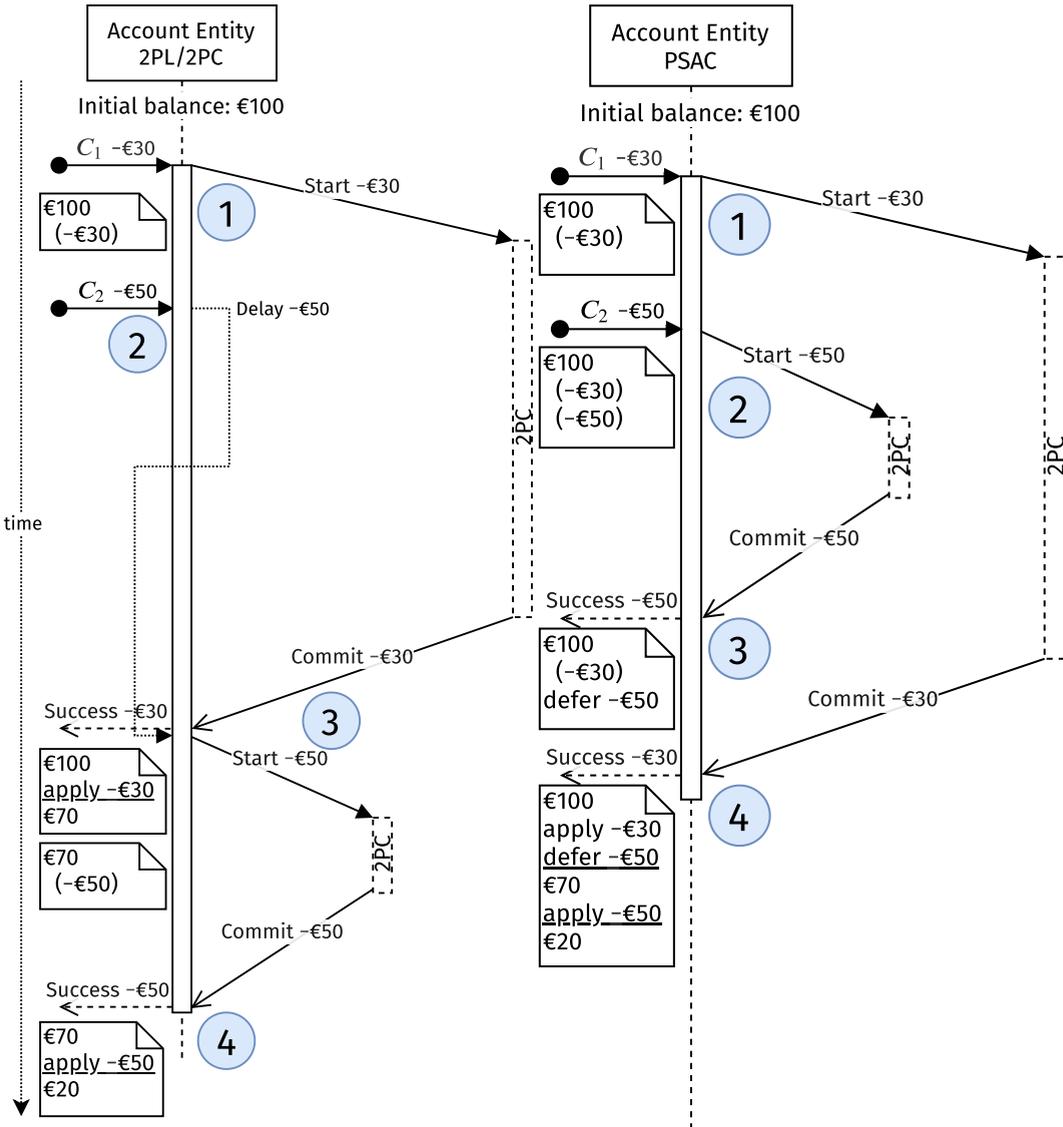


Figure 2.1 Vanilla Two-Phase Commit

Figure 2.2 Path-Sensitive Atomic Commit

individual transfers. This would not scale on such an object-oriented message-based distributed system, because each withdrawal will have to wait on the previous to finish.

PSAC improves on this situation by detecting at run time if transactions can be processed in parallel anyway. The same execution scenario is visualized in figure 2.2, illustrating how PSAC differs from 2PL. We again consider an account object with initial state €100 and a precondition check that prohibits a negative balance. Similar to 2PL, when withdrawal action  $C_1$  is received ① and no other transactions are in progress, a new 2PC-transaction is started, but contrary to 2PL, the object is not completely blocked. When another withdrawal action  $C_2$  arrives ②, it is started because it is independent of whether the earlier action commits or aborts, since there is enough balance to allow the withdrawal to proceed in either case. Therefore,  $C_2$  is immediately started. PSAC can detect this independence, based on in-depth knowledge of the functionality of a bank account via the preconditions and post-effects of its actions.

In the example scenario,  $C_2$  commits ③ earlier than  $C_1$ , but its effect is delayed to maintain linearizability of the account. The original requester can be already notified of the successful result (Success -€50), but not yet the new state of the account, since this is dependent on the outcome of  $C_1$ .  $C_2$ 's effect is deferred. Now, when  $C_1$  commits ④, both effects are applied in order to the account, resulting in a new state of €20.

In situations with non-uniform loads, PSAC delivers on allowing more transactions per time span than 2PL/2PC (and thus higher scalability in terms of throughput). An example with abort is shown in appendix A.1. We detail the algorithm below and evaluate these claims in section 2.5.

### 2.3.2 PSAC Algorithm

Listing 2.1 shows the PSAC algorithm in pseudo-code. The algorithm maintains three lists, `InProgress` containing transactions that have been started, but have not finished yet; `Delayed`, containing the deferred transactions that have to wait till at least one of the in-progress transactions completes; and finally `Queued`, containing the transactions that are successful, but not yet applied to the state of the object, to maintain the original order of arrival.

On arrival of a command  $C_{new}$ , its preconditions are checked against all possible outcomes of the transactions that are currently in progress. If it is allowed in all possible states, the action is independent and can start processing.

## 2.3 Path-Sensitive Atomic Commit (PSAC)

**Listing 2.1** Pseudo-code of a PSAC-enabled object

```

1 inProgress = []
2 delayed = []
3 queued = []
4
5 while true:
6   if incoming command  $C_{new}$ :
7     #See figure 2.3 for this part of the algorithm.
8      $S$  = set of all possible outcome states of
9        $\hookrightarrow C_i \in \text{inProgress}$ 
10    if  $\forall s \in S$ . preconditions of  $C_{new}$  hold:
11      inProgress +=  $C_{new}$ 
12      start  $C_{new}$ 
13    else if  $\neg \exists s \in S$  such that preconditions
14       $\hookrightarrow$  of  $C_{new}$  hold:
15      reply Fail( $C_{new}$ ) to requester of  $C_{new}$ 
16    else if commit of  $C_n$ :
17      reply Success( $C_n$ ) to requester of  $C_n$ 
18      queued +=  $C_n$ 
19
20    else if abort of  $C_n$ :
21      reply Fail( $C_n$ ) to requester of  $C_n$ 
22      inProgress -=  $C_n$ 
23
24     $C_m$  = head(inProgress)
25    if  $C_m \in \text{queued}$ :
26      apply  $C_m$ 
27      inProgress -=  $C_m$ 
28      queued -=  $C_m$ 
29      currentDelayed = delayed
30      delayed = []
31    for  $C_i$  in currentDelayed:
32      handle  $C_i$  as incoming command

```

For such transactions it is as if the object is not locked. If there is no possible outcome where the preconditions of  $C_{new}$  hold, the action is immediately rejected with a failure reply. Otherwise, if there is at least one possible state where the preconditions of  $C_{new}$  hold, the action is dependent on one of the transactions that are currently in progress, so it is delayed by adding it to delayed. For such a transaction, the semantics of PSAC is equal to 2PL.

Whenever an action commits, it is queued for applying the effects to the object's state. Since all actions are stored in order of arrival in inProgress, it will be applied to the state in the same order. This way non-commutative actions do not violate linearizability. If a transaction aborts, the requester is notified of the failure, and it is removed from the inProgress list. Finally, if the first element of inProgress is in queued, its effects are applied to the state, it is removed from inProgress and queued, and all delayed actions are retried. This results in applying the effects in original arrival order and makes sure delayed actions are retried as soon as possible.

The key idea of the algorithm is the use of the preconditions and actions' effects to construct a tree of all possible outcome states of the set of transactions that are currently in progress. At run time, given the current object state, the set of in-progress actions and the new incoming action, we calculate all possible

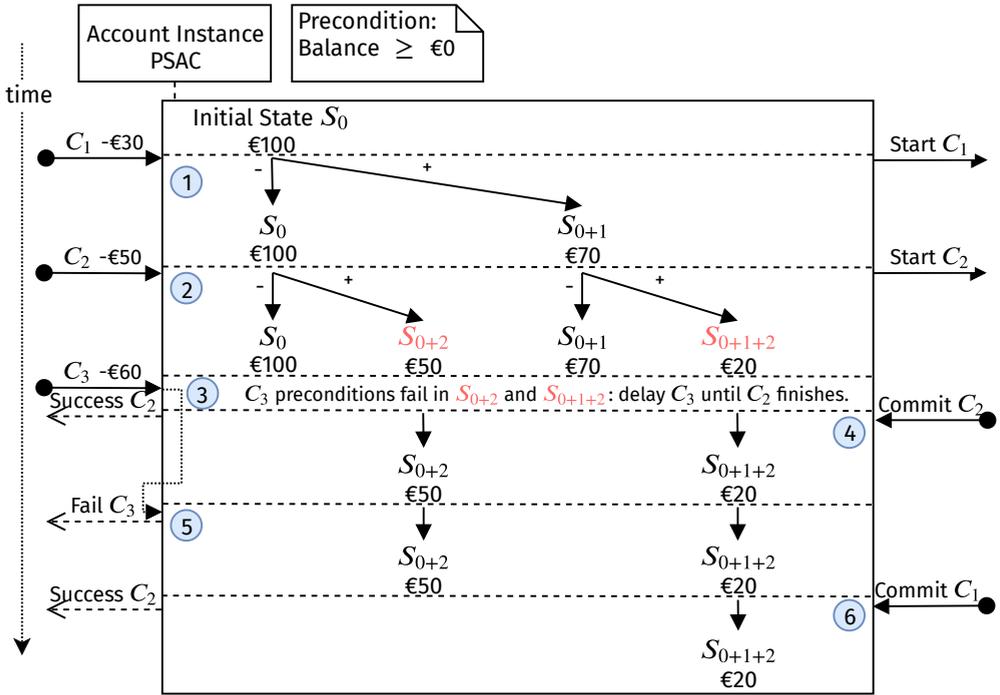


Figure 2.3 PSAC example with internal possible outcome tree and decisions on commands

outcome states of the in-progress actions using the post-effects. This is done by simulating the first in-progress action in the current state, branching into two possible outcomes: one where the in-progress action actually commits and the post-effect is applied, and one where it is aborted and thus not applied. Doing this for all in-progress actions results in a tree with in its leaves the possible outcome states of the object.

figure 2.3 shows an example of the potential outcome tree  $S$  corresponding to the scenario of figure 2.2 and how it is updated when actions arrive. The leaves represent the potential outcomes. Withdrawal  $C_1$  ( $-\text{€}30$ ) arrives ① at a bank account instance using PSAC. The preconditions are valid for  $C_1$ , and given tentative Abort ( $-$ ) or Commit ( $+$ ) by the 2PC transaction, the possible outcome tree branches to two possible outcomes:  $S_0$  and  $S_{0+1}$ , respectively corresponding to a balance of  $\text{€}100$  and  $\text{€}70$ . Withdrawal  $C_2$  ( $-\text{€}50$ ) arrives ② and its preconditions are valid in all possible outcomes  $S_0$  and  $S_{0+1}$ , so both possible outcomes branch in similar fashion. Withdrawal  $C_3$  ( $-\text{€}60$ ) arrives

③, but its preconditions are *not* valid in all possible states, in particular not in  $S_{0+2}$  and  $S_{0+1+2}$ .  $C_3$  is delayed until it is independent from the in-progress actions. In this case  $C_3$  is only dependent on  $C_2$ . The outcome tree is unchanged, since  $C_3$  is not accepted for processing yet. When  $C_2$  is committed by the 2PC coordinator ④, the possible outcome tree is pruned, because the branches where  $C_2$  is aborted are no longer valid, leaving only  $S_{0+2}$  and  $S_{0+1+2}$ . After an in-progress action commits or aborts, in this case  $C_2$  ⑤, delayed actions are retried, here  $C_3$ . Now preconditions fail in all possible outcome states,  $C_3$  is independent and thus rejected. When  $C_1$  commits ⑥, the possible outcome tree is pruned again and a single state  $S_{0+1+2}$  remains. The new state is now calculated by applying the effects in order.

Given all possible outcome states we can check the new incoming action against all outcomes using its precondition. This gives insight if the action conflicts with any in-progress action or combinations thereof. If all or none of possible outcomes satisfy the preconditions, the incoming action is independent and is accepted for processing or immediately rejected.

A difference from 2PL/2PC is that actions that come in later could be accepted for commit earlier. Then the effect of the action is delayed until after the previous actions are committed or aborted, making sure that linearizability of the object is maintained.

## 2.4 Implementation: Rebel and Akka

To compare PSAC to 2PL/2PC in a realistic environment, we prototyped a small accounting service on top of Akka. For the pre- and post-conditions of transactions we use the Rebel specification language, which aligns with the design requirements of PSAC. Our specific use of Rebel and Akka are not essential to the operation of PSAC but they are part of our evaluation setup for the performance evaluation in section 2.5.

### 2.4.1 Rebel: a DSL for Financial Products

Rebel is a domain specific language (DSL) for describing financial products, designed in collaboration with ING Bank, as an experiment to tame the complexity of large financial IT landscapes [105, 106, 108]. Declarative specifications functionally describe financial products, such as current- and savings accounts

and financial transfers between them. Rebel specifications are designed to facilitate unambiguous communication with domain experts, support simulation, verification, testing, and execution through code generation. Rebel and proprietary derivatives are used by ING Bank to prototype and understand many different financial products, such as European payments and open data regulations, banking cards and business lending use cases. For example the SEPA specifications consist of 26 Rebel specifications, totaling 964 lines of code.

An example similar to the bank account example used throughout this chapter is shown in a Rebel-like specification in listing 2.2. A specification declares an identity (using the annotation `@identity`), data fields, and describes the life cycle of a product as a state machine with actions and pre- and post-conditions on those actions in predicate logic plus integer constraints.

Listing 2.2 shows the specification of two classes, **Account** and **MoneyTransfer**. An **Account** is identified by its IBAN bank account number, and has a current balance. The life cycle of an account is as follows: it can be opened, then any number of withdrawals and deposits may occur, and finally it can be closed. Transitions among states are triggered by the actions `Open`, `Withdraw`, `Deposit`, and `Close` respectively. Each event is guarded by preconditions and describes its effect in terms of post-conditions. For instance, the `Withdraw` action requires that the withdrawn amount is greater than zero, and that the withdrawal does not produce a negative balance. The effect of withdrawal is then specified as a post-condition on the balance of this account.

The second class **MoneyTransfer** in listing 2.2 models a transfer of money between two accounts. It can simply be booked via the `Book` action. The `Book` action is triggered on two accounts. The effect of booking a money transfer consists of *synchronizing* the `Withdraw` event on the from account, with the `Deposit` event on the to account. The `sync` represents an atomic transaction between two or more entities. In other words, an underlying implementation must guarantee that either both `Withdraw` and `Deposit` should fail or both should succeed.

The fact that the functional requirements on financial products are formally specified in Rebel separates the “what” from the “how”. In other words, decoupling the description of a financial product from its implementation platform allows us to experiment with different back-ends for Rebel specifications, by developing different code generators for different platforms or different runtime architectures. Below we show how Rebel classes are mapped to Scala classes that can be executed as actors on the Akka platform. In particular this

**Listing 2.2** Rebel specification and state charts of a simple bank account: an **Account** supports events Open, Withdraw, Deposit, and Close. A **MoneyTransfer** can be booked by synchronizing Withdraw and Deposit on two accounts.

```

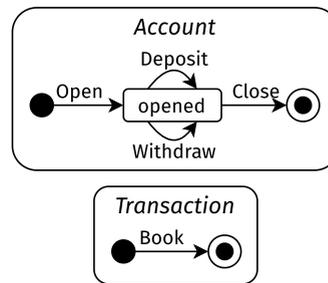
1 class Account
2   accountNumber: Iban @identity
3   balance: Money
4
5   initial init
6   on Open(initialDeposit: Money): opened
7     pre: initialDeposit ≥ €0
8     post: this.balance ≡ initialDeposit
9
10  opened
11  on Withdraw(amount: Money): opened
12    pre: amount > €0, balance - amount ≥
      ↳ €0
13    post: this.balance ≡ balance - amount
14  on Deposit(amount: Money): opened
15    pre: amount > €0
16    post: this.balance ≡ balance + amount
17  on Close(): closed
18
19  final closed

```

```

1 class MoneyTransfer
2   initial init
3   on Book(amount: Money, to: Account,
      ↳ from: Account): booked
4   sync:
5     from.Withdraw(amount)
6     to.Deposit(amount)
7   final booked

```



allows us to experiment with different implementations of the **sync** construct, such as 2PL/2PC and PSAC.

The consistency of the Rebel classes is fully determined by their life-cycle and pre- and post-conditions and local to the class specification. Isolation guarantees however are undefined for Rebel synchronization [108], although Rebel's simulator and model checker use sequential non-overlapping events, which implies serializability.

### 2.4.2 Executing Rebel on Akka

**Deployment** To support fault tolerance and scalability, the execution of Rebel entities is deployed on at least two servers so that customer requests can still be processed when one of the servers breaks down. This means the generated application is a distributed system. One style of implementing a distributed system is by using the actor model [53]. Akka [3] is a well-known toolbox for

actor-based systems that runs on the JVM and is widely used to build distributed, message-driven applications. Mapping Rebel objects to Akka actors is a natural fit and provides sufficient low-level controls to vary the implementation of the SYNC construct. This implementation approach is similar to other reactive architectures such as presented by Debski et al. [26].

Each concrete Rebel class instance is run as an actor in isolation and enables distribution of the computation over multiple cluster nodes. Class instance actors are automatically spread over the available cluster nodes to allow for more optimal spread over resources such as RAM and CPU. This enables scaling in and out by moving the actors to other nodes if needed. Each actor runs as an independent object, so it performs work without having to wait on other actors, allowing concurrent work. In theory this means that actor systems scale out linearly, until they have to synchronize. In practice this means that an actor system scales up until too many of its actors are blocked by multiple transactions at the same time.

**Rebel to Akka** Each instance or entity of a Rebel state machine is implemented as an actor. We use the following features of Akka: `CLUSTER` for cluster management and communication between application nodes; `SHARDING` for distributing actors over the cluster by sharding on the identity; `PERSISTENCE` enables event sourcing for durable storage and recovery; and `HTTP` for HTTP endpoints definitions and connection management. These combined Akka features allow us to spread the Rebel instance actors over a dynamically sized cluster of application nodes. More details on the implementation using Akka are found in appendix A.3. The back-end for persistence is an append-only event sourcing log, for which we use the distributed and linearly scalable Cassandra database [20].

The runtime guarantees that there is a single actor instance per Rebel class instance and thus guarantees linearizability on instance level, in the sense that operations always see all previous updates. Each operation is persisted to the journal before processing the next, to allow for recoverability and durability in case of failure. The journal data is replicated over three Cassandra nodes. Reads and writes use the built-in `QUORUM` consistency level of Cassandra to make sure no stale data is read.

An example of the generated Scala code for the **Account** and **MoneyTransfer** example of listing 2.2 is shown and explained in appendix A.2.

**Synchronization** We first consider the 2PL/2PC synchronization strategy. Our implementation of 2PC follows the description by Tanenbaum and Van Steen [110] extended with the flattened commit protocol [115] to support nested synchronization in Rebel, where 2PC participants can add more transaction participants. As optimization the transaction manager does not wait on the votes of the other participants and immediately aborts the transaction when one participant aborts. There is a single transaction manager per transaction and one or more transaction participants, respectively implemented by Akka PERSISTENT FSMs named `TransactionManager` and `TransactionParticipant`. They both define a state machine following the definition and also persist their state to the persistence back-end, and thus can be recovered in case of failure.

Both manager and participants have timeouts on their initial states, this means that when no initialization message is received within a given time duration, they will timeout and abort the transaction. This makes sure that the system does not deadlock, although it might result in overhead in creation of transaction actors and messaging when lots of timeouts are triggered.

To make sure no deadlocks happen in other states, timeouts are in place that trigger retries and eventually stop the actor. In the unlikely case that a participant or coordinator is not running, the combination of Akka SHARDING and PERSISTENCE will make sure it is restarted. This also works when some of the cluster nodes shut down, are killed, or become unreachable for whatever reason; in that case other nodes will take over automatically,<sup>2</sup> restore the actors and continue the protocol. The blocking aspect of 2PC, when a transaction manager crashes, is also partly mitigated by message retries and recovering on another application node.

PSAC is implemented on top of 2PC. Whenever a new action is received by the actor, an action decider function decides if the action can be safely executed concurrently. If the configurable maximum number of parallel transactions per actor is reached, the action is queued. Otherwise, it calculates the possible outcome states by iterating all the possible in-progress action interleavings and checks the preconditions in the calculated states to decide if it can safely start the 2PC transaction for this action. If dependency is detected, the action is also queued. Note that reducing the maximum number of parallel transactions to 1 results in vanilla 2PL/2PC behavior.

---

<sup>2</sup> The fundamental problem of determining when to fail over, because node failure, slowness and network delay are indistinguishable, is out of scope for this chapter.

## 2.5 Performance Evaluation

### 2.5.1 Research Objectives

In this section, we evaluate the performance of PSAC relative to 2PL/2PC. First, we find out in which scenarios 2PL/2PC is sufficient as a Rebel synchronization back-end and in which scenarios it can no longer maintain sufficient performance. Furthermore, we are interested in determining when PSAC performs better for the cases where 2PL/2PC is no longer sufficient. In order to look at applications that can scale with business requirements, we focus on scalable and resilient applications that can continue to grow when performance demands keep growing. We study applications that can scale over multiple servers.

The experiments are created to fairly compare PSAC and 2PL/2PC against each other in the same synthetic scenarios with same load and configuration. We are interested in the scalability of both 2PL/2PC and PSAC under similar loads. In other words, we are interested in to what extent the throughput increases when more nodes are added to the cluster.

It might seem counter-intuitive that the extra work in PSAC of calculating the possible outcomes tree and checking the preconditions against all of these states, can result in higher performance compared to 2PL/2PC. For an ideally-scheduled batch based system all extra calculations would worsen performance, since every CPU cycle counts. In this case, the most time in 2PL/2PC is lost by waiting for the unlock. PSAC's parallel transactions use this otherwise lost time in between for these extra calculations, to determine safe extra parallelization.

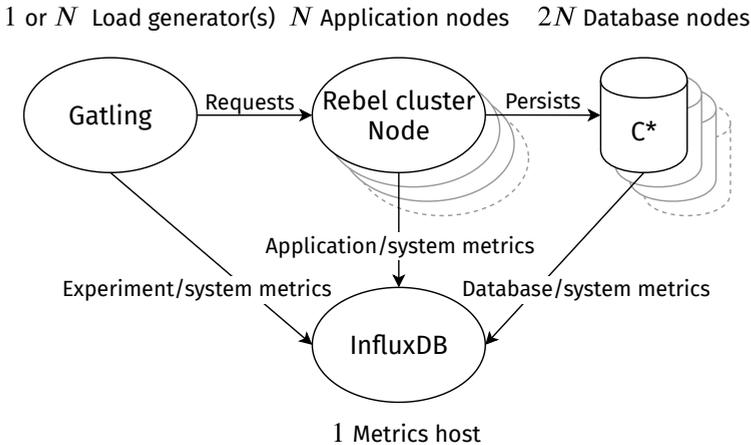
We expect that:

**Hypothesis 1.** 2PL/2PC and PSAC perform similarly in maximum sustainable throughput for actions without synchronization, because objects do not have to wait on each other.

**Hypothesis 2.** 2PL/2PC and PSAC perform similarly in maximum sustainable throughput for actions with low contention synchronization, because synchronization is evenly spread over the objects.

**Hypothesis 3.** PSAC performs better than 2PL/2PC in maximum sustainable throughput for actions with high-contention synchronization, because 2PL/2PC has to block for in-progress actions, where PSAC allows multiple parallel transactions.

Before it can be determined if PSAC is generally useful, first we need to find out whether PSAC pays off in high-contention scenarios. Since PSAC is a



**Figure 2.4** Experiment setup on  $N$  nodes. A single or more load generators do HTTP requests randomly over  $N$  Rebel application nodes, backed by  $2N$  Cassandra database nodes. Relevant experiment and system metrics are reported to InfluxDB for later analysis.

new algorithm running a complex technological context the answers to these hypotheses are not trivial: first the expected gain may not be significant compared to other relevant factors and second the cost of the additional overhead for every transaction may outweigh the benefits. So, these experiments are designed to first isolate the effect of PSAC as compared to 2PL/2PC, and then to try and invalidate the above hypotheses. If the experiments can not invalidate our claims, then we gain confidence in the relevance of the new algorithm.

To be sure that we are not running into the limits of (configuring) the infrastructure, but really into limits of the synchronization implementation, we investigate first how far we can take the Akka infrastructure without any logic or synchronization.

**Hypothesis o.** The actor system infrastructure enables horizontal scalability, which means that adding more compute nodes increases throughput.

### 2.5.2 Deployment Setup

In order to scale to multiple nodes, our experiment setup runs on Amazon ECS (Elastic Container Service) using Docker images for the Database (Cassandra), the Application, Metrics (InfluxDB), and the Load generator (Gatling [36]). Figure 2.4 shows an overview of the setup. The Cassandra version is 3.11.2

on OpenJDK 64-Bit Server VM/1.8.0\_171. The application runs on Akka version 2.5.13, Oracle Java 1.8.0\_172-b11, with tuned garbage collector G1 with `MAXGC_PAUSEMILLIS=100`.

In order to prevent CPU or memory starvation/contention between the application and the load generator tool, we deploy each of the application components on a different virtual host on Amazon Web Services (AWS). We use EC2 instance type `M4.XLARGE`<sup>3</sup> for all VMs, which are located in the Frankfurt region in a single data center and availability zone.

Each of these containers is deployed on its own container instance (host), with the exception of Metrics and Load generator, which share a host. Metrics being sent asynchronously over UDP, to ensure minimal interference with application performance. CPU and other system metrics are monitored to prevent this.

For realism of the experiments we use the production-ready persistent journal implementation Cassandra as an append-only log for the persistent actors, so limited synchronization is done on the database level, although it gives realistic overhead. We over-provision the database to make sure it is not a bottleneck.

Our tooling supports running the performance load from multiple nodes. Experimentally we discovered that setting up the correct experiment for high load is not trivial: such as the correct number of file descriptors for connections; garbage collection tuning; library versions with bugs; careful load generation to capture the sustainable throughput; ratio of application, load and database nodes; collection of metrics for all components; and validating correct deployment before running the experiment. We collect system metrics for all machines in order to monitor overload of any specific part. The low-overhead JDK Flight Recorder profiling is also enabled for after-the-fact bottleneck analysis of our application nodes. The experiment metrics results and profiling files are available at Zenodo [95].

When load testing applications, the crafting of the load is very important, and not trivial. A distinction often used is closed versus open systems [92]. Closed systems have a fixed number of users, each doing requests to the service, one after another, limiting the total number of TCP connections. Open systems have a stream of users requesting at a certain rate, meaning there is no such maximum of concurrent requests as in a closed system. Typically closed systems are used for batch systems and open systems for online usage.

---

<sup>3</sup> `M4.XLARGE`: 4 vCPU, 16 GiB Memory, EBS-based SSD storage, 750 Mbit/s network bandwidth.

For all experiments presented in this chapter, we employ a closed system workload approach. Finding the maximum throughput using an open-world workload quickly results in an overloaded application, both for 2PL/2PC and PSAC, which obscures the differences between them. In an enterprise setting, such as a bank, a (hardware) load balancer translates the open workload behavior to a more closed world behavior by limiting the number of network connections and reusing them.

Each request from the load generator to the application will spawn a 2PC coordinator actor for the request a 2PC participant actor for each synchronization participant. For the bank transfer experiments, this means that for each request, a new Rebel **MoneyTransfer** entity actor is started, one 2PC coordinator actor, and three 2PC participants (for the money transfer and the two accounts). So the number of actors created is roughly five times the number of requests. For our experiment scenarios all actors are equally spread over all the Akka cluster nodes.

### 2.5.3 Baseline Experiments: Akka Scalability

To make sure that Akka or our setup does not influence the result of evaluating the performance of PSAC, we run four experiments on top of plain Akka to establish horizontal scalability. The goal of this experiment is to isolate (environment) noise and reduce confounding factors. The baseline experiments use a setup as similar as possible to the more involved experiments discussed later. We run multiple variants that increase in complexity, building up to all the features used by the Rebel implementation, and measure the maximum sustainable throughput (requests/transactions per second) per each increment of application complexity.

The following experiments were run:

1. BARE – HTTP: responses are immediately given by the HTTP layer.
2. SIMPLE – HTTP + Actors: each request creates an actor which sends the response.
3. SHARDING – HTTP + Sharded Actors: actors are equally spread over the cluster and send the response
4. PERSISTENCE – HTTP + Sharded Persistent Actors: actors are spread equally over the cluster and wait for a successful write to the persistence layer (Cassandra) before responding to the request.

**Table 2.1** Baseline experiment fit to Amdahl’s law and asymptote

experiment	$\lambda$ (tps)	$\sigma$	$a_{inf} = \lambda\sigma^{-1}$ (tps)
BARE	16 751	0.002 923 3	5 729 998
SIMPLE	10 372	0.000 877 3	11 822 028
SHARDING	6303	0.004 728 5	1 332 920
PERSISTENCE	1928	0.008 159 7	236 281

The application responds with a JSON message when the work is described is done. A request is successful when a 200 HTTP status code is received.

Figure 2.5 shows the throughput results of the experiments. Data points are throughput in terms of successful responses per second during the stable load of the experiment, after warm-up and ramp-up of users. For warm- and ramp-up we increase the number of simulated users over time, to give the application some time to get up to speed. The plot also shows a fit to Amdahl’s [6] law using a non-linear least squares regression analysis. For intuitive comparison we include the upper bound of linear scalability line for each of the experiments. Amdahl’s law is defined as:  $X(N) = \frac{\lambda N}{1 + \sigma(N-1)}$ , where  $X(N)$  is the throughput when  $N$  nodes are used. Linear scalability means that the contention  $\sigma$  is 0 and the throughput grows with  $\lambda$ , which denotes the throughput of the single application node. The fitted values for  $\lambda$  and  $\sigma$  are shown in table 2.1.

All variants have very different performance per node. This is expected, by the increasingly complex actions performed. Increasingly complex variants have increasing  $\sigma$ , which can be explained by increased synchronization between the Akka application nodes. All experiments show horizontal scalability up until an expected peak throughput on Amdahl’s law asymptote ( $a_{inf} = \lambda\sigma^{-1}$ ), which is the theoretical maximum throughput which can not be further improved by adding more nodes.

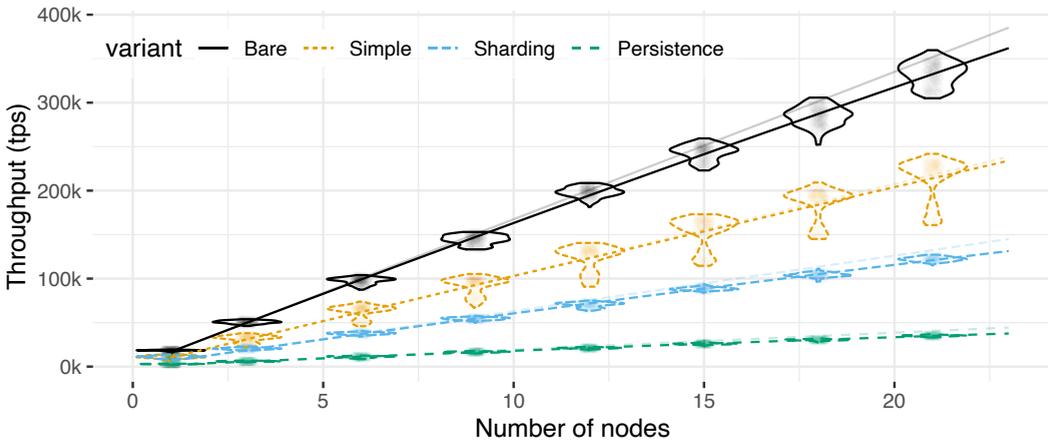
The results show that our implementation using Akka exhibits horizontal scalability and corroborates Ho.

### 2.5.4 Synchronization Experiments: PSAC vs 2PL/2PC

To compare the performance of PSAC and 2PL/2PC, we run three experiments with different synchronization characteristics, linked to the relevant hypothesis:

1. NOSYNC – OPENACCOUNT: A Rebel operation without SYNC. (H1)

## 2.5 Performance Evaluation



**Figure 2.5** Throughput  $X(N)$  (violins), Amdahl fit (colored line) and linear scalability upper bound (transparent line) of baseline experiments

2. SYNC – BOOK: A Rebel operation with SYNC, synchronizing with WITHDRAW and DEPOSIT on two accounts. (H2)
3. SYNC1000 – BOOK on a limited number of accounts, to increase the contention. (H3)

These different scenarios enable us to see if and when PSAC improves over 2PL/2PC, especially in the SYNC1000 high-contention experiment. On the one hand NOSYNC and SYNC show where PSAC performs similarly 2PL/2PC. On the other hand SYNC1000 shows the high-contention scenario where PSAC improves over 2PL/2PC.

All three experiments use a closed system approach [92], where we limit the number of concurrent total users. This ensures that the application is not overloaded by too many requests, causing high failure rates. Each experiment is run consecutively for increasing node count  $N$ , with  $N$  load generator nodes (except SYNC1000) to grow the load proportionally. SYNC1000 runs a single load generator which increases the load in incremental steps in order to determine the maximum throughput until the application overloads.

The high-contention scenario SYNC1000 is designed to be as close as possible to a realistic industry setting, where high-contention objects become a bottleneck. This is similar to the NEWORDER benchmark of the well-known TPC-

## Chapter 2 Path-Sensitive Atomic Commit

c [87] online transaction processing benchmark suite, where a high-contention object is responsible for handing out order IDs.

In all experiments we compare 2PL/2PC's and PSAC's throughput ( $X(N)$ ) for a varying number of application nodes  $N$ .

**NO SYNC** The NO SYNC experiment is the Open Account scenario which does not contain a Rebel SYNC. It corresponds to hypothesis H1, which states that 2PL/2PC and PSAC should have similar throughput when there is no synchronization for the actions involved. The results are plotted in figure 2.6a. We observe that the throughput of the two variants is similar, as expected and thus corroborates H1. The throughput is only limited by the CPU-usage on the nodes and the creation of records in the data store. The metrics data shows that the application CPU usage drops to around 80% and the data store CPU usage is almost 100%.

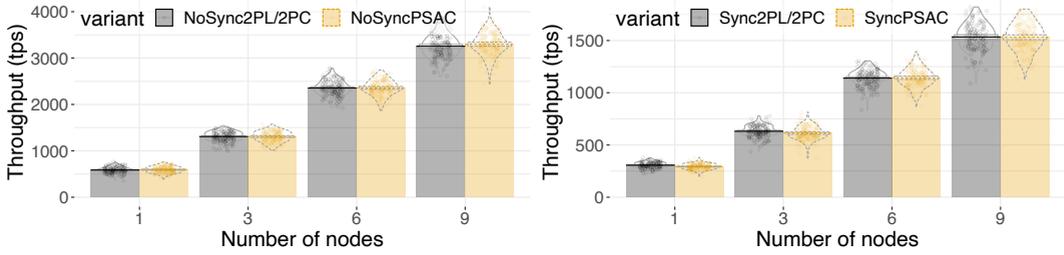
**SYNC** The SYNC experiment contains a SYNC in the BOOK action and corresponds to H2, which states that we expect that PSAC and 2PL/2PC also have similar throughput in this low-contention scenario. The results are shown in figure 2.6b. Here we also see the same performance for both 2PL/2PC and PSAC, corroborating H2. This can be explained by the experiment setup: The BOOK actions are done between two accounts uniformly picked from 100 accounts initialized before the experiment. With a maximum throughput of roughly 1500 and uniformly spread bookings the probability of overlapping transactions on a single account is low.

The absolute throughput numbers are lower than NO SYNC, however, which is explained by the fact that BOOK has to do more work, since it involves three instances: one **MoneyTransfer** and two **Accounts**.

**SYNC1000** Finally, SYNC1000 introduces artificial contention by reducing the number of accounts to 1000, corresponding with H3. H3 states in high-contention scenarios that PSAC is expected to have higher throughput than 2PL/2PC, because it is able to avoid blocking where 2PL/2PC can not. This results in a difference between 2PL/2PC and PSAC, as seen in figure 2.6c. Since this is the most interesting case we have run the experiment for higher node counts, and include a fit on Amdahl's law, shown in figure 2.7. Figure 2.6d contains the fitted parameters. The results show that PSAC consistently achieves higher throughput than 2PL/2PC.

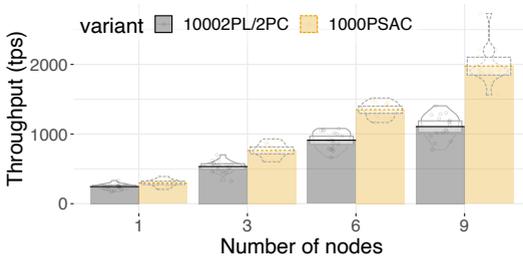
The metrics show that both application and data store CPU usage starts dropping for node counts  $> 9$ . This can be explained by contention: busy

## 2.5 Performance Evaluation



(a) Throughput  $X(N)$  of NoSync

(b) Throughput  $X(N)$  of SYNC



variant	$\lambda$ (tps)	$\sigma$
2PL/2PC	180	0.049 880 9
PSAC	296	0.049 587 8

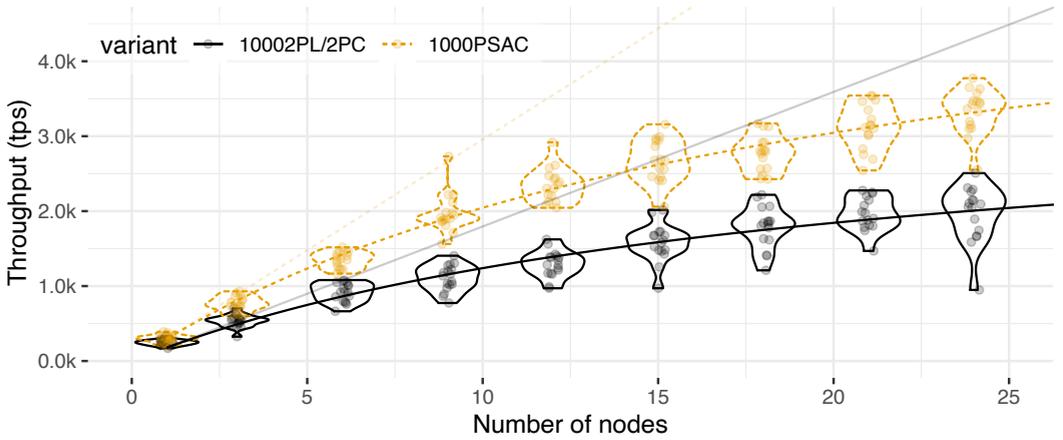
(c) Throughput  $X(N)$  of SYNC1000

(d) SYNC1000 experiment fit to Amdahl's law

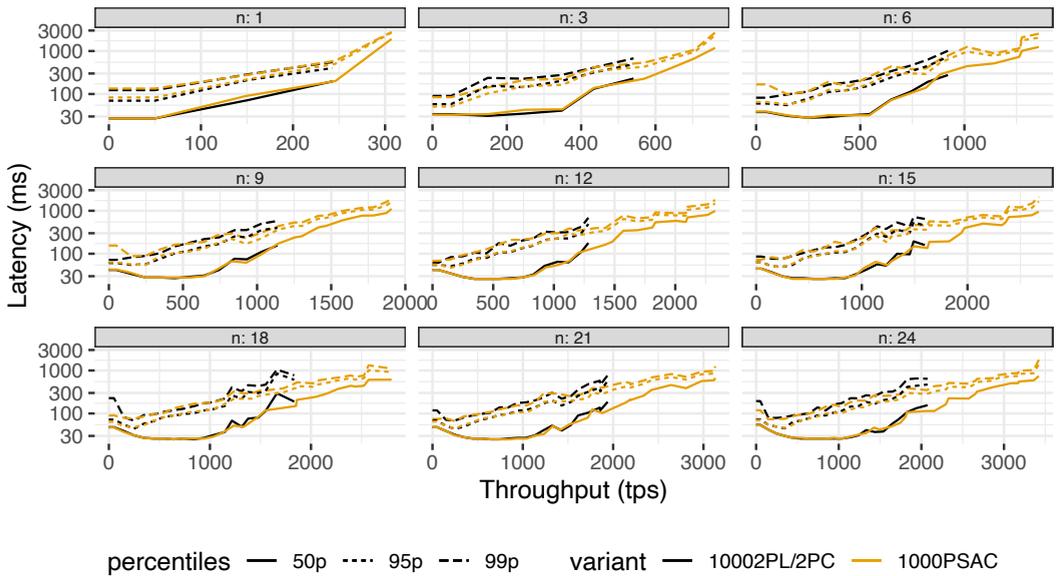
**Figure 2.6** Throughput  $X(N)$  against number of nodes  $N$ . This pirate plot is a combination of violin plot, box plot and bar chart. Line is the median, points are the data points. This gives a complete overview of the data (data points and distribution in violin plot) and an aggregated view.

entities are at their maximum throughput for 2PL/2PC transactions. In the case of PSAC this also happens, because the number of parallel transactions is limited by configuration at 8. Nevertheless, PSAC consistently achieves higher throughput.

The graphs in figure 2.8 display the latency percentiles against increasing throughput. Since the Y-axis of the different graphs is the same, we can see that the latencies for all node sizes are similar, but the throughput grows larger when node size increases. This also shows clearly that PSAC reaches higher throughput levels and that PSAC is on par or better latency-wise with 2PL/2PC up to at least the breaking point of 2PL/2PC, which is explained by the improved parallelism on PSAC.



**Figure 2.7** Plot of Amdahl fit and corresponding linear scalability upper bound (transparent) for SYNC1000 on higher node counts



**Figure 2.8** Latency percentiles (logarithmic scale) of Sync1000/ grouped by number of nodes (n). Y-axis shared for latency comparison, lower is better. X-axis: higher is better.

## 2.6 Discussion

### 2.6.1 Threats to Validity

We distinguish between construct validity, internal and external threats to validity. Construct validity discusses if the test measures what it claims to measure. Internal threats are concerned with problems of configuration and bugs in the implementation. External threats are about the generalization of the results.

**Construct validity** Regarding construct validity we have mitigated this risk by first doing a infrastructure and a NoSYNC experiment (Ho), in order to make sure that we actually measure the intended construct: comparing PSAC against 2PL/2PC. How sure can we be that in our situation with a noisy cloud environment, the results are significant compared to coincidental variation? Our experiments compare between variants under the same benchmark and implementation to be sure we are correctly comparing the relevant synchronization implementation parts of the setup. The baseline experiment (Ho) makes sure that the setup and environment are correctly configured, and provides bounds in throughput and latency in which the results of the actual experiments are to be interpreted. The SYNC1000 experiment is set up in such a way that if PSAC did not significant improve performance, this would be visible in the results. The other experiments (NoSYNC, SYNC1000) are its baseline to show PSAC's and 2PL/2PC's variance is limited in other (low-contention) situations. This shows that PSAC's performance improvement in the high-contention scenario is not due to noise or external factors.

**Internal threats to validity** To make sure there are no differences in configuration and deployment of our experiments, we designed and implemented an experiment runner to automatically run the different scenarios required for each experiment on the available AWS VMs. The experiments are defined using declarative configuration, to make them reproducible and without configuration mistakes. For each experiment each node size is run separately on AWS. The use of Docker images and automated tooling makes sure that the configuration and artifacts for each of the experiments are the same, except for the specific differences that we want to compare.

Another threat is the Amazon virtual machine environment: this can be a noisy environment, which influences our experiments. Nodes are run on possibly shared hosts, which may impact performance depending on noisy neighbors, differences in hardware, or even time of day. Warm up time is frequently the bottleneck in data parallel distributed systems on the JVM [73], so this factor may not be eliminated by our experiments. Experiments may also not have been run long enough to obtain reliable results. We mitigated this threat partially by (a) designing our experiments to compare between variants under the same conditions and (b) running the experiments on many different occasions and manually validating that the results are similar to previous runs. There is a threat that our findings do not generalise to a broader range of scenarios.

Another possible influence on the performance results is the persistence layer. In order to make sure the persistence layer is not the bottleneck, we should monitor metrics of the database nodes, such as CPU, IO and memory usage. If none of them continuously peak, we assume this is not a bottleneck. However, during the execution of some of the experiments the persistence layer has not been monitored consistently.

**External threats to validity** For PSAC to be correct and consistent, the defined pre- and post-conditions have to be precise and fully define the checks and effects of an implementation. In practice PSAC's implementation uses the same non-side-effecting code to calculate the possible outcomes as for the actual state changes. When PSAC is used as part of another implementation, care has to be taken.

The load might be too hard on the system, resulting in higher throughput but worse response times than we want. This could obscure comparison and generalization. For instance, the SYNC1000 experiment for PSAC showed overall higher throughput, but also increasing latencies. We expect that tuning of the load reduces the pressure on the application and will result in improved latencies to  $2PL/2PC$  but at higher throughput.

The experiments reported on in this section are still relatively isolated. In order to claim generalized applicability, further work is needed to obtain results in different settings, and different kinds of loads. Related work [100] studies statically independent events, which is a subset of the independent actions discussed in this chapter. They show that at least 60% of event pairs in state machine models from industry can benefit from independent actions. To show PSAC's performance gains in real-life scenarios, orthogonal research is needed

to show that these independent actions occur in high-contention scenarios. For instance, it would be interesting to see how PSAC performs on some well-known benchmarks, such as TPC-C [87], the twitter-like Retwis Workload [71], YCSB [22], the SmallBank benchmark [19] and the OLTPBench benchmark runner [27]. Modeling TPC-C's NEWORDER is non-trivial in Rebel, because of a mismatch with SQL transactions, which can contain multiple queries and updates based on each other, where a Rebel event is non-interactive.

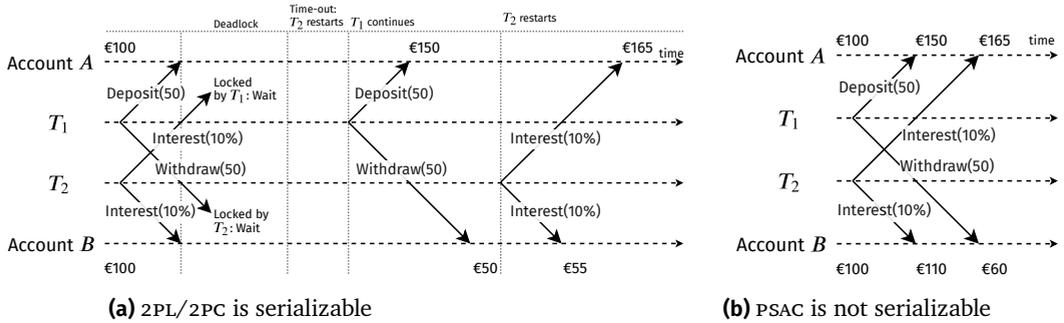
A geo-located setup, furthermore, would make the experiments more realistic, because round trip times to application nodes and database nodes are relatively large. We expect contention to be more of a problem there, because the latency of individual transactions (and thus the amount of locking) goes up. PSAC could be extended to employ techniques similar to Explicit Consistency [11] to allow parallel multi-regional actions without immediate communication.

### 2.6.2 Limitations

PSAC results in performance gains when actions are independent and there is much contention. So in practice the benefit depends on the use case, because it might not be a high-contention scenario. PSAC's benefit is most clear in the situation where (a) objects are involved in many (long running) synchronized actions from different other objects, making that single object a bottleneck for the others and (b) when all actions being used for synchronization are independent. This situation results in a scalability limiting factor where PSAC improves throughput and latency performance over 2PL/2PC. In the case with many objects which do not interact via synchronization or the request volume is low, PSAC's performance gain is limited, although never worse than 2PL/2PC in the same situation, as shown by the NoSync and Sync experiments.

A current limitation of PSAC is that it does not offer fairness for dependent actions. PSAC accepts new independent actions when there are also dependent actions in the wait queue. In a pathological scenario this results in a new in-progress action that keeps the queued action dependent, and thus will potentially never be removed from the queue. A potential solution is to consider the dependency of the queued actions on the incoming action, when determining independence, so that queued actions are never requeued indefinitely. Another, simpler but less fair, solution is to make sure only a limited number of independent actions can go before the dependent action.

## Chapter 2 Path-Sensitive Atomic Commit



**Figure 2.9** Example with two events, to show difference in isolation between 2PL/2PC and PSAC

**PSAC is not Serializable** PSAC does not guarantee serializability, while 2PL/2PC does. Consider the following situation, as shown in in figure 2.9: Two distributed transactions  $T_1$  and  $T_2$  are concurrently started.  $T_1$  consists of two actions, Deposit(50) and Withdraw(50) on respectively Account A and Account B, both with a balance of €100.  $T_2$  consists of two actions Interest(10%) on both Account A and Account B.  $T_1$  first arrives at Account A and  $T_2$  arrives first at Account B.

For strict 2PL (see figure 2.9a) this means that A is locked by  $T_1$  and B is locked by  $T_2$ . Now both transactions are waiting on the other Account to acquire locks. This deadlock situation is solved by a deadlock mechanism, such as timeouts: one of the two transactions times out, its lock is released and the other makes progress.

In this situation PSAC will allow both transactions to take a shared lock (see figure 2.9b), since for each transaction the already in-progress event's outcome does not influence the validity of the precondition of the other: Interest(10%) is valid on Account A, regardless of the commit or abort of Deposit(50). The same holds vice versa for Account B. Both transactions commit and have their effects applied. For Account A this results in applying the effects in order of arrival, first Deposit(50), then Interest(10%):  $(100 + 50) * 1.10 = 165$ . +For Account B first Interest(10%), then Withdraw(50):  $(100 * 1.10) - 50 = 60$ . Both accounts are in a valid state according to their specification, but notice that the transactions are applied in different order for the accounts. A first applies  $T_1$ , then  $T_2$ . B first applies  $T_2$ , then  $T_1$ .

For a valid serializable schedule for the whole system, in this case the two entities, the resulting state should be equivalent to an outcome state of a sequential execution of all transactions. Serializability requires one of two possible histories:  $\langle T_1, T_2 \rangle$  or  $\langle T_2, T_1 \rangle$ . The results of these histories are respectively:

$$\{A : 100, B : 100\} \xrightarrow{T_1} \{A : 150, B : 50\} \xrightarrow{T_2} \{A : 165, B : 55\} \text{ and}$$

$$\{A : 100, B : 100\} \xrightarrow{T_2} \{A : 110, B : 110\} \xrightarrow{T_1} \{A : 160, B : 60\}$$

The outcome for PSAC in this situation,  $\{A : 165, B : 60\}$ , which is not one of the valid serializable configurations. Ergo, this counterexample shows that PSAC is not serializable. Determining the isolation guarantees of PSAC more precisely is part of future work.

### 2.6.3 Evaluation

We have seen in the previous section that PSAC outperforms 2PL/2PC in throughput and its request latency is on par or better. This is due to less locking by PSAC, which is isolated by the NoSync experiment, where both PSAC and 2PL/2PC perform similarly when no transaction have to wait due to locking. However, PSAC does not give the same serializable isolation guarantee as 2PL/2PC. In our experiment scenarios with withdraw and deposit events this does not lead to different results or outcome states as in a serializable schedule, because these event's effects are commutative and result in serializable histories with PSAC.

## 2.7 Related work

**Distributed Transactions in Actor Systems** Orleans [83] is an actor based distributed application framework that implements transactions [29] in a similar way to 2PL/2PC, but with a central Transaction Manager, which decides if transactions are incompatible. To support high throughput the distributed object releases the 2PL lock when it prepares successfully, and already applies the new state. If a transaction triggers an abort, all the actions on top are also aborted (cascading abort). This solution enables high throughput, but it drops when aborts happen regularly on congested instances.

Reactors [88] is a distributed computing framework defined on reactors: actors reacting to events. It uses Reactor transactions with nested sub-transactions, but is not yet tested in a cluster of nodes. Their current implementation also uses 2PC in the transaction manager.

**Coordination** More recent work in distributed systems is investigating requirements to keep a program functionally correct, instead of focusing on data consistency (or memory consistency) where registers with single data items are always in a consistent state, which is what 2PL ensures. The CALM paper [50] hints at creating programs that are monotonic by construction, by using languages that help monotonic specification. PSAC makes sure objects only increase monotonically on the life-cycle lattice of an entity as defined by its specification. Parallel events are only allowed when the functional application properties (pre-/post-conditions) allow this. This makes sure that entities are monotonic by construction, w.r.t. their specification. This is a step towards CALM in the sense that it allows designers to write specifications with coordination, which in the end are run without local coordination by PSAC.

ROCOCO [77] reorders transactions at run time, whenever possible, instead of aborting. It uses offline detection, but only works on stored procedures. Coordination Avoidance [8, 10] focuses on lock-free algorithms in a geo-replicated setting. It makes sure that transactions do not conflict, and allows them on multiple geo-located data centers without coordination. They are eventually merged in an asynchronous fashion. Bailis [8] states: “Invariant Confluence captures a simple, informal rule: coordination can only be avoided if all local commit decisions are globally valid.” PSAC focuses on local avoidance of coordination of transactions on objects and it is yet to be seen how well it works in a geo-replicated setting.

PSAC is based on detecting independence of actions at run time. A compatible approach [100] to avoid blocking is to use static analysis of pre- and post-conditions to determine whether certain types of actions are always independent of other types of actions for all possible run-time states and action field values. Actions which never influence the outcome of later actions, such as depositing money in the running example, can always be safely started in parallel, without checking all possible outcomes of in-progress actions.

Using commutative operations to reduce coordination is a productive area [8, 10, 11, 38, 50, 78, 85, 118]. Commutative operations always result in the same outcome state, even when the operations are reordered. These works prevent coordination by relying on reordering and commutativity of operations in order to allow parallel operations in mainly geo-distributed data center environments.

**Other related work** The Escrow Method [80, 115] is a way to handle high-contention records for long running transactions. Balegas et al. [11] discuss

Escrow reservations with numeric fields divided over multiple (geo-located) nodes. Each node locally decides up to a maximum amount, and communicates with the rest when it needs more. In the banking example this is analogous to splitting the balance of an account in parts and allow nodes to locally mutate that part without synchronization. Although PSAC is not optimized for geo-separation, since an object is not divisible in multiple parts, it is not limited to numeric fields.

PSAC is related to Predicate locks [30, 42, 58] and Precision locks [42, 58], but differs in the sense that the latter operate at the level of tuples. PSAC supports more granular locking because two independent actions can change the same field or tuple.

Phase Reconciliation [78] is a run-time technique that splits high-contention objects over multiple CPU cores. It allows specific commutative operations of a single type to be processed locally on the core in parallel and after a configurable window the results are reconciled again. PSAC operations also cannot return values, however PSAC does not require commutative operations or all operations to be of a single type.

Flat Combining [51] is a technique to speed up concurrent access to data structures. The first thread to get the lock on a shared data structure, processes the operations of concurrent operations in a batch and informs the requester threads of their respective results, resulting in improved throughput. PSAC focuses on distributed transactions, where the actual transition is determined externally from the object by a transaction manager and not on applying operations sequentially as fast as possible.

## 2.8 Further Directions

In this chapter, we have presented PSAC informally. Further research is needed to obtain precise results about the isolation guarantees that PSAC offers. A potential direction could be to formally verify the correctness of PSAC, for instance, using  $TLA^+$  [69], or state-based formalization [23].

Further, the implementation of PSAC could be improved by applying well-known optimizations of 2PC. For instance, using half round trip time locks [8] the set of participants is forwarded by the previous participant to the next, in a linked list-like fashion. This results in half the round trip time for acquiring

## Chapter 2 Path-Sensitive Atomic Commit

the locks compared to the approach where locks are acquired one-by-one by the transaction coordinator.

Additional optimizations are possible in the representation of the outcome tree. For instance, outcomes could be grouped by abstractions, such as minimum or maximum values, sets of outcomes, or predicates deduced from pre- and post-conditions. This reduces the size of the tree, and thus faster precondition checking.

PSAC can be further improved by reordering of actions, however this requires commutative operations. At run time it can be checked if an incoming action is commutative with all in-progress actions, and safely reordered. However, the pre- and post-conditions should be explicit about time sensitive or otherwise important functional action ordering.

The depth of the possible outcomes tree is limited by configuration, because it grows exponentially in the number of in-progress actions. Benchmarks, not shown in this chapter, show that when it grows too big for the bank transfer use case, actions start timing out. This performance impact of varying this depth greatly depends on how computationally expensive actions are and how much contention there is, but also how many other resources are running and their contention. More in-progress actions, result in more running actors and extra calculations of the tree. It is future work to find an approach to tune this tree depth.

In order to evaluate the boundaries of applicability of PSAC, an extra experiment that tries to maximize the overhead of PSAC can be created. If computing preconditions or post states is expensive, the extra calculation overhead could result in worse performance than the sequential 2PL/2PC approach.

For online user experience keeping tail latencies low is important. We can apply the techniques presented in The Tail at Scale [25] to make PSAC more latency tail-tolerant. This requires sending multiple omnipotent requests to different application nodes and effectively increasing replication factors of the entities. The current design does not fit this yet, since the runtime makes sure only one instance of each entity is alive in the cluster. PSAC can be extended, however, to support read-only versions, inferring actions that are commutative to be applied on different nodes in arbitrary order order.

## 2.9 Conclusion

Large organizations such as banks require enterprise software with ever higher demands on consistency and availability, while at the same time controlling the complexity of large application landscapes. In this chapter, we have introduced path-sensitive atomic commit (PSAC), a novel concurrency mechanism that exploits domain knowledge from high-level specifications that describe the functionality of distributed objects or actors. PSAC avoids locking participants in a transaction by detecting whether requests sent to objects can be handled concurrently. Whether the effects of two or more requests are independent is established by analyzing the applicability and effects of message requests at run time.

PSAC has been implemented in the actor-based back-end of Rebel, a state machine-based DSL for describing business objects and their life cycle. Rebel specifications are mapped to actors running on top of the Akka framework. Using different code generators this allowed us to explicitly compare standard 2PL to PSAC as locking strategies for when objects need to synchronize in transactions. We conducted an empirical evaluation on an industry-inspired case of PSAC compared to an implementation based on standard two-phase commit with strict two-phase locking (2PL/2PC). We designed multiple experiments to show specifically where PSAC and 2PL/2PC perform similar and where PSAC outperforms 2PL/2PC.

Our results show that in low contention scenarios with and without synchronization the throughput is similar, because no actions can be parallelized. However, PSAC performs up to 1.8 times better than 2PL/2PC in terms of median throughput in high-contention scenarios. This is especially relevant, for instance, when a bank has to execute a large number of transactions on a single bank account. Latency-wise PSAC is on par or better than 2PL/2PC. Furthermore, PSAC scales as well as 2PL/2PC, and under specific non-uniform loads even better.



# 3

## Static Local Coordination Avoidance for Distributed Objects

**Abstract** In high-throughput, distributed systems, such as large-scale banking infrastructure, synchronization between actors becomes a bottle-neck in high-contention scenarios. This results in delays for users, and reduces opportunities for scaling such systems. This chapter proposes Static Local Coordination Avoidance, which analyzes application invariants at compile time to detect whether messages are independent, so that synchronization at run time is avoided, and parallelism is increased. Analysis shows that in industry scenarios up to 60% of operations are independent. Initial performance evaluation shows that, in comparison to a standard 2-phase commit baseline, throughput is increased, and latency is reduced. As a result, scalability bottlenecks in high-contention scenarios in distributed actor systems are reduced for independent messages.

### 3.1 Introduction

Enterprise software systems are large, complex, and hard to maintain. For instance, banks such as ING Bank, deal with large and complex IT landscapes consisting of many heterogeneous communicating applications, under high transaction loads. There is increased demand for high throughput and scalability for such systems.

---

This chapter is previously published as: Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Static local coordination avoidance for distributed objects”. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2019*. ACM Press, 2019, pages 21–30. ISBN: 9781450369824. DOI: 10.1145/3358499.3361222

In a distributed setting, both throughput and latency are heavily influenced by the amount of synchronization that is required between distributed objects. For instance, the generic atomic commit protocol Two-Phase Commit (2PC) [42] only allows a single event to be in progress per actor, all other events are queued. For high contention objects this leads to high latency and time-outs.

Recent work [10, 11] shows that invariants to maintain program-level consistency can be leveraged to optimize the implementation of synchronization. Invariant Confluence [10] shows that for the TPC-C benchmark [87] ten of twelve invariants are invariant confluent and require no coordination.

In this chapter we propose a similar, but novel concurrency mechanism, called Local Coordination Avoidance (LoCA), that allows multiple concurrent in-progress events per object, when it can be determined that such events are independent of each other. One event is independent of another if the commit or abort of the latter can never invalidate the result of the former. In that case, processing of the latter can be started without waiting.

As an example, consider the text book example of a bank account entity with withdraw and deposit events, where withdraw has a precondition that the balance should be sufficient for the withdrawal. In this case the deposit event is independent of deposit itself, because depositing money can never invalidate the requirements of a deposit. Deposit is also independent of withdraw, because depositing money is always possible, even if the in-progress withdraw would fail. A withdraw event, however, is not independent from withdraw, because the failure or success of in-progress withdraw might influence the precondition of the second withdraw.

LoCA is informed by static analysis of state machine models. In our case, we use Rebel [108], a domain specific language (DSL) to model financial products as state machines which communicate using atomic, synchronized events. LoCA consists of statically analyzing application invariants declared as pre- and post-conditions in the state machine models. This results in pairs of events that are independent, regardless of local state at run time.

We have implemented the independent event analysis by transforming Rebel state machine models to constraint definitions for the z3 Satisfiability Modulo Theory (SMT) solver [76], which computes the set of independent event pairs. This set is then input to the run-time system, which safely skips the precondition check if a new event comes in that is independent of all in-progress events,—otherwise it falls back to 2PC.

We have run the analysis on state machine models manually derived from the standard TPC-C [87] benchmark, and state machine models currently being

prototyped inside ING Bank. In both cases, the results show that around 60% of event combinations are independent, suggesting that the benefits of LoCA could be substantial. Initial performance evaluation shows that LoCA, or a variant of LoCA that detects independence at run time increases throughput and reduces latency compared to vanilla 2PC in high contention scenarios.

The contributions of this chapter are as follows:

- We formalize the notion of Statically Independent Events (*SIE*), a characterization of state machine models that captures when an event's preconditions are always independent of in-progress event's effects, and show how an SMT solver can be used to compute independent pairs from state machine models (section 3.2);
- We describe a novel run-time concurrency control mechanism, Local Coordination Avoidance (LoCA), which uses independent events to speed up synchronization on distributed objects. We present a LoCA implementation leveraging *SIE* analysis results: LoCA<sup>S</sup> (section 3.3);
- We evaluate the *SIE* analysis on two realistic examples: the TPC-C benchmark and Rebel specifications developed at ING. We evaluate the performance of both 2PC and LoCA variants, and show that LoCA<sup>S</sup> outperforms 2PC in high contention scenarios (section 3.4).

Source code of the *SIE* analysis and LoCA implementation, together with result data is found in [96].

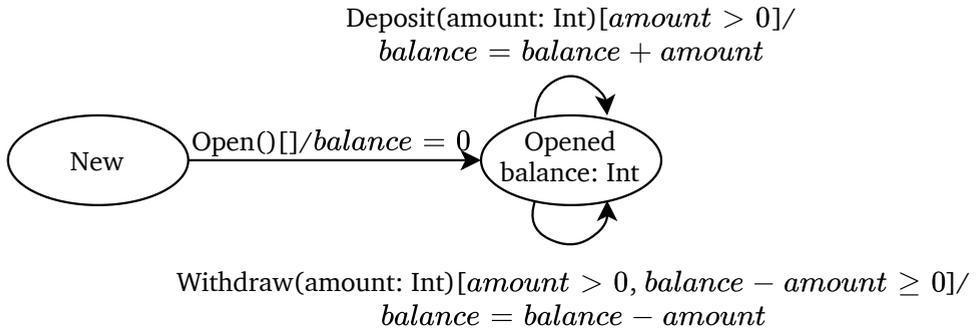
We conclude with a discussion and limitations (section 3.5); related work (section 3.6); further directions for research (section 3.7); and a conclusion (section 3.8).

## 3.2 Independent Events

### 3.2.1 Bank Account Example

Consider an example of a bank account state machine as seen in figure 3.1. For simplicity it has two states *New* and *Opened* with an integer data field *balance*, and three Events: *Open*, *Deposit* and *Withdraw*, the latter two with an integer amount event parameter, that should be a positive integer. The precondition on withdrawal ( $balance - amount \geq 0$ ) makes sure that the balance does not become negative. The effect of the events are represented by labels on the edges in state chart notation. The effect of *Open* sets the balance to 0 and the

### Chapter 3 Static Local Coordination Avoidance for Distributed Objects



**Figure 3.1** State machine of example simple account. Events are defined in state chart notation: Event(fields)[guard]/effect

state to Opened. The effect of Deposit and Withdraw, respectively increases or decreases the balance with the events' amount value.

Now, consider an instance of this bank account state machine running as an actor. The bank account actor handles a single event at one moment in time. A sequence of events is handled sequentially, one at a time. If an incoming event is part of a distributed 2PC-transaction with other actor participants, the actor first decides if the event is allowed by checking its preconditions, but it cannot transition to the next state yet. To maintain the serializability requirements of the distributed transaction, the actor has to wait on the whole transaction to either commit or abort the event. Other incoming events have to wait on the in-progress event.

In this particular example it becomes clear that waiting on in-progress events is not always necessary to ensure valid state machine transitions. For example, an incoming event Deposit(10)'s precondition check is independent of an in-progress Withdraw(20)'s effect in state Opened(100). It does not matter if the in-progress Withdraw(20)'s effects are actually applied or not. The state machine stays in the Opened state, the only difference is the balance, which is only decremented on commit of the Withdraw(20) event. The new incoming Deposit(10) can always already start, since the Opened state allows it and the specific balance does not invalidate the precondition of Deposit(10). The incoming Deposit(10) event is thus independent of the in-progress Withdraw(20) event.

### 3.2.2 Independent Events

An incoming event  $e_2$  is independent of an in-progress event  $e_1$ , iff  $e_2$  is accepted by the state machine, independent of whether  $e_1$ 's effects are actually applied or not.

In order to formalize the notion of independent event pairs, we consider a finite state machine, with states with data and events with parameters. We assume that the transition function is encoded in the preconditions  $pre : Event \times State \rightarrow Boolean$ . The predicate  $pre(e, s)$  is true iff event  $e$  is valid in the given state  $s$ . An event is valid in a state if the transition function and its transition guards, the preconditions, allow it. The resulting outcome state of a transition, given the current state and event is encoded in  $post : Event \times State \times State \rightarrow Boolean$ .  $post(e, s_{from}, s_{to})$  is true iff event  $e$  in state  $s_{from}$  leads to post state  $s_{to}$ .

Given an in-progress event  $e_1$  and an incoming event  $e_2$  in some starting state  $s$ , the independent event relation  $IE(e_1, e_2, s)$  is defined as follows:

$\forall s' \in State.$

$$pre(e_1, s) \wedge post(e_1, s, s') \rightarrow (pre(e_2, s) \leftrightarrow pre(e_2, s')) \quad (3.1)$$

$IE(e_1, e_2, s)$  denotes that  $e_2$ 's acceptance is independent of the outcome of  $e_1$  in state  $s$ . The predicate  $pre(e_1, s)$  makes sure that  $e_1$  has valid preconditions in  $s$ , describing the situation where  $e_1$  is already accepted by the participant, but has not yet committed nor aborted.  $post(e_1, s, s')$  binds  $s'$  to the post state when  $e_1$ 's effects are applied on  $s$ . An event  $e_2$  is independent of an event  $e_1$  iff, for all possible post states  $s'$ , the evaluation of the preconditions of  $e_2$  in both  $s$  and  $s'$  give the same result. Intuitively this means that whether  $e_1$  eventually commits or aborts and its effects are applied or not, does not influence the precondition check of  $e_2$ . There is no possible way for the result of  $e_1$  to influence the validity of  $e_2$ .

### 3.2.3 Statically Independent Events

The  $IE$  relation captures independence at run time: it considers preconditions and postconditions, given a current state machine state (e.g., balance). This however, requires run-time computation when dispatching incoming events, which could be expensive. Here we introduce statically independent events, which avoid this computation step.

Table 3.1 displays all statically independent event pairs of the bank account example. It shows the decision on event of type  $E_2$  given that an event of type

**Table 3.1** Static independency of bank account events.  $E_1$  in rows,  $E_2$  in columns.

$SIE(E_1, E_2)$	Open	Deposit	Withdraw
Open	DELAY	DELAY	REJECT
Deposit	REJECT	ACCEPT	DELAY
Withdraw	REJECT	ACCEPT	DELAY

$E_1$  is in progress. For instance, Deposit is statically independent of both Deposit and Withdraw, because no matter the actual run-time state of the bank account, deposits can always be directly accepted. Similarly, if a Deposit is in progress, an Open event should be immediately rejected, since the state machine’s transition function disallows it. For all event pairs that are not independent, the decision is delay.

$SIE$  is a relation between two types of events,  $E_1$  and  $E_2$ , without considering their parameter values or the run-time state of an actor.

$$SIE(E_1, E_2) = \forall s \in State, e_1 \in E_1, e_2 \in E_2. IE(e_1, e_2, s)$$

where  $\forall e_i \in E_i$  means for all possible event instances of type  $E_i$ .  $SIE(E_1, E_2)$  is true when all possible instances of the event types  $E_1$  and  $E_2$  are independent in all possible starting states  $s$ .

### 3.2.4 Computing $SIE$

The  $SIE$  definition can be used to transform state machine models to first-order logic formulas as input to SMT-solvers, like z3 [76] to find the statically independent event pairs at compile time. For this we assume a mapping of the state machine’s transition relation and the pre- and postconditions of each type of event as formulas in first-order logic.

In order to let an SMT-solver find the statically independent events, we let the solver search for counterexamples, the *dependent* event pairs. This means that for every combination of event types  $E_1$  and  $E_2$ , we ask the solver whether the formula  $\neg SIE(E_1, E_2)$  can be satisfied. If it is satisfiable, the resulting model represents a counterexample witnessing the fact the  $E_2$  is dependent on  $E_1$ , otherwise they are independent.

The negation of  $SIE(E_1, E_2)$ , after inlining the definition of  $IE$  is:

$$\begin{aligned} \exists s, s' \in State, e_1 \in E_1, e_2 \in E_2. \\ pre(e_1, s) \wedge post(e_1, s, s') \wedge \neg(pre(e_2, s) \leftrightarrow pre(e_2, s')) \end{aligned}$$

To satisfy this formula the solver needs to find a model instance with some starting state  $s$ , and two event instances  $e_1$  and  $e_2$ , for which it holds that event  $e_1$  is valid in  $s$  and its follow-up state is  $s'$ , but event  $e_2$  should be invalid in only one of states  $s$  or  $s'$ . Such a model instance denotes that  $e_2$  is *dependent* on  $e_1$ 's outcome. The resulting  $e_1$  and  $e_2$  event instances are the counterexample for the static independence of  $E_2$  on  $E_1$ , making event type  $E_2$  statically dependent on event type  $E_1$ .

Withdraw is not statically independent of Deposit and the first instance found by z3 is indeed an example of this: Withdraw(35) is only allowed when the Deposit(1202) actually commits in state Opened(34), otherwise the balance would not be sufficient. Another example is where Open is dependent on the outcome of another Open in state New, which makes sense since an account can only be opened once.

However, checking  $\neg SIE(\text{Withdraw}, \text{Deposit})$  will return “unsat” which means that no counterexample could be found to show that the outcome of Withdraw would influence the acceptance of Deposit.

#### 3.2.5 Always Accept or Always Reject?

The  $SIE$  relation determines whether one event is independent of the other, but does not say if an event should always be accepted or always be rejected, as shown in table 3.1.

To obtain this information we partition the definition of  $SIE$  in two variants  $SIE^{Accept}$  and  $SIE^{Reject}$ , where the equivalence used in  $IE$  (Definition 3.1) is split in the case where the preconditions of both events are true, and the case where neither of them are true:

$$\begin{aligned} SIE^{Accept}(E_1, E_2) = \forall s, s' \in State, e_1 \in E_1, e_2 \in E_2. \\ pre(e_1, s) \wedge post(e_1, s, s') \rightarrow (pre(e_2, s) \wedge pre(e_2, s')) \end{aligned}$$

$$\begin{aligned} SIE^{Reject}(E_1, E_2) = \forall s, s' \in State, e_1 \in E_1, e_2 \in E_2. \\ pre(e_1, s) \wedge post(e_1, s, s') \rightarrow \neg(pre(e_2, s) \vee pre(e_2, s')) \end{aligned}$$

To ensure that the solver does not return “junk” models in which events are always invalid, regardless of the state (e.g.,  $\text{Withdraw}(-1000)$ ), we instruct the solver to only consider such events by asserting  $\exists s \in \text{State.pre}(e_2, s)$ . Following the same process as with *SIE* above,  $\text{SIE}^{\text{Accept}}$  and  $\text{SIE}^{\text{Reject}}$  can be used to find statically independent event pairs, knowing whether the decision should be accept or reject.

### 3.3 Local Coordination Avoidance (LoCA)

A run-time system for a state machine-based language like Rebel is typically implemented as follows. A distributed object receives a request as part of a 2PC distributed transaction. It then becomes participant in this transaction. If the request is valid according to the corresponding event’s preconditions, the object is locked until the transaction completes. If the preconditions do not hold, the object declines the request immediately and does not have to lock.

In this case, for each incoming request, the participant object has to check the preconditions and act accordingly. If the request is valid, it votes to commit the transaction and the object is locked for other requests in order to maintain the consistency guarantees of 2PC. Even though the participant object has voted to commit, it cannot continue until the coordinator responds, since it does not know if other transaction participants have voted to abort. Incoming transactions are delayed in order of arrival until the transaction coordinator commits or aborts the transaction. This results in potentially high wait times and thus high transaction latency for busy objects.

Local Coordination Avoidance (LoCA) is our novel concurrency control mechanism that leverages *SIE* information at run-time to run multiple parallel 2PC requests per participant object. *IE* allows an implementation to safely start processing new events when previous events are still in progress. LoCA first checks independence according to the pre-computed results of the static independent event analysis (*SIE*). If two types of events are not statically independent, the actual event occurrences can still be dynamically independent; this is checked according to the *IE* relation at run time. If events are dynamically dependent still, LoCA falls back to vanilla 2PC.

For evaluation purposes, we distinguish variants of LoCA according to which kinds of independence checking are done at run time:  $\text{LoCA}^S$  (only checks based on *SIE*),  $\text{LoCA}^D$  (only checks based on *IE* [IO3]), and  $\text{LoCA}^{SD}$  (first *SIE*, then *IE*).

### 3.3.1 Static LoCA

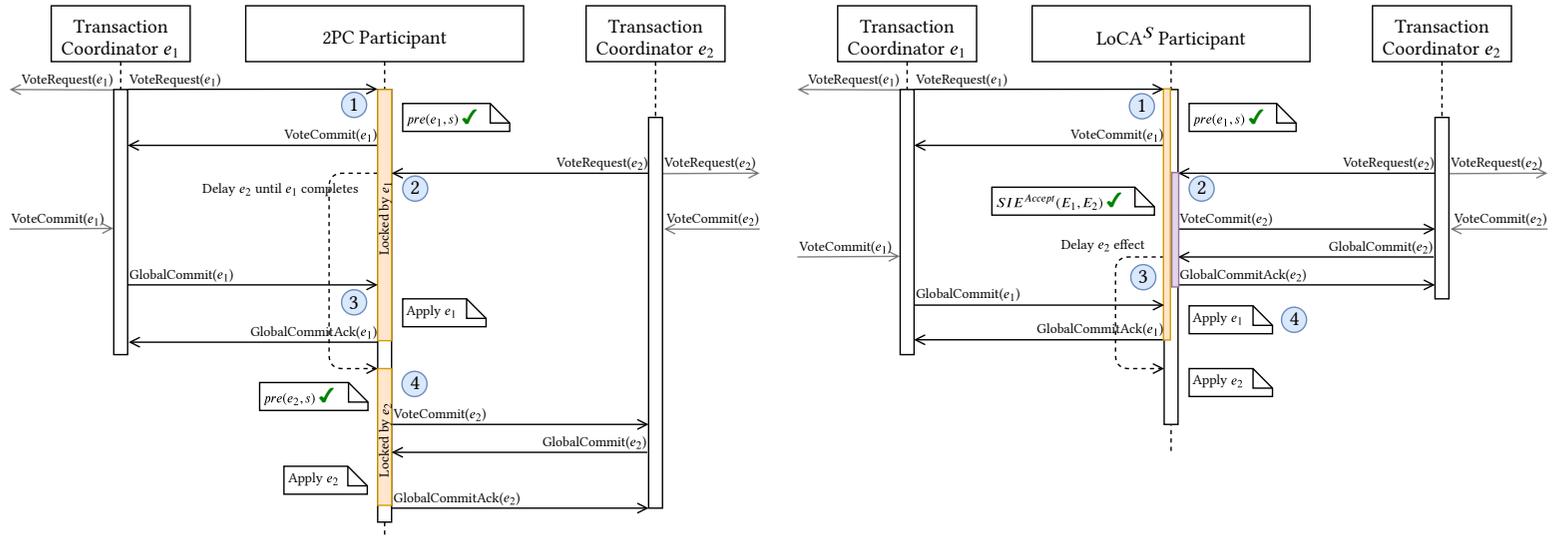
Static LoCA ( $LoCA^S$ ) first considers  $SIE^{Accept}$ , then  $SIE^{Reject}$ , and falls back to 2PC.

$LoCA^S$  leverages the  $SIE$  analysis results. In the case where the transaction participant is waiting on a response of an in-progress transaction's coordinator, it can use the  $SIE$  independent event pairs to determine if it is always safe to start another incoming request in parallel.

If the incoming event's type is an independent accept for all the in-progress events' type, as determined by  $SIE^{Accept}$ , it can be started immediately without checking its preconditions. If not matched by  $SIE^{Accept}$ , and the incoming event is an independent reject, determined by  $SIE^{Reject}$ , for all in-progress events, it can be immediately rejected. If the request is not statically independent for both sets, the incoming event is dependent on at least one of the in-progress events' type and has to be delayed until it is finished.

Figure 3.2 shows sequence diagrams to compare vanilla 2PC to  $LoCA^S$  in the case that  $LoCA^S$  can directly accept an event. For 2PC an action is delayed when another action is in progress. For  $LoCA^S$  the action's transaction is started when  $SIE^{Accept}$  allows it. In figure 3.2a, ① a 2PC-participant receives a  $VOTEREQUEST(e_1)$  message, and responds with  $VOTECOMMIT(e_1)$  because the preconditions  $pre(e_1, s)$  allow it. This locks the resource until  $e_1$  commits or aborts. When  $e_1$  is still in progress, the participant receives ② a  $VOTEREQUEST$  for another event  $e_2$ , and it is delayed until  $e_1$  completes. On receiving of  $GLOBALCOMMIT(e_1)$  ③, the effects of  $e_1$  are applied, the state is updated and acknowledgement is replied. Now the delayed  $e_2$  is started ④, and a  $VOTECOMMIT(e_2)$  is send.  $e_2$  is eventually committed, and its effects applied.

In figure 3.2b, a  $LoCA^S$ -participant receives ① a  $VOTEREQUEST(e_1)$  message, and similarly accepts the event because the precondition  $pre(e_1, s)$  holds. Unlike 2PC the resource is not locked, but guarded by static independence guarantees. When  $VOTEREQUEST(e_2)$  arrives ②, it now checks if it is safe to execute  $e_2$  in parallel with  $e_1$ , by checking the event's types in  $SIE^{Accept}(E_1, E_2)$ . In this case  $SIE^{Accept}(E_1, E_2)$  holds and  $VOTECOMMIT(e_2)$  is readily sent. Now  $e_2$  commits earlier ③, and in order to maintain serializability, the effects of  $e_2$  are delayed to preserve the original order. When  $e_1$  is allowed to commit ④, its effects are applied, and the postponed effects of  $e_2$  as well. The case for immediate reject is analogous.



(a) Vanilla Two-Phase Commit

(b) Static Local Coordination Avoidance

Figure 3.2 2PC and LoCA<sup>S</sup>, Direct Accept.

## 3.4 Evaluation

In this section we evaluate static local coordination avoidance to answer two questions:

**RQ 1.** How often are events independent in realistic scenarios?

**RQ 2.** What is the effect of LoCA on performance in terms of throughput and latency?

### 3.4.1 Independence in Realistic Scenarios (RQ 1)

In order to evaluate the relevance of *SIE* analysis we have analyzed two sets of Rebel models and computed the independence results. The first set consists of Rebel state machine models manually derived from the standard TPC-C benchmark [87], the second set consists of models of payment infrastructure developed at ING Bank.

**TPC-C** There is no direct mapping for TPC-C's transactions to Rebel, but we can model the tables and the transaction's operations on them. TPC-C consist of 9 database tables and 5 transactions, which we model as state machines where transactions are represented as events. Some TPC-C transactions make decisions based on data which is read from the state machine. To avoid that in-progress events modify such data, we model this data flow using event parameters.

While this approach makes sure that exposed values are not changed by in-progress events, it is an over-approximation and leads to false negatives of dependent event pairs. Incorrectly detected dependent event pairs cannot be parallelized at run time, which is still correct, but not as efficient as when correctly identified.

Table 3.2 shows the *SIE* analyses' results for each table specification and a percentage describing the ratio between independent and all event pairs. As can be seen, many events are independent for this case. This is the case because often the specific tables' data is only read, and not used for decisions later in the transaction. Note that these results are only for local independency decisions in a state machine instance, representing a single row in the database tables.

**ING Bank Account Models** In order to evaluate *SIE* effectiveness on an industrial use case, we run the analysis on Rebel specifications being developed at ING

**Table 3.2** TPC-C *SIE* Analyses

TPC-C Table	#States / #Events	#Direct Accept / #Direct Reject	Independence Ratio
Stock	1 / 2	4 / 0	100%
NewOrder	3 / 3	2 / 2	44%
Order	2 / 2	4 / 0	100%
District	1 / 3	6 / 0	67%
Customer	1 / 4	12 / 0	75%
OrderLine	2 / 4	6 / 3	56%
Warehouse	1 / 1	1 / 0	100%
History	2 / 1	0 / 0	0%
Item	2 / 2	0 / 1	25%
Total	15 / 22	35 / 6	64%

Bank, containing multiple types of bank accounts and Single Euro Payments Area (SEPA) bank transactions. Our specification data set consists of 29 Rebel specifications. 7 of them used features not yet supported by our analysis tool. The remaining 22 specifications are analyzed with small changes. For some of them the data types and preconditions are simplified, in such a way that it does not influence the *SIE* analysis, for instance changing DateTime fields to Integer fields and mapping static set membership tests to string equality.

The analysis results of the resulting 22 specifications are presented in table 3.3. Most specifications are relatively small in terms of number of states and events. Many independent events pairs are direct reject, since many events are not allowed in multiple states. More than 60% of all event pairs are independent, suggesting that *SIE* analysis would be beneficial in industrial scenarios. This analysis shows how often events are independent in realistic scenarios answering RQ 1.

### 3.4.2 Throughput and Latency (RQ 2)

LoCA<sup>S</sup> is expected to show performance benefits when a transaction participant is involved in multiple transactions at the same moment in time and for independent events. In low-contention scenario we expect little extra performance in using LoCA<sup>S</sup> compared to 2PC.

**Table 3.3** SIE Analyses' results of ING product specifications

Specification	#States / #Events	#Direct Accept / #Direct Reject	Independence Ratio
CreditTransfer	9 / 9	1 / 52	65%
Restriction	3 / 4	6 / 3	56%
DepositBlock	3 / 4	2 / 5	44%
DirectDebitBlock	3 / 4	2 / 5	44%
WithdrawBlock	3 / 4	2 / 5	44%
Limit	5 / 5	1 / 12	52%
NoLimit	3 / 3	1 / 2	33%
RevolvingAccount	2 / 3	4 / 2	67%
DirectDebitAccount	2 / 3	4 / 2	67%
TreasuryAccount	4 / 4	4 / 8	75%
CreditTransferBatch	5 / 6	10 / 5	42%
CreditBooking	4 / 3	0 / 2	22%
DebitCreditorBooking	3 / 2	0 / 1	25%
FromExternalDebitBooking	3 / 2	0 / 1	25%
ToExternalDebitBooking	3 / 2	0 / 1	25%
DebitBooking	13 / 17	0 / 188	65%
CurrentAccount	3 / 4	4 / 4	50%
LocalCreditTransfer	6 / 5	0 / 12	48%
SepaCreditTransfer	6 / 8	15 / 29	69%
Arrangement	4 / 4	2 / 7	56%
BankPayment	4 / 4	0 / 6	38%
ThirdPartyPayment	5 / 3	0 / 5	56%
Total	96 / 103	58 / 357	(mean) 61%

### Chapter 3 Static Local Coordination Avoidance for Distributed Objects

The goal of the performance evaluation is to find out if this expectation holds. We ran several synthetic scenarios in microbenchmarks in order to confirm these expectations.

In order to evaluate `LoCA` we prototyped a small accounting service providing dependent and independent events in the state machine DSL Rebel [108]. The *SIE* Analysis translates Rebel specifications to SMT using Rascal [65] and runs the analysis using the state-of-the-art SMT-solver z3 [76]. The *SIE* results are used in a `LoCA` implementation [96], an actor-based runtime system, based on the Akka actor toolkit [3], on the JVM.

Akka enables fault tolerance and horizontal scalability by sharding actors over multiple servers and provides locational transparent message passing between them. These features together with persistence and state machine primitives, are used to implement `LoCA` and `2PC`. The Rebel state machine models are translated to communicating run-time actors.

The `2PC` implementation follows the description by Tanenbaum and Van Steen [110]. In order to avoid deadlocks, we make sure that all transactions participants are locked in increasing order.

For all experiments in this chapter, we limit the maximum number of parallel events per actor for `LoCA` to a configurable limit of 8. A higher number results in reduced throughput and worse latency when contention increases. This is a problem, especially for `LoCAD` and `LoCASD` with run-time dependent events, when the computation time grows exponentially due to more concurrently in-progress actions.

We present multiple microbenchmarks and their throughput and latency results on a single application node. Each benchmark is run using the different synchronization implementation variants, `2PC`, `LoCAS`, `LoCAD`, `LoCASD`, and increasing contention rate. The benchmark scenarios are the following:

- Statically dependent events – Withdraws on single account
- Statically independent events – Deposits on single account
- Distributed transactions with statically dependent and statically independent event – Money transfers between two accounts
- Distributed transactions with high-contention statically independent events and low-contention dependent events – Tax direct debit use case: Deposits on a single tax account & Withdraws on 10 000 taxed accounts

The first two benchmarks represent statically dependent and independent events. These are baseline benchmarks to determine whether `LoCAS` improves

throughput when contention increases for independent events, but has to fall back to 2PC for dependent events.

The distributed transaction cases are more realistic. In the money transfer case between two accounts, the expectation is that  $\text{LoCA}^S$  improves performance only slightly over 2PC, because the whole distributed transaction has to wait on the slowest dependent event participant. On the other hand for  $\text{LoCA}^D$  and  $\text{LoCA}^{SD}$  we expect better performance, since the Withdraws are run-time independent, because enough balance is available for multiple parallel Withdrawals.

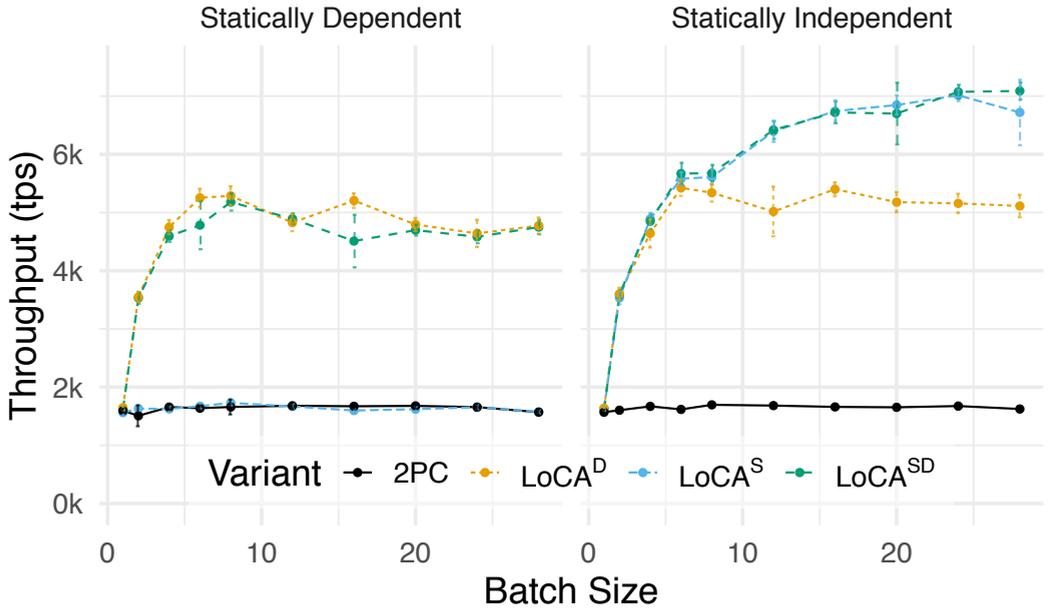
For the tax use case, the expectation is that  $\text{LoCA}^S$  performs better than 2PC and is on par with  $\text{LoCA}^D$  and  $\text{LoCA}^{SD}$ , because the statically independent Deposits on the tax account can be parallelized.

The microbenchmarks are run using JMH [57] and measure the maximum throughput in transactions per second and transaction latency. The hardware used is a dual core Intel i7-7567U 3.5 GHz up to 4 GHz with 32 GiB of ram on Linux using Java AdoptOpenJDK HotSpot 11.0.2+9 64bit. Each run consists of 5 warmup cycles and 20 measure cycles of each 10 seconds.

To increase contention, multiple parallel events are requested in batches. The batch size is varied in order to find out when contention becomes a problem and determines the maximum number of events in-progress on a participant.

**Microbenchmark results** The throughput results for the statically dependent and independent events are shown in figure 3.3. For statically dependent events, in low-contention scenarios (Batch Size = 1), all variants reach 1800 transactions per second. As expected for higher contention (Batch Size > 1),  $\text{LoCA}^S$  performs the same as 2PC, because both algorithms only allow a single event to be in progress for statically dependent events. Since  $\text{LoCA}^{SD}$  falls back to  $\text{LoCA}^D$  for run-time independent events there is a higher maximum throughput around 5000 transactions per second. At run time  $\text{LoCA}^D$  determines that the Withdraw events can be safely run concurrently because enough balance is available.

For statically independent events, 2PC has the same maximum throughput as the statically dependent variant, because it handles all events sequentially.  $\text{LoCA}^S$  reaches higher maximum throughput than the dependent variant, around 7000, because for independent events, it can immediately accept.  $\text{LoCA}^{SD}$  has similar performance, because for independent events it is equivalent to  $\text{LoCA}^S$  and does not have to fall back to  $\text{LoCA}^D$ .  $\text{LoCA}^D$  also detects the independence



**Figure 3.3** Statically Dependent and Independent events' throughput. Higher is better.

at run time, but suffers a computational overhead compared to  $LoCA^S$ , resulting in a lower maximum throughput, but still performs better than 2PC.

Interestingly,  $LoCA^{SD}$  performs the best in both cases. It leverages the static knowledge when events are independent so no precondition calculations are necessary, and in the dependent case it can profit from  $IE$ 's dynamic independence check.

**Distributed transaction: Money transfer** Figure 3.4 shows the throughput results for the money transfer microbenchmark. As expected, all variants perform the same in the low-contention case (Batch Size = 1). 2PC and  $LoCA^S$  have similar maximum throughput at 1000 transactions per second. This is expected because, although the Deposit is statically independent, the whole transaction has to wait on the statically dependent Withdraw, limiting the overall throughput. Both  $LoCA^D$  and  $LoCA^{SD}$  perform better at around 2500 transactions per second, which can be explained by the dynamic independence of Withdraw.

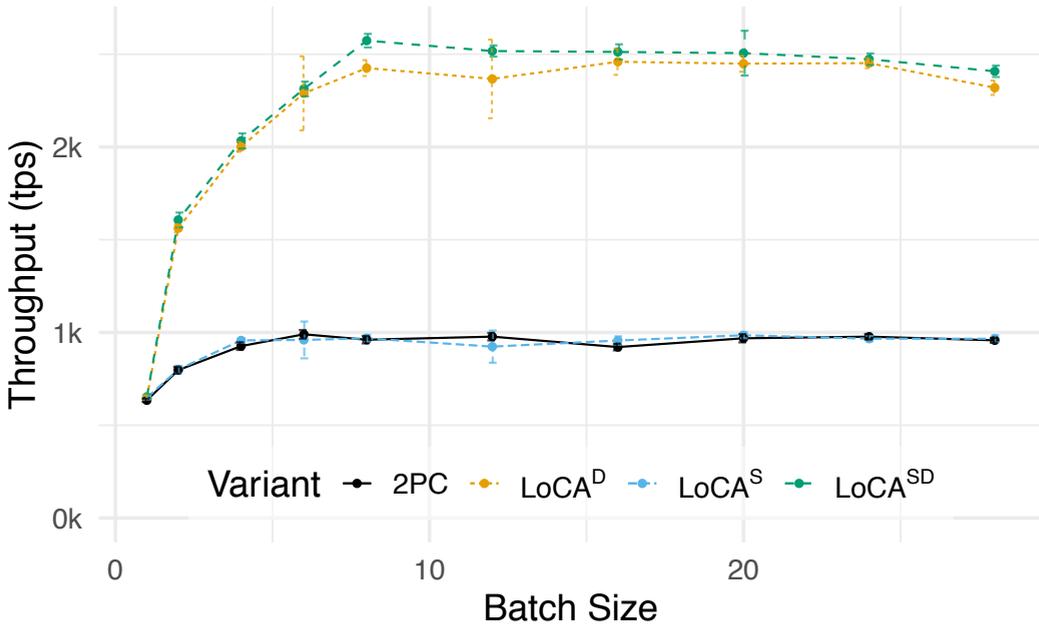


Figure 3.4 Throughput of the money transfer microbenchmarks

**Distributed transaction: Tax collection** For the tax collection, the throughput results shown in figure 3.5 are as expected. 2PC performs up to 1750 transactions per second and for higher contentions slowly drops, which is explained by larger batch size having to wait longer for the sequential handling by the tax account.  $\text{LoCA}^S$  and  $\text{LoCA}^{SD}$  perform similar, up to 3000 tps, since throughput is limited by the tax account.  $\text{LoCA}^D$  performs slightly worse because of the computational overhead.

**Latency Results** For all microbenchmarks, latency results were also collected [96]. Overall, the latency percentiles follow the same curve for all algorithm variants, where higher throughput corresponds to lower latency per operation.

Overall, all  $\text{LoCA}$  variants outperform 2PC, both in throughput and latency, except for the low-contention case, where it performs similar. We thus answer RQ 2 on the effect of  $\text{LoCA}$  and variants on performance.

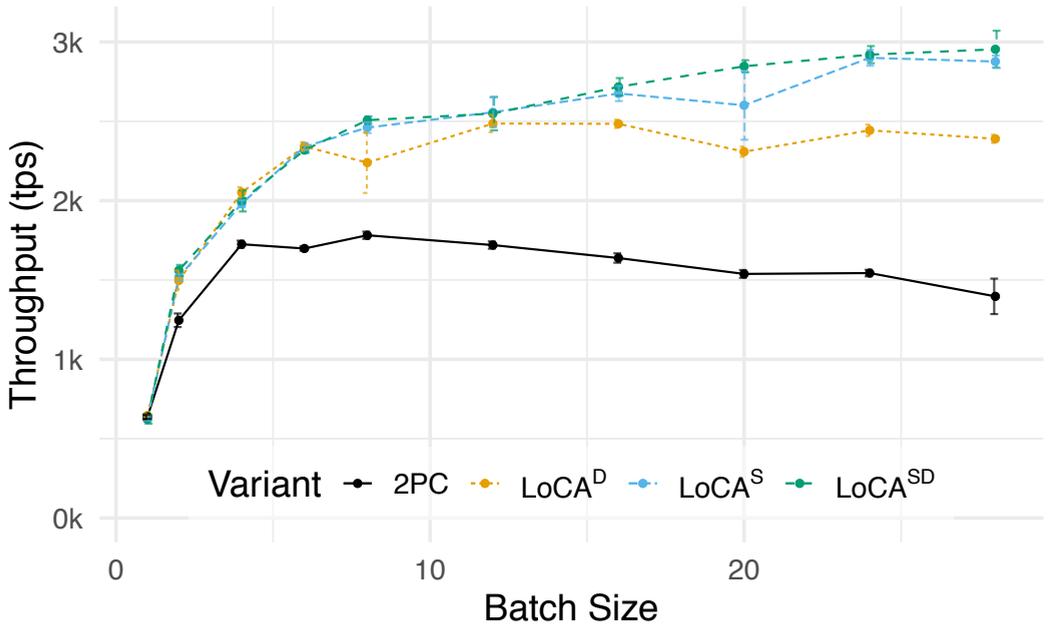


Figure 3.5 Throughput of the tax microbenchmarks

### 3.5 Discussion

The performance evaluation has shown that *SIE* analysis may increase performance both in throughput and latency, for situations where multiple requests arrive at objects in a small amount of time. The statically independent events make sure that only the event types have to be inspected.

In distributed transactions with statically independent events,  $LoCA^{SD}$  and  $LoCA^S$  perform better than  $LoCA^D$  since  $LoCA^D$  is computationally more expensive when the number of parallel events increases. However, in scenarios where the independence can only be determined dynamically, the combined version  $LoCA^{SD}$  still outperforms 2PC. It turns out that all  $LoCA$  variants,  $LoCA^S$ ,  $LoCA^D$  and  $LoCA^{SD}$ , perform as least as well as 2PC, and can therefore be used as a replacement in all cases where pre- and postconditions are known. The *SIE* analysis finds the independent events pairs in order of seconds per state machine model.

*SIE* analysis is applicable in the specific scope where requesters of operations are only interested in the success or failure of the operation. The requester

receives the acknowledgement directly, but not yet the post state of an entity, when this depends on other in-progress events. The new state can be queried in a new transaction.

*IE* is an asymmetric relation, and also does not require events to be commutative. Even though multiple events can be in progress at the same time, their effects always are applied in order of arrival, leading to serializable behavior and commutativity is not necessary. In our specific implementation unconditional acceptance of events is already communicated to the requester, but not the outcome state.

**Limitations** *LoCA* results in performance gains in the specific scenario of independent actions and high-contention. *SIE* can find independent events, but from this information it is not directly clear if this independent result allows *LoCA* to speed up performance in practice, since it might not be a high-contention scenario. In practice the events could very well be low-volume or the requests are already spread out on many different specification instances.

*LoCA*'s benefit is most visible when the following conditions hold:

- objects are involved in many synchronization steps from different objects, each with low load, but making the single objects a bottleneck for all
- objects are involved in a single type of synchronization step from the same objects

Both cases are a scalability limiting factor when request volumes continue to increase and are typical for a bank like ING Bank. If the machines are low volume and the instances spread out, *LoCA*'s performance gain is limited, although it will never worsen the performance compared to 2PC.

## 3.6 Related Work

*IE* and *LoCA* focus on optimizing performance of a single object, running in a single location, which is highly-contended. It uses conventional actor architecture approaches to shard actors over multiple machines. Much literature focuses on how one can do parallel updates in multiple geo-distributed locations without communication overhead, and only synchronize if really necessary for application consistency. Running multiple instances of the same object in order to allow parallel operations, improves performance in high-contention

scenarios as well. *SIE* analysis is related to checking which operations are commutative and violate invariants.

Many distributed databases focus on scaling by splitting data into partitions, such as Cassandra [20], H-Store/VoltDB [109], Spanner [4]. Within partitions they fall back to sequential operations, such as 2PC and Optimistic Concurrency Control. *LoCA* focuses on avoiding coordination locally in these partitions and could thus be implemented inside other database systems to speed up these sequential operations, when program invariants are known.

**Program-level consistency** The notion of program-level consistency, instead of generic data-consistency, is a valid way to capture what a program should functionally do and also gives opportunity to improve performance while maintaining the program invariants. Work in this direction tries to find ways to characterize this notion, which in turn enables optimized implementations.

The *CALM* theorem [50] says that monotonic programs do not need coordination. So ideally programs should have only monotonic parts. *IE* and *LoCA* describe on a local object level, how one can execute events in parallel, improving performance, and make sure the object only grows monotonically in its lifecycle, by exploiting the programs functional requirements.

Coordination Avoidance [8] states that coordination can be avoided if all local commit decisions are globally valid. *IE* describes local avoidance of coordination between events on the object.

Explicit Consistency [11] also uses an *SMT*-solving approach to “identify which operations would be unsafe under concurrent execution”. For unsafe operations, it presents approaches on changes to make concurrent execution safe or requires an explicit synchronization implementation. It focuses on parallel changes on geo-located data centers.

Observable Atomic Consistency [118], related to RedBlue Consistency[72], categorizes operations in two categories: Commutative operations on *CRDTs* which can be handled in any order by different replicas, and totally ordered operations, for which the replicas need to coordinate.

*SIE* and *LoCA* operate in a different design space, where all operations on an object go through a single actor, which is not designed for a geo-distributed setting. It would be interesting to explore this space and an extension of *LoCA*.

Conflict-free Replicated Data Types [39, 93] guarantee Strong Eventual Consistency. This means that all replicas converge to the same state if they receive the same messages, not necessarily in the same order. *IE* provides strong

consistency, meaning serializability, since events are processed in the original order, but are internally processed concurrently.

Adahbi<sup>1</sup> [89] reasons that in many cases the high-contention bottleneck in 2PC can be avoided by making the precondition check of another participant a local decision, for example by querying the data required for its precondition check from the other participant. In that way data flows only one way, and both participants can locally decide, sometimes with data from the other, if the transactions will commit or abort without waiting on each other. *IE* focuses on local avoiding of delays, but still uses 2PC for the coordination of the transaction.

**Single node optimizations** Phase Reconciliation [78] is a run-time technique, that splits contended objects over multiple cores, and allows multiple commutative operations of the same type in parallel on each core. After a configurable window, the split versions are recombined in a reconciliation phase maintaining serializability. This improves throughput for contended objects. Similarly to *LOCA*, it thus allows safe parallel operations on an object, and the operations should not return values. Differences are that the operations are limited to commutative operations and only allow a single operation type per split phase. *LOCA* and *IE* do not require commutativity for operations and allow different operation types to run in parallel, as long as they are independent, which is either detected statically or dynamically, without special effort by the specification designer. Phase Reconciliation's implementation does not support durable writes yet, which *LOCA* explicitly supports. A difference in scope is that *LOCA* applies effects in the original order. Its parallelism has nothing to do with the kind of effects, but whether it influences preconditions of other events. Phase Reconciliation splits up the effects over threads, which *LOCA* does not do. Phase Reconciliation could be embedded within *LOCA*, to speed up the applying of effects within a *LOCA* object.

Flat Combining [51] and a distributed version of it [54] show an interesting way to speed up concurrent access to data by keeping track of concurrent operations on an object, and letting the first thread obtain the lock, batch process all the operations and notify the requesters with the result of their operation. Flat Combining focuses on reads and writes on data structures, and thus focuses on effects. *LOCA* differs in the sense that it is focused on distributed transactions, where it is externally decided by other transaction participants

---

<sup>1</sup> <https://dbmsmusings.blogspot.com/2019/01/its-time-to-move-on-from-two-phase.html>

if in-progress operations will be commit or aborted. An interesting part of Flat Combining is the canceling out of sequential operations, locally in the concurrent operations list, e.g. push and pops in a stack. This results in reduced sending of operations to the actual data structure. Interesting future work related to this, is to statically detect, using analyses similar to *SIE*, which events can safely be combined, and (partly) cancel each other out within invariant bounds.

### 3.7 Future Work

This chapter describes Independent Events pairs on event type level, ignoring parameters. It would be interesting to have a more granular approach by including symbolic field values. The *SMT*-solver can synthesize computationally cheap field bound checks that *LOCA* can use at run time to determine when events are independent.

In order to make *IE* more generally applicable, reading of data can be added. Now, static independence is determined by event types, resulting in either accept or reject. *IE* does not support state or return values for events. This has to be simulated as seen in the *TPC-C* use case. If exposed (computed) values are specified, this can be taken into account for the analysis. The “exposed” effect describes which (computed) values are exposed by events next to the postconditions, which only describe internal state changes. This would result in support for *SQL* like transactions, which can use sub queries and can represent *TPC-C*’s usage of data, resulting in fewer false-negative dependent events. This can also support analysis of nested synchronization in *Rebel*, by tracking if nested participants rely on exposed event parameter values.

*IE* focuses on parallel distributed transactions. It would be interesting to explore if the *IE* property can be exploited in other non-transactional cases, for example in the context of Active Objects [52].

The current *SIE* analysis can take false-positive dependent event pairs into account, because the *SMT*-solver is allowed to synthesize any state possible, also states that cannot occur in the normal state machine life cycle. Extra assertions could be added to make sure only reachable states are used. A drawback could be that the state machine representations becomes more involved and solve times can become higher.

In order to avoid deadlocks, transaction participants are locked in increasing order. In many cases `LoCA` does not need this, because it allows multiple transactions in parallel. `LoCA` should only need increasing order locking when deadlocks can happen for dependent events. Static analysis could detect this and switch to parallel requesting of locks, saving multiple round trip times in transaction latency, compared to increasing order locking.

This chapter presents microbenchmark performance evaluations on a single application node. Since the implementation is based on actors, which can run as-is in a clustered environment, it would be interesting to do further scalability performance evaluation on a cloud environment. It would be interesting to replicate `ING`'s workloads using `LoCA` to find a benefit compared to their current implementation.

Commutativity of statically independent events can most probably be statically determined. This would allow reordering of events at run time and would allow requesters to see outcome states earlier. Reordering would require designers of specifications to take care with pre- and postconditions to make sure that time sensitive or otherwise important event orders are captured explicitly.

Offline analysis using `SMT` solvers can also be used to support specification designers by giving insight in potential performance bottlenecks at design time. Research directions include detection of events which are used in multiple synchronized steps but never independent, and suggestions on how to make events independent. The latter can be done by systematically removing preconditions from dependent event pairs, until it becomes independent. This signals which preconditions might be weakened by the specification designer to reduce performance bottlenecks.

`LoCA` uses an atomic commitment protocol to implement the actual transaction, which is now `2PC`. Optimistic concurrency control, instead of `2PC`, could provide even more performance improvements, since fewer rollbacks or aborts would be required for independent events.

## 3.8 Conclusion

Atomic commitment protocols such as Two-Phase Commit (`2PC`) may lead to bottlenecks for high-contention objects, because requests have to wait on previous events to finish. It is possible to improve throughput and latency, by

### Chapter 3 Static Local Coordination Avoidance for Distributed Objects

increasing parallelism of events on an object, while maintaining application consistency.

Independent Event (*IE*) pairs capture when a state machine object can safely start processing events when other events are still in progress. Statically Independent Events (*SIE*) analysis enables detection of types of event pairs that are always independent, at compile-time. We have implemented the *SIE* analysis on top of the Rebel state machine DSL by translating object invariants to SMT constraints and checking the *SIE* property. Local Coordination Avoidance (*LoCA*) leverages the resulting independence information to start more events per object concurrently, when it is determined that new events cannot violate the object's invariants.

We have shown that in two sets of realistic Rebel specifications, around 60% of events are always independent, which suggests that *LoCA* potentially increases throughput in distributed systems. Preliminary performance evaluation shows that, compared to 2PC, *LoCA* performs at least similar to 2PC, but *LoCA* does increase throughput and reduce latency in high-contention scenarios with independent events.

# 4

## Automated Validation of State-Based Client-Centric Isolation with TLA<sup>+</sup>

**Abstract** Clear consistency guarantees on data are paramount for the design and implementation of distributed systems. When implementing distributed applications, developers require approaches to verify the data consistency guarantees of an implementation choice. Crooks *et al.* define a state-based and client-centric model of database isolation. This chapter formalizes this state-based model in TLA<sup>+</sup>, reproduces their examples and shows how to model check runtime traces and algorithms with this formalization. The formalized model in TLA<sup>+</sup> enables semi-automatic model checking for different implementation alternatives for transactional operations and allows checking of conformance to isolation levels. We reproduce examples of the original paper and confirm the isolation guarantees of the combination of the well-known 2-phase locking and 2-phase commit algorithms. Using model checking this formalization can also help finding bugs in incorrect specifications. This improves feasibility of automated checking of isolation guarantees in synthesized synchronization implementations and it provides an environment for experimenting with new designs.

---

This chapter is previously published as: Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Automated Validation of State-Based Client-Centric Isolation with TLA<sup>+</sup>”. In: *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops - ASYDE, CIFMA, and CoSim-CPS, Amsterdam, The Netherlands, September 14-15, 2020, Revised Selected Papers*. Edited by Loek Cleophas and Mieke Massink. Volume 12524. Lecture Notes in Computer Science. Springer, 2020, pages 43–57. DOI: 10.1007/978-3-030-67220-1\_4

## 4.1 Introduction

Automatically generating correct and performant implementations from high-level specifications is an important challenge in computer science and software engineering. Ideally one makes high-level specifications, which completely describe the functional and relevant parts of an application, without having to bother with low-level implementation details at the same time. Implementation is left to specialized tools and approaches that benefit from automated model checking and other debugging tools.

A benefit of high-level specifications is that they enable more specialized and fine-tuned implementations than general purpose implementation strategies, which in essence have to take into account all possible variations of operations users can define. High-level domain knowledge offers the potential to automatically generate and optimize code, e.g. removing locks and blocking for improved performance when it can derive that this is never necessary in the specific situation.

Such optimizations often involve managing concurrency and parallelism on accessing data. These optimizations of course need to be correct w.r.t. the specification: data consistency needs to be guaranteed. Application logic defines the functional consistency and transaction isolation manages the consistency of concurrent operations. Historically, isolation concerns have been outsourced to database systems, using general purpose transactions and similar constructs. These databases generally support ACID transactions, with a variety of isolation guarantees [9, 32], where Serializability is the strongest guarantee.

In order to optimize the performance of specialized implementations, some parts of the general purpose transaction mechanism are incorporated either in the application itself or in the database implementation. When developing these specialized implementations of higher-level specifications, we need to be sure that they guarantee the ACID properties, or, if not, to what extent. The seminal definition of isolation levels is given by Adya [1]. Adya uses transaction histories, where transactions have dependencies on each other based on accessing the same data. If a cycle can be found in the graph of these dependencies, an isolation anomaly is present. Crooks *et al.* [23] model a state-based and client-centric approach to isolation and prove that it is equivalent to Adya's formalization.

Various tools are available which try to find or visualize isolation anomalies [59, 61, 63]. Many rely on specific scripted error scenarios to show anomalies. The ELLE tool [59] can be used to validate traces of implementations using

## 4.2 Background: State-Based Client-Centric Consistency

Adya's formalization, but still required careful setup and tuning of a test setup. It infers the histories Adya requires from client-centric observed transactions. Crooks' formalization is defined from a client-centric perspective and is directly defined in terms of observed transactions. The state-based and client-centric isolation definitions of Crooks *et al.* are referenced as Crooks' Isolation (CI) throughout this chapter.

This chapter describes an approach using formal methods to (semi-)automatically validate the isolation level of observed transactions using CI. First, we give an introduction to CI and a formalization of it in  $\text{TLA}^+$ . Next we discuss how this formalization is used to validate the consistency guarantees of a transaction algorithm using two-phase commit (2PC) with two-phase locking (2PL), and use it to find a specification bug.

The formalization of CI and the  $\text{TLA}^+$  model checker enable rapid checking of multiple isolation levels of different synchronization algorithms. This technique can be used to both validate observed transactions from run-time systems and of formalizations of algorithms.

The main contributions of this chapter are:

1. Formalization of the core of CI in  $\text{TLA}^+$  and updated definitions to allow incremental model checking (section 4.3).
2. Reproduction of the claims and properties [23] using model checking (section 4.4).
3. Formalization of 2PL/2PC in  $\text{TLA}^+$  and validation of Serializability using model checking of the CI  $\text{TLA}^+$  formalization (section 4.5).
4. An example of finding isolation bugs in the algorithm specification of 2PL/2PC (section 4.5.3).

Section 4.6 discusses results, limitations and future work based on this approach. We conclude in section 4.7. All source code can be found on Zenodo [97].

## 4.2 Background: State-Based Client-Centric Consistency

Crooks *et al.* [23] define a state-based and client-centric consistency model (CI) for reasoning about isolation levels. It defines predicates to state if a set of observed transactions occurs under a given isolation level. The main concepts of CI are transactions and executions. A transaction is a sequence of operations,

$$\left\{ \begin{array}{l} A \mapsto 100 \\ B \mapsto 100 \end{array} \right\}^{S_0} \xrightarrow{T_1} \left\{ \begin{array}{l} A \mapsto 150 \\ B \mapsto 50 \end{array} \right\}^{S_1} \xrightarrow{T_2} \left\{ \begin{array}{l} A \mapsto 165 \\ B \mapsto 55 \end{array} \right\}^{S_2}$$

**Figure 4.1** Example execution with initial state  $S_0$  for transactions  $T_1 = \langle r(A, 100), r(B, 100), w(A, 150), w(B, 50) \rangle$  and  $T_2 = \langle r(A, 150), r(B, 50), w(A, 165), w(B, 55) \rangle$ .

consisting of reads and writes which includes observed keys and values:  $r(k, v)$  /  $w(k, v)$ . An execution represents a possible ordering of a set of transactions with the resulting intermediate database states. A state is a mapping from all database keys to a specific value. Within an execution each following state only differs in the values written by the intermediate transaction on the previous or parent state.

Figure 4.1 shows an example execution of two bank accounts  $A$  and  $B$ , which both have a balance of €100 in the initial state  $S_0$ . Transaction  $T_1$  is money transfer: €50 is deposited from account  $A$  and withdrawn from account  $B$ , realized using two reads and two writes. Transaction  $T_2$  is paying of interest: 10% of the balance is added to both accounts; this transaction also involves two reads and two writes. Note that from a starting state and an ordering of transactions the other states can be derived by applying the intermediate transaction's writes.

For a set of observed transactions  $T$  to satisfy an isolation level  $I$ , a commit test  $CT$  for  $I$  should hold for a possible execution  $e$  of  $T$ :  $\exists e : \forall t \in T : CT_I(t, e)$ . The commit test for serializability, for example, is that all reads in a transaction must be able to have read their value from the direct parent state. In our example all the values of  $T_1$ 's and  $T_2$ 's read operations are the same as their parent state's values for each corresponding key, e.g.  $T_1$ 's  $r(A, 100)$  can read from  $T_1$ 's parent  $S_0$ 's  $A \mapsto 100$ .

Another isolation level is Snapshot Isolation, where the commit test requires that all reads of a single transaction can be read from the same earlier, not necessarily parent, state, which represents the database snapshot.

## 4.3 Formalizing CI in TLA<sup>+</sup>

TLA<sup>+</sup> [69] is a formal specification language for action-based modeling of programs and systems. PLUSCAL [70] is an abstraction on top of TLA<sup>+</sup> for concurrent and distributed algorithms and compiles to TLA<sup>+</sup>. In practice TLA<sup>+</sup> is used to model distributed algorithms and systems [18, 41, 45, 75, 79]. TLA<sup>+</sup> models states and transitions. A specification defines an initial state and atomic steps to a next state. Complex state machines and their transitions can be represented this way. Multiple concurrently-running state machine define their local steps and the global next step non-deterministically picks one machine to progress each step. This captures all possible interleavings of these multiple machines.

CI is formalized as properties that hold on a TLA<sup>+</sup> state. This enables querying the system if an initial database state together with a set of observed transactions satisfies an isolation level, e.g., `Serializability(initialState, setOfTransactions)`. When using TLA<sup>+</sup> to formally specify an algorithm, this isolation property is added as an invariant during model checking. TLA<sup>+</sup>'s model checker TLC can then check the isolation guarantees at every state in the algorithm's execution and produce a counterexample if the invariant is violated.

To formalize CI, we assume the following TLA<sup>+</sup> definitions:

- 1 State  $\equiv$  [Keys  $\rightarrow$  Values]
- 2 Operation  $\equiv$  [op: {"read", "write"}, key: Keys, value: Values]
- 3 Transaction  $\equiv$  Seq(Operation)
- 4 ExecutionElem  $\equiv$  [parentState: State, transaction: Transaction]
- 5 Execution  $\equiv$  Seq(ExecutionElem)

The system State is modeled as a mapping from keys to values. Keys and Values are left abstract on purpose here, since they differ per concrete model. In TLA<sup>+</sup> sets and set membership are often used. [Keys  $\rightarrow$  Values] represents the set of possible tuples of Keys and Values, we bind this to State to easily reference this later in the specification. Operations are a read or write of a value on a key and a Transaction is a sequence of these operations. An Execution is represented as a sequence of transactions with their parent state.

**Listing 4.1** TLA<sup>+</sup> ReadStates

```

1 ReadStates(execution, operation, transaction) ≡
2 LET Se ≡ SeqToSet(executionStates(execution))
3   sp ≡ parentState(execution, transaction)
4 IN { s ∈ Se: |* s ∈ Se
5   ∧ beforeOrEqualInExecution(execution, s, sp) |* a: s →* sp
6   ∧ ∨ s[operation.key] = operation.value |* b1: (k, v) ∈ s
7     |* b2: ∃w(k, v) ∈ ΣT
8   ∨ ∃ write ∈ SeqToSet(transaction):
9     ∧ write.op = "write" ∧ write.key = operation.key
10    ∧ write.value = operation.value
11    |* b2: w(k, v) to r(k, v)
12    ∧ earlierInTransaction(transaction, write, operation)
13    ∨ operation.op = "write"
14  }
    
```

As intuitively sketched earlier CI checks if values could have been read from earlier states. The following definition of  $\mathcal{RS}$  (“read states”) captures this for an execution  $e$  and an operation  $o = r(k, v)$ :

$$\mathcal{RS}_e(o) = \left\{ s \in S_e \left| \begin{array}{l} \text{a} \\ s \xrightarrow{*} s_p \end{array} \wedge \left( \begin{array}{l} \text{b1} \\ (k, v) \in s \end{array} \vee \begin{array}{l} \text{b2} \\ (\exists w(k, v) \in \Sigma_T : w(k, v) \xrightarrow{to} r(k, v)) \end{array} \right) \right. \right\}$$

Read states are a subset of the states in the execution  $S_e$ , which are: (a) up to and including the parent state  $s_p$  in the execution; (b1) have the same key and value as the operation  $o = r(k, v)$ ; or (b2) there exists a write operation  $w(k, v)$  with the same key and value earlier in the same transaction’s operations ( $\Sigma_T$ ).

The TLA<sup>+</sup> version of this definition is shown in listing 4.1. These read states are defined for each operation given an execution. TLA<sup>+</sup>’s syntax allows grouping of conjunctions ( $\wedge$ ) and disjunctions ( $\vee$ ) by vertical indentation. The function `executionStates` denote the sequence of states in an execution. `parentState` extracts the parent state of a transaction given an execution. **LET .. IN** has the standard semantics. The rest of `ReadStates` (lines 4 to 5) follows the CI definition quite literally, except that the third alternative (line 13) is not captured in the CI definition for  $\mathcal{RS}$  above, but represents the “convention [that] write operations have read states too” [23] to include all states up until the parent state for writes.

A state is complete when all reads of a transaction could have read their values from it. It is the intersection of the states in which each operation of the transaction could read from. The following definition is extended to take into account transactions without operations to support the iterative construction of transactions, starting with the empty ones:

$$\text{COMPLETE}_{e,T}(s) \equiv s \in \left( \bigcap_{o \in \Sigma_T} \mathcal{RS}_e(o) \cap \{s' \in S_e \mid s' \xrightarrow{*} s_p\} \right)$$

We omit the  $\text{TLA}^+$  version (Complete) for the sake of brevity, but it closely follows the mathematical definition, just like ReadStates did compared to  $\mathcal{RS}$ .

A *commit test*  $CT_I(T, e)$  determines if a set of transactions  $T$  is valid under an isolation level  $I$  and execution  $e$ . For a set of transactions to satisfy an isolation level, there needs to exist at least one possible ordering, for which the commit test holds for all transactions. Transactions describe the values that a client observes including the actual values read and written. The values observed by the client are compatible with an ordering of the transactions that satisfies the isolation level. This is why it is sufficient for a single possible execution ordering to satisfy the commit test. The specific commit test for an isolation level  $I$  abstracts over which reads are valid for  $I$ .

Different isolation-level commit tests are shown in table 4.1, both mathematically and in  $\text{TLA}^+$ . Note that the CI definitions and their  $\text{TLA}^+$  counterparts are very similar. The definitions of NoConf, Preread, strictBefore and beforeOrEqualInExecution can be found in listing 4.2.

## 4.4 CI examples

The static examples of the CI-paper are reproduced using  $\text{TLA}^+$ 's model checker TLC and the **ASSUME** operator. The model checker checks if the assumed property is valid. Figure 4.2 shows a minimal example of transactions ta to te, which are checked for four different isolation levels given initial state  $s_0$ . TLC checks the assumptions and all evaluate to **TRUE**. The source code [97] reproduces more checks on this example.

**Bank Transfer Example** The bank transfer example introduced by Crooks *et al.*, shows the difference between Snapshot Isolation and Serializability. Alice and Bob simultaneously take money out of their joint current and savings accounts,

**Listing 4.2** TLA<sup>+</sup> helper definitions for CI

```

1 WriteSet(transaction) ≡  $\mathbb{I}^* \mathcal{W}_T = \{k \mid w(k, v) \in \Sigma_T\}$ 
2 LET writes ≡ { operation ∈ SeqToSet(transaction) : operation.op = "write" }
3 IN { operation.key : operation ∈ writes }
4
5 NoConf(execution, transaction, state) ≡  $\mathbb{I}^* \text{NO-CONF}_T(s) \equiv \Delta(s, s_p) \cap \mathcal{W}_T = \emptyset$ 
6 LET Sp ≡ parentState(execution, transaction)
7   delta ≡ { key ∈ DOMAIN Sp : Sp[key] ≠ state[key] }
8 IN delta ∩ WriteSet(transaction) = {}
9
10 Preread(execution, transaction) ≡  $\mathbb{I}^* \text{PREREAD}_e(T) \equiv \forall o \in \Sigma_T : \mathcal{RS}_e(o) \neq \emptyset$ 
11  $\forall$  operation ∈ SeqToSet(transaction): ReadStates(execution, operation, transaction) ≠ {}
12
13  $\mathbb{I}^* T_1 <_s T_2$ 
14 strictBefore(t1, t2, timestamps) ≡ timestamps[t1].commit < timestamps[t2].start
15 beforeOrEqualInExecution(execution, state1, state2) ≡  $\mathbb{I}^* s_1 \xrightarrow{*} s_2$ 
16 LET states ≡ executionStates(execution)
17 IN Index(states, state1) <= Index(states, state2)
    
```

**Table 4.1** Commit tests and corresponding TLA<sup>+</sup> definitions.

Isolation Level	Commit Test	TLA <sup>+</sup> definition
Serializability	$\text{COMPLETE}_{e,T}(s_p)$	Complete(e, T, parentState(e, T))
Snapshot Isolation	$\exists s \in S_e \cdot \text{COMPLETE}_{e,T}(s_p) \wedge \text{NO-CONF}_T(s)$	$\exists s \in \text{toSet}(\text{states}(e)):$ Complete(e, T, s) $\wedge$ NoConf(e, T, s)
Read Committed	$\text{PREREAD}_e(T)$	Preread(e, T)
Read Uncommitted	True	<b>TRUE</b>
Strict Serializability	$\text{COMPLETE}_{e,T}(s_p) \wedge \forall T' \in \mathcal{T} : T' <_s T \Rightarrow s_{T'} \xrightarrow{*} s_T$	<b>LET</b> Sp ≡ parentState(e, t) <b>IN</b> Complete(e, T, Sp) $\wedge \forall$ otherT ∈ transactions(e): strictBefore(otherT, T, ↪ timestamps) ⇒ beforeOrEqualInExecution(e, parentState(e, otherT), Sp)

```

1 |* Initial State, all 0
2  $s0 \equiv [k \in \{x,y,z\} \mapsto 0]$ 
3 |* Helper functions for operations
4  $r(k,v) \equiv$ 
5  $[op \mapsto \text{"read"}, key \mapsto k, value \mapsto v]$ 
6  $w(k,v) \equiv$ 
7  $[op \mapsto \text{"write"}, key \mapsto k, value \mapsto v]$ 
8
9  $ta \equiv \langle\langle w(x,1) \rangle\rangle$ 
10  $tb \equiv \langle\langle r(y,1), r(z,0) \rangle\rangle$ 
11  $tc \equiv \langle\langle w(y,1) \rangle\rangle$ 
12  $td \equiv \langle\langle w(y,2), w(z,1) \rangle\rangle$ 
13  $te \equiv \langle\langle r(x,0), r(z,1) \rangle\rangle$ 
14
15  $trs \equiv \{ta, tb, tc, td, te\}$ 
16 ASSUME Serializability( $s0$ ,  $trs$ )
17 ASSUME SnapshotIsolation( $s0$ ,  $trs$ )
18 ASSUME ReadCommitted( $s0$ ,  $trs$ )
19 ASSUME ReadUncommitted( $s0$ ,  $trs$ )

1 |* Initial state of Current and Savings
    $\hookrightarrow$  accounts.
2  $bInit \equiv (C :> 30) @@ (S :> 30)$ 
3
4  $talice \equiv$ 
5  $\langle\langle r(S,30), r(C, 30), w(C,-10) \rangle\rangle$ 
6  $tbob \equiv$ 
7  $\langle\langle r(S,30), r(C,-10) \rangle\rangle$ 
8  $(* w(S,-10) \text{ does not happen } *) \gg$ 
9  $bTrx \equiv \{talice, tbob\}$ 
10
11 ASSUME Serializability( $bInit$ ,  $bTrx$ )
12 ASSUME SnapshotIsolation( $bInit$ ,  $bTrx$ )
13 ASSUME ReadCommitted( $bInit$ ,  $bTrx$ )
14 ASSUME ReadUncommitted( $bInit$ ,  $bTrx$ )

```

**Figure 4.2** Running example (left) and serializable bank account example (right) from Crooks *et al.* [23] in TLA<sup>+</sup>.

both from the other account. The bank requires the sum of the balances of both accounts to stay positive.

The following execution contains the transactions

$T_{alice} = \langle r(S, 30), r(C, 30), w(C, -10) \rangle$  and  $T_{bob} = \langle r(S, 30), r(C, -10), abort \rangle$ . A serializable implementation requires  $T_{bob}$  to abort.  $T_{alice}$  reads both balances of  $C$  and  $S$  and withdraws €40 from  $C$ .  $T_{bob}$  reads the result and aborts because not enough balance is available for his withdraw of €40 from  $S$ :

$$\left\{ \begin{array}{c} S_1 \\ C \mapsto 30 \\ S \mapsto 30 \end{array} \right\} \xrightarrow{T_{alice}} \left\{ \begin{array}{c} S_2 \\ C \mapsto -10 \\ S \mapsto 30 \end{array} \right\} \xrightarrow{T_{bob}} \left\{ \begin{array}{c} S_3 \\ C \mapsto -10 \\ S \mapsto 30 \end{array} \right\}$$

The TLA<sup>+</sup> code to check this is shown on the right of figure 4.2.

The same example is considered under Snapshot Isolation with transactions  $T_{alice} = \langle r(S, 30), r(C, 30), w(C, -10) \rangle$  and  $T_{bob} = \langle r(S, 30), r(C, 30), w(S, -10) \rangle$ .

Both  $T_{alice}$  and  $T_{bob}$  read from  $S_1$  and find that there is enough total balance available. They both withdraw €40 from respectively  $C$  and  $S$ :

$$\left\{ \begin{array}{l} C \mapsto 30 \\ S \mapsto 30 \end{array} \right\}^{S_1} \xrightarrow{T_{alice}} \left\{ \begin{array}{l} C \mapsto -10 \\ S \mapsto 30 \end{array} \right\}^{S_2} \xrightarrow{T_{bob}} \left\{ \begin{array}{l} C \mapsto -10 \\ S \mapsto -10 \end{array} \right\}^{S_3}$$

Snapshot Isolation allows this because both  $T_{alice}$  and  $T_{bob}$  read from a valid snapshot or complete state and there is no conflict in their writes, because they write to different accounts. However, this violates the overall invariant that the sum of the balances should remain positive. This is the write skew isolation anomaly [1] and is checked by using a specification similar to the right-hand side of figure 4.2, with modified transactions ( $tbAlice \equiv \langle\langle r(S, 30), r(C, 30), w(C, -10) \rangle\rangle$ ;  $tbBob \equiv \langle\langle r(S, 30), r(C, 30), w(S, -10) \rangle\rangle$ ), and observing that Serializability is found to be **FALSE** by the model checker.

## 4.5 Model Checking Algorithms Using ci

In contrast to the previous, static examples, where TLA<sup>+</sup>'s state steps are not used, we now look at a TLA<sup>+</sup> specification of a transactional protocol (2PL/2PC) using states. At each step of the algorithm TLC checks if the isolation guarantees hold.

### 4.5.1 Formalizing 2PL/2PC

Two-Phase Commit (2PC) combined with Two-Phase Locking (2PL) forms a protocol used to implement ACID transactions. 2PC takes care of atomicity of a transaction and 2PL provides Serializable isolation. We extend the formalization of 2PC by Gray and Lamport [41] to support multiple parallel transactions via 2PL.

We model 2PL/2PC in the PLUSCAL algorithm language, which is compiled down to regular TLA<sup>+</sup>, but provides a higher-level notation, closer to imperative programming languages. PLUSCAL describes multiple possibly different processes with atomic steps. During model checking, one of the processes takes a single step, which allows processes to be interleaved. The model checker makes sure all possible interleavings are explored.

The PLUSCAL encoding of 2PL/2PC consists of two types of processes: transaction managers and transaction resources. The actual number of processes

**Listing 4.3** PLUSCAL specification of 2PL/2PC manager

```

1 fair process tm ∈ transactions
2 begin
3 INIT: sendMessage([id ↦ self, type ↦ "VoteRequest"]);
4 WAIT: either !* receive commit votes
5 await ∃ rm ∈ resources: [id ↦ self, type ↦ "VoteCommit", rm ↦ rm] ∈ msgs;
6 sendMessage([id ↦ self, type ↦ "GlobalCommit"]);
7 goto COMMIT;
8 or !* receive at least 1 abort votes
9 await ∃ rm ∈ resources: [id ↦ self, type ↦ "VoteAbort", rm ↦ rm] ∈ msgs;
10 sendMessage([id ↦ self, type ↦ "GlobalAbort"]);
11 goto ABORT;
12 or !* or timeout, solves deadlock when transactions lock each others resources
13 sendMessage([id ↦ self, type ↦ "GlobalAbort"]);
14 goto ABORT;
15 end either;
16 ABORT: goto Done; COMMIT: goto Done;
17 end process

```

is defined by model constants transactions and resources. Message passing is modeled by a monotonically growing set of messages. This means that messages are never lost, but a recipient process might handle them out of order or not at all.

Listing 4.3 shows the definition of the transaction manager. There is a tm process for each of the transactions. PLUSCAL processes do atomic steps, each represented by a label such as INIT:. A label can intuitively be viewed as a state in the process' state machine. All statements within a step are done as a single step.

A transaction manager first sends out the VOTEREQUEST message by adding a tuple with the transaction's identifier self and the message label "VoteRequest" to the msgs set. Then its next step is WAIT in which three alternatives (**either ... or**) can occur: 1) either it receives messages of type "VoteCommit" of each resource occurring in the set of messages, and sends GLOBALCOMMIT; 2) or one message of type "VoteAbort" and sends GLOBALABORT; 3) or it times out and aborts (to prevent deadlock). The **await** construct ensures that a step only happens if its precondition is fulfilled. TLC makes sure that all alternatives are explored. **goto**'s are added to explicitly label the steps for readability in the

**Listing 4.4** 2PL/2PC resource in TLA<sup>+</sup>

```

1 fair process tr ∈ resources
2 variables maxTxs = 5,    /* Limit number of transactions to limit search space
3     voted = {},    /* Transactions which this resource voted for
4     committed = {}, /* Committed transactions
5     aborted = {}, /* Aborted transactions
6     state = 0;    /* Counter to represent state changes for CC
7 begin TR_INIT:
8 while maxTxs >= 0 do
9     either skip; /* skip to not deadlock
10    or /* Wait on VoteRequest
11    with tld ∈ transactions \ voted do
12    await [id ↦ tld, type ↦ "VoteRequest"] ∈ msgs;
13    either /* If preconditions hold, VoteCommit, else VoteAbort
14    sendMessage([id ↦ tld, type ↦ "VoteCommit", rm ↦ self]);
15    voted := voted ∪ {tld};
16    or sendMessage([id ↦ tld, type ↦ "VoteAbort", rm ↦ self]);
17    voted := voted ∪ {tld}; aborted := aborted ∪ {tld}; goto STEP;
18    end either; end with;
19    READY: /* Wait on Commit/Abort
20    either with tld ∈ voted \ committed do /* receive GlobalCommit
21    await [id ↦ tld, type ↦ "GlobalCommit"] ∈ msgs;
22    committed := committed ∪ {tld};
23    operations[tld] := operations[tld] ◦ << r(self, state), w(self, state+1) >>;
24    state := state + 1;
25    end with;
26    or with tld ∈ voted \ aborted do /* receive GlobalAbort
27    await [id ↦ tld, type ↦ "GlobalAbort"] ∈ msgs;
28    aborted := aborted ∪ {tld};
29    end with; end either; end either;
30    STEP: maxTxs := maxTxs - 1;
31 end while; end process;
    
```

model checker's execution. Done is a special PLUSCAL label, which represents the process being completed.

The PLUSCAL specification of a transaction resource, shown in listing 4.4, is slightly more involved. The resource process has local variables (lines 1 to 6) to track of stopping, votes, commits, aborts and resource's state. An integer represents the abstract state and is used when checking `cr`.

When the resource is started (lines 7 to 18), it **either** does nothing (**skip**) and decrements `maxTxs`, **or** receives a "VoteRequest" message. **with**  $tId \in \text{transactions} \setminus \text{voted}$  denotes choosing a transaction ID `tId` from the set of transactions minus the transactions already voted for. The resource can then either `VOTE_COMMIT` or `VOTE_ABORT`. The `voted` local variable keeps track of the transactions it has already voted for and is updated to make sure to only vote once per transaction.

Next, it becomes `READY` (lines 19 to 30) and waits on either `GLOBAL_COMMIT` or `GLOBAL_ABORT`, but only for transactions which it voted for, and has not committed to yet. It keeps track of the committed and aborted transactions in order to not send duplicate messages and to later check the atomicity of the transactions. Each **while** iteration decrements `maxTxs` to ensure termination.

In order to model check `CI` it captures the read and written values in operations (line 23) and updates its local state. Both reads and writes are added on commit and not on vote, because if reads were added on vote, it could be the case that the resource reads a later committed value when responding to the `VOTE_REQUEST` later which will always be aborted anyway. This results in a violation of Serializability for the `CI` check, while it is technically never an observed value.

#### 4.5.2 Model Checking 2PL/2PC

As sanity check for the formalization of 2PL/2PC, first atomicity and termination are checked:

- 1 Atomicity  $\equiv$
- 2  $\forall id \in \text{transactions}: pc[id]=\text{Done} \Rightarrow$
- 3  $\forall a1, a2 \in \text{resources}: \neg id \in \text{aborted}[a1] \wedge id \in \text{committed}[a2]$
- 4
- 5 AllTransactionsFinish  $\equiv \langle \rangle (\forall t \in \text{transactions}: pc[t] = \text{Done})$

For atomicity, when all transactions are completed (process counter `pc` is `Done`), for all pairs of resources it should not ( $\neg$ ) be the case that a transaction is aborted by one resource, but committed by the another. So all should either commit or abort the transaction. Property `AllTransactionsFinish` makes sure that eventually ( $\langle \rangle$ ) all transactions complete.

To model check the isolation guarantees an instance of the `CI` formalization is added, which gives access to the previously defined isolation level tests (see section 4.4), given the initial state and the observed transactions.

**Table 4.2** Run time durations of TLC on CI checks for different number of transactions and resources  $n$  of 2PL/2PC. Results on MacBook Pro (13-inch, 2016) with 3.3 GHz Intel Core i7 with 4 worker threads and allocated 8 GB RAM on AdoptOpenJDK 14.0.1+7, on TLC 2.15 without profiling and using symmetry sets for constants.

#tx	$n = 1$	$n = 2$	$n = 3$
1	7 s	9 s	19 s
2	8 s	21 s	5 m 55 s
3	11 s	1 m 53 s	3 h 21 m 54 s

- 1 ccTransactions  $\equiv$  Range(operations) */\* Operations without transaction IDs*
- 2 InitialState  $\equiv$  [k  $\in$  resources  $\mapsto$  0] */\* Initial state is 0 for all resources*
- 3
- 4 CC  $\equiv$  **INSTANCE** ClientCentric **WITH** Keys  $\leftarrow$  resources, Values  $\leftarrow$  0..10
- 5 Serializable  $\equiv$  CC!Serializability(InitialState, ccTransactions)
- 6 SnapshotIsolation  $\equiv$  CC!SnapshotIsolation(InitialState, ccTransactions)
- 7 ReadCommitted  $\equiv$  CC!ReadCommitted(InitialState, ccTransactions)
- 8 ReadUncommitted  $\equiv$  CC!ReadUncommitted(InitialState, ccTransactions)

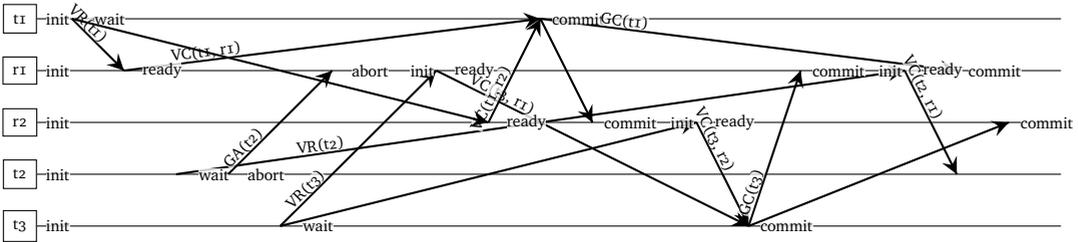
In this case all cases are valid when we run the TLC model checker for transactions  $\equiv$  {t1, t2} and resources  $\equiv$  {r1, r2, r3}.

The model checker then checks the isolation guarantees for each step of the algorithm. When the isolation test fails, it presents a counterexample. Table 4.2 gives an intuition on the relative time durations of the TLC model checker on different numbers of transactions and resources. The model checker checks the four CI isolation levels (Serializability, Snapshot Isolation, Read Committed, Read Uncommitted) on each of the model’s steps. It never invalidates the checks, so it traverses the entire state space.

### 4.5.3 2PL/2PC Bug Seeding

To additionally stress the formalization presented above, we have introduced a subtle, but realistic bug in the definition of transaction resource. When the resource is in the ready state and waiting on a GLOBALCOMMIT or GLOBALABORT message from the transaction manager, the resource should only wait for these messages when it is the actual transaction it voted for. This is guaranteed by

## 4.5 Model Checking Algorithms Using ci



**Figure 4.3** Non-serializable trace found for bugged 2PL/2PC specification. Horizontal lines represent processes over time with state changes. Arrows represent messages sent and received. Message labels are abbreviations of 2PC messages: VOTEREQUEST, VOTECOMMIT, GLOBALABORT and GLOBALCOMMIT.

with  $tId \in \text{voted} \setminus \text{committed}$  in listing 4.4 line 20. The bug is to replace this with  $\text{with } tId \in \text{transactions} \setminus \text{committed}$ . This means  $tId$  can faultily represent a never-seen before transaction as well.

When this model is checked with two transactions and resources, all of the invariants hold and no problem is found. However, with three transactions and two resources the Serializability invariant is violated and a counterexample with 20 steps is found within half a minute; this trace shown in figure 4.3. The example shows that due to this bug it is possible for a resource to side-step an in progress transaction, by responding to the GLOBALCOMMIT of a different transaction.

First  $t1$  and  $t2$  request to vote and  $r1$  votes to commit for  $t1$ , then  $t2$  aborts due to timeout with GLOBALABORT( $T2$ ).  $r1$  then uses this abort to abort its waiting on  $t1$ . This is possible because  $\text{with } tId \in \text{transactions} \setminus \text{committed}$  allows  $r1$ . It receives the GLOBALABORT( $T2$ ), aborts and steps to receive the next transaction. The model checker requires some more steps to find non-serializable behavior, when the other transactions  $t1$  and  $t3$  commit and their effects are applied in different order on  $r1$  and  $r2$ , hence the system is not serializable.

These kinds of bugs during specification can occur naturally, for example when specializing algorithms for specific applications with the goal of added efficiency [103]. Using `ci` in model checking helps us find bugs while designing new algorithms and also for validating claims of existing algorithms.

## 4.6 Discussion and Future Work

The formalization of CI in  $\text{TLA}^+$  is relatively straightforward. The definitions for the base abstractions, such as State and Execution, influence the whole formalization. Staying as close as possible to the mathematical model however, results in quite verbose output, since there are no labels on transactions. The definition on read states was improved to support incremental model checking, starting with empty transactions.

The main limitation of using model checking to find isolation violations is the state explosion when the numbers of processes grows. As seen in table 4.2, running times grow rapidly and model checking becomes infeasible when more transactions are added. Since the model checker evaluates the isolation guarantees in every algorithm state we assume, however, that most isolation violations can be found in small examples. The small scope hypothesis [56] supports this saying that most bugs have small counterexamples. Nevertheless, we can not be entirely sure that anomalies that only occur in larger interactions and longer traces are found by the current approach, but it gives us confidence in the the checked isolation level, while keeping it feasible.

There is a lot of research focusing on proving distributed consistency properties. Model checking tools, such as Uppaal [12], Spin [55], LTSMin [15], mCRL2 [44] and  $\text{TLA}^+$  [69] are used to verify distributed systems and algorithms as well as real-world implementations and protocols [39, 45, 75, 79].

There are also many approaches [9, 62, 86, 110] that try to balance the trade-off between performance and data-consistency by choosing different isolation guarantees. Our work adds to this knowledge by providing a reusable framework to investigate and model check distributed consistency protocols.

To further evaluate the usefulness of our approach for real-life systems, it would be insightful to reproduce known isolation bugs in older versions of database implementations, such as found by Jepsen [59, 60] and Bailis *et al.* [9]. In order to do this we could either create one or more clients that capture the observed transactions, or instrument the database to store this information for offline model checking.

Furthermore the scripts of the isolation anomalies of Hermitage [63] can be reproduced as TLC model checks to strengthen (our formalization of) CI. The  $\text{TLA}^+$  Toolbox also features a theorem prover. The CI formalizations could be extended by proving certain properties, such as reproducing the proofs on equivalence with Adya's formalization and proving conformity to isolation levels for specific algorithms.

Generating performant and correct implementations from high-level specifications is an attractive goal in software engineering, as it would bring the benefits of (semi-)automatic verification to correct-by-construction implementation.

For instance, the Rebel domain-specific language has been used to specify realistic systems (for instance, in the financial domain), from which highly scalable implementations are generated using novel consistency algorithms [100, 103, 108]. It is however, a far from trivial endeavour to state and prove isolation guarantees of some of these algorithms. CI can be extended to support operations on a semantically higher level than reads and writes, such as the semantically richer operations used in Rebel. A  $\text{TLA}^+$  formalization can then be used to allow for rapid prototyping of synchronization implementation alternatives for Rebel, while leveraging the higher-level semantics [114]. The checking of isolation guarantees can then be automated.

## 4.7 Conclusion

This chapter formalizes Crooks' state-based client-centric isolation model (CI) in  $\text{TLA}^+$  in order to check conformance to isolation levels using model checking. The running examples of Crooks *et al.* [23] are reproduced and validated in  $\text{TLA}^+$ . An example of a transaction implementation using two phase locking (2PL) and two phase commit (2PC) is formalized in  $\text{TLA}^+$ . The TLC model checker is used to automatically show conformance to the CI formalization. The CI formalization is also used to find a bug in the algorithm's formalization.

Formalizing CI in  $\text{TLA}^+$  enables automatic validation of isolation guarantees of synchronization implementations by mapping their algorithms to read and write operations. It can be used both for checking isolation conformance of runtime traces of (distributed) systems and of formal specification of algorithms.



# 5

## Safely Exploiting Contract-Based Return-Value Commutativity for Faster Serializable Transactions

**Abstract** A key challenge of designing distributed software systems is maintaining data consistency. We can define data consistency and data isolation guarantees – e.g. serializability – in terms of schedules of atomic reads and writes, but this excludes schedules that would be semantically consistent. Others use manually provided information on “non-conflicting operations” to define guarantees that work for more applications allowing more parallel schedules. To be safe, an engineer might avoid marking operations as non-conflicting, with detrimental effects to efficiency. To be fast, they might mark more non-conflicting operations than is strictly safe.

Our goal is to help engineers by automatically deriving commutative operations (using their respective contracts) such that more parallel schedules with global consistency are possible. We define a new general consistency and isolation guarantee named “Return-Value Serializability” to check consistency claims automatically, and we present distributed event processing algorithms that make use of the same “Contract-based Commutativity” information. We validated both the definitions and the algorithms using model-checking with  $\text{TLA}^+$ . Previous work provided evidence that local coordination avoidance such as applied here has a significant positive effect on the performance of distributed transaction systems.

Client-centric return-value commutativity promises to hit a sweet spot in design trade-offs for business applications, such as payment systems, that must scale-out while their operations are not embarrassingly parallel and consistency guarantees are of the highest priority. It can also provide design feedback, indicating that some operations will simply not scale together even before a line of code has been written.

---

This chapter is previously published as: Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Contract-Based Return-Value Commutativity: Safely exploiting contract-based commutativity for faster serializable transactions”. In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2021*. ACM Press, 2021. DOI: 10.1145/3486601.3486707

## 5.1 Introduction

Fast implementations of serializability and other strongly consistent isolation levels are inherently complex in a distributed setting, due to the inherent trade-off between performance and consistency. Typical approaches to increase concurrency of distributed operations operate at the level of low-level reads and writes. Looking at higher-level abstractions (methods, operations, etc.), however, allows for more leniency: more schedules of operations are serializable because the semantics of high-level operations is used directly, rather than decomposed in their constituent parts. Weikum’s multi-level serializability [115] is a model of how this can work. State-dependent commutativity and return-value commutativity describe when it is safe to reorder operations without violating serializability. For instance, deposits on the same bank account are commutative and therefore non-conflicting. In practice this means that deposit operations can be reordered (swapped) resulting in the same account balance, but potentially better performance. As long as no later operations expose the intermediate state this swap is valid under serializability.

These descriptive formalisms are not used much in practice due to use-case specificity and the need for manual specification of non-conflicting operations. In this chapter we propose a constructive alternative, called Contract-Based Commutativity (CBC) that can be leveraged at run time to determine if operations are potentially commutative. Next to that we formalize (and implement), Local-Coordination Avoidance (LoCA), which uses CBC to increase parallelism in high-contention scenarios, while maintaining serializability isolation guarantees. In order to validate the correctness of the algorithm, the notion of Return-Value Serializability (RV-SER), based on the same contracts as CBC, is defined and formally specified in  $TLA^+$ . The LoCA algorithm is validated using model checking to maintain RV-SER.

Our approach focuses on (distributed) state machines with clearly defined operations, but can be generalized to other settings. Communication is done via transactions of synchronized operations, in which multiple objects do atomic synchronized transitions. For instance, a withdraw on a bank account needs to happen atomically with a deposit on another account state machine. The contract is that each operation has a return value and effect given an object state.

For example, take a simple bank account, with a balance  $b$ , and higher level operations Deposit and Withdraw. The contract is denoted as:  $\text{operation}(\text{arguments})/\text{effect} \uparrow \text{return value}$ . To prevent overdraft, Withdraws

**Table 5.1** All contributions, concepts and abbreviations introduced and referenced in this chapter.

Abbr.	Description	Contribution or related work	Sect.
CBC	Contract-Based Commutativity (CBC), a constructive definition to determine which operations can safely run concurrently at run time without violation of serializability. A sufficient condition for SDC and RVC.	Contribution	5.3
CBC*	Optimized variant of CBC, used in the LoCA implementation.	Contribution	5.3
SCBC	Static CBC, an encoding in SMT that allows computing static CBC for state-machine models, including a comparison between SIE and SCBC.	Contribution	5.5.1
RV-SER	Return-Value Serializability (RV-SER), a serializability definition and formalization for high-level operations in $TLA^+$ which defines if a schedule is compatible with observed return values when swapping operations, using the same contract as CBC.	Contribution	5.4, 5.6, 5.7
LoCA	Local Coordination Avoidance (LoCA), an algorithm, formalization and implementation leveraging conflict relations at run time to increase concurrency. Contributed conflict relations CBC* and statically determined SCBC maintain RV-SER.	Contribution and Soethout et al. [100, 103]	5.5, 5.6, 5.7
2PL/2PC	Two-Phase Locking/Two-Phase Commit, respectively providing Isolation and Atomicity. Used as building blocks for LoCA and to show RV-SER is sufficient to find serializability violations in a bugged formalization.	Gray and Lamport [41], $TLA^+$ model [101]	5.5, 5.6, 5.7
IE	Independent Events, a definition of independent operation pairs, guaranteeing local internal state machine consistency, but not global serializability.	Soethout et al. [100, 103]	5.5
SIE	Subset of IE, statically determined for all possible object states using an SMT solver.	Soethout et al. [100]	5.5
CI	Client-Centric Isolation model, on which RV-SER is inspired, based on low-level reads and writes.	Crooks et al. [23], $TLA^+$ model [101]	5.4, 5.7
SDC	State-Dependent Commutativity, a definition based on return values, describing when schedules with swapped operations are serializable given a specific state.	Weikum and Vossen [115]	5.2, 5.3
RVC	Return-Value Commutativity, a definition based on return values, describing when schedules with swapped operations are serializable given an arbitrary sequence of previous operations.	Weikum and Vossen [115]	5.2, 5.3

check if enough balance is available and only then returns an updated state:  $\text{Withdraw}(a)/\text{if}(b \geq a) b - a \text{ else } b \uparrow b \geq a$ . Deposits always return success (OK):  $\text{Deposit}(a)/b + a \uparrow \text{OK}$ . The simplest way to guarantee isolation is to only have a single operation active at any moment in time, but this also means that operations have to wait on each other. LOCA with CBC allows multiple operations active at the same moment in time, but only when local object invariants and global serializability invariants are maintained, e.g. multiple Withdraw operations can only run in parallel if there is enough balance available for all. CBC checks if committing or aborting the operation does not change the return values (success of the withdraw) of the others.

Earlier variants of LOCA based on Independent Events [100, 103] instead of CBC, can exhibit non-serializable behavior where operations are applied in different order on different objects, even though the behavior is locally consistent and does not violate the object consistency/lifecycle definitions. This chapter improves on this by guaranteeing serializable behavior with CBC. Since LOCA parallelizes operations when they are non-conflicting, performance improvements in high-contention scenarios are similar to Independent Events [100, 103] for CBC operations. The formalization of LOCA and RV-SER in  $\text{TLA}^+$  also enables validating of run-time schedules of implementations.

This chapter's contributions are detailed in table 5.1 including section references and the most important abbreviations used. Section 5.2 gives background on the grounding of RV-SER and CBC. Sections 5.3 to 5.6 describe the main contributions. Section 5.7 evaluates the formalizations with model checking approach in  $\text{TLA}^+$  to show that interleaved processes using LOCA indeed maintain RV-SER. Discussion of the work including threats to validity are found in section 5.8. Lastly we discuss related work (section 5.9), and conclude in section 5.10. All source code and reproducibility scripts are available on Zenodo [98].

## 5.2 Background: State-Dependent Commutativity (SDC) and Return-Value Commutativity (RVC)

Aguilera and Terry [2] identify two kinds of consistency: state consistency and operation consistency. State consistency covers when an application is in a correct state using invariants on states. This definition is very application specific, because it is defined on application dependent states and corresponds

## 5.2 Background: State-Dependent Commutativity and Return-Value Commutativity

to the consistency in ACID. Operation consistency concerns operations that may return values and relates to isolation in ACID. This is often depicted as abstract operations such as reads, writes on data items. Note that operation consistency allows an application's state to be inconsistent as long as it is not visible/inconsistent for clients querying/doing operations: external operation consistency is maintained. State consistency is defined by invariants on operations and the local object state.

Weikum and Vossen look at higher-level operations and describe State-Dependent Commutativity and Return-Value Commutativity [115] as a way that enables multi-level serializability. Informally, non-conflicting, commutative operations can be swapped in a schedule while maintaining serializability. Swapping commutative operations is a proof that the schedule is equivalent to a serial schedule and thus also valid under serializable isolation. The main insight is that operations are commutative on a higher semantic level, and if their direct lower level children such as reads and writes on data are atomic (no crossing tree arcs), they can be swapped without loss of serializability.

Non-conflicting operations are either on different objects ( $A.\text{Deposit}(x)$  and  $B.\text{Withdraw}(x)$ ); or commutative ( $A.\text{Deposit}(x)$  and  $A.\text{Deposit}(y)$ ).

A schedule with operations,  $A.\text{Withdraw}(x)$  and  $A.\text{Deposit}(x)$  of transaction  $t_i$  on object  $A$  are abbreviated respectively as  $+x_A^{t_i}$  and  $-x_A^{t_i}$ . Schedule  $-10_B^{t_1} - 20_B^{t_2} + 20_A^{t_2} + 10_A^{t_1}$  is not serializable under Adya's [1] and Crooks' [23] reads/writes level models, since there is a cycle in the dependency graph between transactions:  $t_1 \leftrightarrow t_2$ . However at a higher operation level it is equivalent to serializable orders:  $-10_B^{t_1} + 10_A^{t_1} - 20_B^{t_2} + 20_A^{t_2}$  and  $-20_B^{t_2} + 20_A^{t_2} - 10_B^{t_1} + 10_A^{t_1}$ , because it results in the same end state.

**SDC** State-Dependent Commutativity describes if two operations  $p$  and  $q$  are commutative in a concrete object state  $\sigma$ .  $p$  and  $q$  are SDC if schedule  $pq\omega \rightarrow qp\omega$ , where the return values of  $p$ ,  $q$  and all possible later operations  $\omega$  should stay the same when  $p$  and  $q$  are swapped given state  $\sigma$ .

**RVC** Return-Value Commutativity abstracts from a concrete run-time state and looks at all possible sequences of previous operations  $\alpha$ , instead of only to a state  $\sigma$ :  $\alpha pq\omega \rightarrow \alpha qp\omega$ , where also the return values should be the same when  $p$  and  $q$  are swapped.

**Return Values** Both definitions depend on the notion of return values. An operation (e.g.  $+20_A \uparrow \text{OK}$ ) is invoked on an entity ( $A$ ), has a name or type

(Deposit/+), input parameters (20) and return values (OK). All operation have an either explicit ( $+50_A \uparrow \text{NOK}$ ) or implicit ( $\text{GetBalance}_A \uparrow 100$ ) success (OK) or failure (NOK) return value. These return values should not differ when  $p$  and  $q$  are swapped.

### 5.3 Contract-Based Commutativity: actionable SDC and RVC

SDC and RVC describe formally when sequences of operations are serializable. In order to use this knowledge in practice we require a constructive form that can be used at run time to determine when swapping and concurrent operations are safe.

Contract-Based Commutativity (CBC) is geared towards local run-time computability in an object, without communication with other objects. Given in-progress operations, it determines if a new incoming operation can run concurrently without violating consistency and isolation guarantees. CBC defines constructive requirements, which are sufficient for SDC and RVC.

To exploit these higher level semantics CBC depends on detecting conflicting operations. Operations on different objects are always non-conflicting and operations that expose the same return values when swapped are non-conflicting depending on the operations and the object state. In order to detect the latter, CBC requires a contract on the operations of the object, consisting of two deterministic side-effect free functions. The effect determines the next internal object's state and the return value determine which values are exposed to the outside.

First we look at computing dynamically CBC at run time. An object locally determines which operations are safe to run in parallel. Next (section 5.5.1) we look at which of these operations are always safely parallelized independently of the run-time state. This reduces run-time computation overhead for specific use cases. The parallel here is with respect to SDC and RVC. Dynamic CBC is valid in a specific run-time state  $\sigma$  from SDC, where static CBC holds in all possible run-time states, corresponding with all possible previous sequences of operations  $\alpha$  from RVC.

### 5.3 Contract-Based Commutativity: actionable sdc and rvc

**Table 5.2** CBC for different return values OK / NOK. Properties in braces are always true/tautologies.  $\equiv$  is state equivalence.

$\text{CBC}(s, p, q)$	$q \uparrow \text{OK}$	$q \uparrow \text{NOK}$
$p \uparrow \text{OK}$	$q \uparrow \text{OK in } s \wedge$ $q \uparrow \text{OK in } s_p \wedge$ $(p \uparrow \text{OK in } s) \wedge$ $p \uparrow \text{OK in } s_q \wedge$ $s_{pq} \equiv s_{qp}$	$q \uparrow \text{NOK in } s \wedge$ $q \uparrow \text{NOK in } s_p \wedge$ $(p \uparrow \text{OK in } s) \wedge$ $(p \uparrow \text{OK in } s_q) \wedge$ $s_{pq} \equiv s_{qp}$
$p \uparrow \text{NOK (in } s)$	$q \uparrow \text{OK in } s \wedge$ $(q \uparrow \text{OK in } s_p) \wedge$ $(p \uparrow \text{NOK in } s) \wedge$ $p \uparrow \text{NOK in } s_q \wedge$ $s_{pq} \equiv s_{qp}$	$q \uparrow \text{NOK in } s \wedge$ $(q \uparrow \text{NOK in } s_p) \wedge$ $(p \uparrow \text{NOK in } s) \wedge$ $(p \uparrow \text{NOK in } s_q) \wedge$ $(s_{pq} \equiv s_{qp})$

#### 5.3.1 Computing CBC at Run Time

Consider a run-time object which receives an operation (return value yet to be calculated). Its current internal state  $s$  is known. Now, since the operation is part of a larger set of operations on multiple objects (transaction), it can abort due to another object. So, before the operation is definitely committed, it is not final and its effects can not yet be applied. The object can thus have one or more of these tentative operations queued. When another operation arrives, it decides whether it can already determine the return values, or wait until more tentative operations finish.

In order to define  $\text{CBC}(s, p, q)$ , for a run-time state  $s$  and operations  $p$  and  $q$  we look at a simple abstraction, based on operations that can only return OK and NOK. Table 5.2 contains the different case distinctions possible,  $p$  and  $q$  either return OK or NOK. This directly corresponds with SDC's  $pq\omega \rightarrow qp\omega$ . The matching return values in  $\omega$  are over-approximated by equating the post state in both orders ( $s_{pq} \equiv s_{qp}$ ), since any difference in return values in later operations can only come from difference in internal state. The first row corresponds with  $\text{CBC}(s, p \uparrow \text{OK}, q)$ , where  $q$  has two possible return values (per column). Operation  $p$  with return value OK is already in progress, meaning it is waiting on the transactions to signal to commit and apply  $p$ . CBC holds if  $q$  returns the same values in state  $s$  and  $s_p$ .  $s_p$  denotes state  $s$  with  $p$ 's effects applied.

## Chapter 5 Contract-Based Return-Value Commutativity

$p$  tautologically returns OK in  $s$  for this row, because  $p$  is already in-progress given state  $s$ . For the  $q \uparrow \text{OK}$  column it has to be checked if  $p$  still returns OK when  $q$  is first applied. For  $q \uparrow \text{NOK}$  this is always the case, because effects of NOK operations are always empty, meaning that  $s \equiv s_q$ . When leaving in the tautological properties a pattern can be observed that generalizes to arbitrary return values:

$$\begin{aligned} \text{CBC}(s, p, q) = & q \uparrow rv_q \text{ in } s \quad \wedge \\ & q \uparrow rv_q \text{ in } s_p \quad \wedge \\ & p \uparrow rv_p \text{ in } s \quad \wedge \\ & p \uparrow rv_p \text{ in } s_q \quad \wedge \\ & s_{pq} \equiv s_{qp} \end{aligned}$$

where  $q$ 's return value  $rv_q$  is the same in  $s$  and after  $p$  in  $s_p$ , and  $p$ 's return value  $rv_p$  is the same in  $s$  and after  $q$  in  $s_q$ .

Above definition leads to a constructive, computable definition under the assumption that return values can be calculated deterministically without side effects from a state and an operation using  $retVal(s, o) : State \times Operation \rightarrow ReturnValue$ . CBC is defined as follows:

$$\begin{aligned} \text{CBC}(s, p, q) = & retVal(s, p) \equiv retVal(s_q, p) \wedge \\ & retVal(s_p, q) \equiv retVal(s, q) \wedge \\ & s_{pq} \equiv s_{qp} \end{aligned}$$

$s_{pq} \equiv s_{qp}$  might lead to false negatives (operations not being marked as CBC), but never to false positives (operations erroneously being marked as CBC).

If  $s_{pq} \equiv s_{qp}$  would be omitted, the definition would be wrong. For example, when tracking a history of past deposits and withdrawals in a bank account and a later query operation (part of  $\omega$ ) returns this sequence, Deposit and Withdraw should no longer be CBC. Because the history is not represented in the return values of  $p$  and  $q$  but can be visible in later operations. Such a model with a history sequence is thus inherently non-parallelizable, but another model, for example tracking a set instead of sequence is. CBC can be used to detect this, as is shown in section 5.7.

**Example** Consider the following example with CBC, based on money transfers between two bank accounts without overdraft (balance  $\geq \text{€ } 0$ ). There are three transactions  $T_1 : B \xrightarrow{\text{€ } 10} A$ ,  $T_2 : B \xrightarrow{\text{€ } 20} A$ ,  $T_3 : A \xrightarrow{\text{€ } 30} B$ , transferring money between

### 5.3 Contract-Based Commutativity: actionable sdc and rvc

the accounts  $A$  to  $B$  with a respective starting balance of  $\text{€ } 0$  and  $\text{€ } 100$ . Where each transfer consists of a withdrawal ( $-10_B$ ) and deposit ( $+10_A$ ) operation on the relevant account. A Withdraw returns `NOK` if not enough balance is available, otherwise `OK`. Deposit always returns `OK`.

A possible run-time trace is:  $+10_A+30_B+20_A-20_B-30_A-10_B$ , where all operations have `OK` return values. Operations are ordered differently on the different accounts, because of arrival order.:  $A : \langle T_1, T_2, T_3 \rangle$  and  $B : \langle T_3, T_2, T_1 \rangle$

Applying CBC shows if this schedule is compatible with a serializable schedule. A schedule is serializable when all operations of all transactions do not interleave with other transaction's operations. In order to find out if the current schedule is compatible or equivalent with a serial schedule, we consider swappable operations with respect to CBC. Operations on different objects (or in this case accounts) are always commutative and can be swapped, so it is sufficient to see if both accounts' operations can be swapped to arrive at the same transaction order.

For  $B$  to arrive at  $\langle T_1, T_2, T_3 \rangle$  two CBC-swaps are required:  
 $+30_B-20_B-10_B \xrightarrow{\text{CBC}(B_{+30}, -20, -10)} +30_B-10_B-20_B \xrightarrow{\text{CBC}(B, +30, -10)} -10_B-20_B+30_B$ ,  
 where  $B_{+30}$  represents the state of  $B$  with effects of shown operation applied. So, if  $\text{CBC}(\text{€ } 130, -20, -10)$  and  $\text{CBC}(\text{€ } 100, +30, -10)$  hold, these schedules are compatible and the original schedule is serializable.

$$\text{CBC}(\text{€ } 130, -20, -10) = \text{OK} \equiv \text{OK} \wedge \text{OK} \equiv \text{OK} \wedge \text{€ } 100 \equiv \text{€ } 100$$

$$\text{CBC}(\text{€ } 100, +30, -10) = \text{OK} \equiv \text{OK} \wedge \text{OK} \equiv \text{OK} \wedge \text{€ } 120 \equiv \text{€ } 120$$

Both hold, so the swap is valid, and thus the original schedule is CBC-equivalent to a serial order and thus serializable. Note that both CBC-checks above are also checked by computing  $\text{CBC}(B, [+30, -20], +10)$  in an implementation, as covered in the next section.

A non-equivalent schedule with the same order of operations, but with account  $B$  also starting with a state of  $\text{€ } 0$ , results in not allowing the same swaps and therefore not being serializable, since  $\text{CBC}(\text{€ } 0, +30, -10)$  does not hold:

$$\text{CBC}(\text{€ } 30, -20, -10) = \text{OK} \equiv \text{OK} \wedge \text{OK} \equiv \text{OK} \wedge \text{€ } 0 \equiv \text{€ } 0$$

$$\text{CBC}(\text{€ } 0, +30, -10) = \text{OK} \equiv \text{OK} \wedge \text{OK} \equiv \text{NOK} \wedge \text{€ } 20 \equiv \text{€ } 30$$

This property can be calculated at run time, because all arguments are available locally. When more operations are in progress, the new incoming

operation should be CBC with all of them, meaning it can be swapped to become the earliest operation in progress. When an incoming operation is not CBC it need to be delayed until offending in-progress operations commit or abort.

### 5.3.2 CBC for Multiple In-progress Operations

The approach sketched so far only considers a pair of two operations. This section describes the induction step from  $\text{CBC}(s, o_1, o_i)$  to  $\text{CBC}(s, [o_1, \dots, o_n], o_i)$ , where  $o_1, \dots, o_n$  represent multiple in-progress operations.

For example, first no operations are in progress on an object. A first operation  $o_1$ , part of a transaction  $t_1$  can start processing. Due to other (slower) participants, it is not known if the  $o_1$  actually commits and if  $o_1$ 's effects should be applied. In a locking implementation, another arriving operation  $o_2$  has to wait until  $t_1$  commits or aborts. However, if  $\text{CBC}(s, o_1, o_2)$  holds,  $o_1$  and  $o_2$  can effectively be swapped, without changing the return values of both. Schedules  $o_1o_2$  and  $o_2o_1$  are compatible, because  $\text{CBC}(s, o_1, o_2)$  holds. Therefore  $o_2$  can also be started. Now there are two operations in progress.

When another operation  $o_3$  arrives, it effectively must be swappable with both in-progress operations in order to stay serializable, because all swapping orders need to be compatible.

CBC with multiple in-progress operations, represented as a list of operations in the second argument, is reducible to CBC with a single in-progress operation:

$$\begin{aligned}
 \text{CBC}(s, [o], o_i) &= \text{CBC}(s, o, o_i) \\
 \text{CBC}(s, [o_1, \dots, o_{n-1}, o_n], o_i) &= \text{CBC}(s, [o_1, \dots, o_{n-1}], o_i) \wedge \quad (\text{A}) \\
 &\quad \text{CBC}(s_{1..n-1}, o_n, o_i) \wedge \quad (\text{B}) \\
 &\quad \text{CBC}(s, [o_1, \dots, o_{n-1}], o_n) \quad (\text{C})
 \end{aligned}$$

CBC holds when: (A)  $o_i$  is CBC without the last operation in progress; (B)  $o_i$  is CBC with the last in-progress operations in the state with all earlier operations applied ( $s_{1..n-1}$ ); and (C) also the last in-progress operation  $o_n$  is CBC with all previous in-progress operations.

An implementation can skip calculating part C, because arriving at an incoming operation  $o_i$  at  $\text{CBC}(s, [o_1, \dots, o_n], o_i)$ , means that  $\text{CBC}(s, [o_1, \dots, o_{n-1}], o_n)$  is already determined at an earlier stage, when  $o_n$  was the incoming operation. This means that an implementation can compute CBC as follows:

$$\text{CBC}^*(s, O, o_i) = \forall o_n \in O. \text{CBC}(s_{1..n-1}, o_n, o_i) \quad (5.1)$$

where  $O$  is the sequence of in-progress operations. This optimized version of CBC, dubbed  $\text{CBC}^*$ , is used in the LOCA implementation in section 5.5 to achieve serializable isolation with increased concurrency.

## 5.4 Return-Value Serializability

Definitions of isolation guarantees, such as Adya [1], use read and write operations to determine violations. In order to also define these guarantees on higher level operations and fairly evaluate algorithms leveraging CBC, this section introduces Return-Value Serializability (RV-SER). RV-SER defines which schedules are serializable w.r.t. commutative operations and is formalized in  $\text{TLA}^+$ , which enables model checking of schedules of operations and algorithms which capture such schedules. The definitions follow a structure similar to a client-centric isolation model from Crooks et al. [23], referred to as Crooks' Isolation (CI), and the formalization in  $\text{TLA}^+$  builds on earlier work [101].

**Crooks' Isolation** This client-centric model of database isolation defines which sets of observed transactions, consisting of read and write operations with their values, are valid under different isolation levels. For each level, such as serializability, a commit test defines if the set complies. Only a single possible ordering of transactions, adhering to the commit test has to exist. This means that the observed transaction could have occurred under that isolation level.

$$\exists e \in E. \forall t \in T : CT_I(t, e)$$

defines for an isolation level  $I$ , and its commit test  $CT_I$ , where  $E$  is the set of all possible orderings of transactions (executions) and  $T$  is the set of observed transactions. Executions  $E$  consist of transactions as a whole, constructed by applying the writes of the transaction to the previous state. Commit tests check if reading from earlier state is valid. In this chapter we focus on serializability, but the approach can be extended to different levels. The commit test for serializability under CI specifies that all reads should be able to read the observed value from the direct parent state.

**RV-SER** The main insight for RV-SER is to not look at reads and writes, or try to map higher operations to lower level reads/writes, but to consider operations at a higher level as a whole. This is based on multi-level serializability by

Weikum and Vossen [115], which states that if operations are not interleaving on a lower-level, they can be swapped on the higher level if non-conflicting, while maintaining serializability. Similar to the client-centric approach, *RV-SER* considers observed values from the operations, in this case the values returned by the operations. For low levels this corresponds to the read or written values, but for higher level operations, this is different, e.g. a *Withdraw* or *Deposit* might just return *OK* or *NOK* to signal operation success or failure and a *GetBalance* operation returns a single balance value.

As commit test for serializability, *RV-SER* defines that newly computed return values for all operations should be the same as the observed return values:

$$\forall o \uparrow o' \in T.\text{retVal}(s_p, o) \equiv o' \quad (5.2)$$

where  $o'$  represents the observed return value, which should be equivalent to the return values in  $o$ 's parent state in the execution  $s_p$ .

The main differences with Crooks' Isolation are:

- Operations consist of observed return values, a *retVal* function to calculate return values given arbitrary state and an effect function to calculate next state  $\text{eff}(s, o)$ . This is the same contract as for *CBC*.
- Commit test checks return values, instead of read/write values

**Examples** Consider the same execution schedule as before, now including return values:  $+10_A \uparrow \text{OK}; +30_B \uparrow \text{OK}; +20_A \uparrow \text{OK}; -20_B \uparrow \text{OK}; -30_A \uparrow \text{OK}; -10_B \uparrow \text{OK}$  consisting of three transactions:  $T_1 = \langle +10_A \uparrow \text{OK}, -10_B \uparrow \text{OK} \rangle$ ,  $T_2 = \langle +20_A \uparrow \text{OK}, -20_B \uparrow \text{OK} \rangle$  and  $T_3 = \langle -30_A \uparrow \text{OK}, +30_B \uparrow \text{OK} \rangle$ .

Below we see the execution of  $\langle T_1, T_2, T_3 \rangle$ . An execution consists of data(base) states with keys and values, each next state is determined by applying the operations of the relevant transactions.

$$\left\{ \begin{array}{l} A \mapsto 0 \\ B \mapsto 100 \end{array} \right\}^{s_0} \xrightarrow{T_1} \left\{ \begin{array}{l} A \mapsto 10 \\ B \mapsto 90 \end{array} \right\}^{s_1} \xrightarrow{T_2} \left\{ \begin{array}{l} A \mapsto 30 \\ B \mapsto 70 \end{array} \right\}^{s_{12}} \xrightarrow{T_3} \left\{ \begin{array}{l} A \mapsto 0 \\ B \mapsto 100 \end{array} \right\}^{s_{123}}$$

For all transactions, as per the commit test above (equation (5.2)) all operations have the same return values given the parent state in this execution as the observed return value, e.g. for  $T_2$ :  $\text{retVal}(s_1, +20_A) = \text{OK}$  and  $\text{retVal}(s_1, -20_B) = \text{OK}$ . This means that this execution is serializable, and therefore the original schedule is compatible and also serializable.

## 5.5 Local Coordination Avoidance (LoCA)

The other example with the same schedule, except  $B$ 's starting balance is also 0, results in a different execution. Note that Withdraw operations do not update the balance when not enough balance is available.

$$\left\{ \begin{array}{l} A \mapsto 0 \\ B \mapsto 0 \end{array} \right\}^{s_0} \xrightarrow{T_1} \left\{ \begin{array}{l} A \mapsto 10 \\ B \mapsto 0 \end{array} \right\}^{s_1} \xrightarrow{T_2} \left\{ \begin{array}{l} A \mapsto 30 \\ B \mapsto 0 \end{array} \right\}^{s_{12}} \xrightarrow{T_3} \left\{ \begin{array}{l} A \mapsto 0 \\ B \mapsto 30 \end{array} \right\}^{s_{123}}$$

Now the commit test for  $T_2$  fails:  $retVal(s_1, +20_A) = \text{OK}$  and  $retVal(s_1, -20_B) = \text{NOK}$ , which are different from the observed return values in the schedule. Other executions with different transaction orderings also fail the commit test. This means that this schedule is not RV-SER.

**RV-SER in TLA<sup>+</sup>** RV-SER is formalized in TLA<sup>+</sup> and confirms these examples. TLA<sup>+</sup> [69] is a formal specification language for action-based modeling of programs, algorithms and (distributed) systems [18, 41, 45, 75, 79]. TLA<sup>+</sup> models states and transitions and its accompanying model checker TLC checks properties on each state, providing counterexamples with error traces. This formalization enables checking sets of observed transactions, and validating if algorithms (tracking observed transactions) implement RV-SER. The source code and instructions on how to run are found on Zenodo [98]. In section 5.7 we see that RV-SER finds serializability bugs in a known serializable algorithm (Two-Phase Locking) when bug-seeded and validates LoCA which leverages CBC to be serializable.

## 5.5 Local Coordination Avoidance (LoCA)

The LoCA algorithm leverages CBC\* in a local object at run time, in order to increase concurrency and with that improve throughput and latency, while maintaining (return-value) serializability. It supports statically computed CBC pairs or computes CBC at run time based on current state, and effects and return values of operations using equation (5.1). LoCA is also implemented [100, 103] using the Akka actor framework, using 2PC for atomicity. LoCA is compatible with different consensus or atomic commit algorithms, such as Raft [82] and Paxos [68].

**LoCA in a nutshell** LoCA is an algorithm that can be locally run in a (distributed) object that receives commands to execute. It requires a conflict relation, such

**Table 5.3** Static commutative (scBC) of bank account operations. Static independency (*SIE*) values shown abbreviated in braces.  $E_1$  in rows,  $E_2$  in columns. ACCEPT (A) and REJECT (R) for *SIE* correspond to Go for scBC since scBC does not distinguish between direct accepts and rejects, because failing preconditions do not necessarily abort the transaction. DELAY corresponds with No.

scBC( $E_1, E_2$ )	Open	Deposit	Withdraw	Interest
Open	No	No	Go (R)	No
Deposit	No (R)	Go	No	No (A)
Withdraw	Go (R)	No (A)	No	No (A)
Interest	No (R)	No (A)	No	Go

as CBC, to determines when it is safe to run multiple operations concurrently. If objects need to synchronize with other objects LoCA uses the two-phase commit (2PC) protocol in order to assure atomicity: either all objects do the operation, or none. Two-phase locking (2PL) is used to ensure isolation, but different conflict relations result in different isolation guarantees. LoCA with CBC as conflict relation is (return-value) serializable.

When a LoCA object receives an operation, it starts a 2PC resource manager to handle communication with other objects. If other operations are already in progress, it first checks if the incoming operation is compatible (using the conflict relation) with already in-progress operations. If non-compatible the operation is delayed until compatibility is detected or all in-progress operations finish.

### 5.5.1 LoCA with Independent Events

In earlier work LoCA was used with (Statically) Independent Events ((S)IE) [100, 103]. In this chapter we maintain the global algorithm (and implementation) and swap in CBC in order to achieve serializability.

**Static CBC at compile time using SMT** A subset of CBC, dubbed static CBC (scBC), is independent of the current run-time state, e.g. deposits are always allowed, independent of the actual amount or balance. Determining scBC offline results in less computational overhead at run time. Table 5.3 shows static scBC and how values differ with *SIE* for a simple bank account example. Both are generated by leveraging an SMT-solver (z3 [76]) in which the resource’s preconditions, effects

and states are modeled (or generated from another specification), similarly to *SIE*'s analysis [100].

SCBC is a subset of CBC:  $SCBC(p, q) = \forall s. CBC(s, p, q)$ , denoting which operations are always CBC independently of a specific run-time state, corresponding to RVC. The SMT-solver finds these pairs of non-conflicting operations by searching for counterexamples where operations do conflict.

*SIE* is more lenient because it assumes that in-progress operations are valid on the resource (return OK). In order to maintain RV-SER, SCBC is more strict and also requires operations to be swapped without exposing different return values next to the OK or NOK of an operation. Also note that SCBC does not have any REJECT since it considers NOK to be just another return value.

Since this analysis is offline, LoCA can use the results to reduce computational overhead at run time.

## 5.6 Model Checking LoCA and RV-SER

In this section we look at two algorithms for synchronization and atomic commitment, their formalization in  $TLA^+$  and their conformance to RV-SER. We also seed bugs and wrong input models to validate that the model checker finds mistakes with counterexamples in section 5.7. This shows that LoCA is indeed RV-SER.

RV-SER is formalized similarly to CI [101] in  $TLA^+$ . The formalization of 2PL/2PC and CBC are structured similarly to the formalization of 2PL/2PC and Crooks' Isolation [23] in related work [101]. Full  $TLA^+$  source code is available online [98].

**RV-SER** Transactions are encoded as sequences of operations, which consist of operation types, parameters and observed return values.  $TLA^+$  module extensions enable modularization by extending different models, representing different objects with their own operations, internal state and effect functions. The RV-SER and CBC definitions only require  $RetVal(s,o)$  and  $Eff(s,o)$  functions to be present. RV-SER itself is a direct specification of commit test in equation (5.2). RV-SER is checked with the property:  $RVSerializability(InitialState, transactions)$ . This enables a) "unit testing" by model checking hard coded values and b) model checking of conformance for algorithms represented in  $TLA^+$  or PLUSCAL.

**2PL/2PC** The formalization of 2PL/2PC specifies two processes: the transaction manager and the resource manager. The transactions manager asks multiple resources to vote on an operation of the transaction. If all accept, the transaction manager tells the resources to commit the operation. If one of the resources voted abort, the manager aborts all operations in the transactions. This guarantees atomic commit: either all resources commit the transaction, or none. The assumptions are, without loss of generality, that messages between these resources are a monotonically growing shared set, meaning that they are never lost, but can be received out of order.

When a resource manager commits an operation, the operation with observed return value is tracked. The model checker TLC checks if the operations are valid under RV-SER for each execution state. It turns out this is indeed the case for models up to at least 3 transactions and resources.

**LoCA** The formalization of LoCA follows the same format as the 2PL/2PC formalization, except the resource manager can have multiple transactions in progress at the same moment in time, hence the improved concurrency. After handling messages, the resource processes the queued (committed) and delayed operations when applicable. Committed operations are tracked for RV-SER property validation by the model checker. LoCA only allows operations in parallel that pass the constructive CBC from property equation (5.1).

LoCA's pseudo-specification is found in listing 5.1 and follows the message contract of a 2PC resource manager. The main difference with 2PL/2PC is the simultaneous receiving of all message types of 2PL/2PC representing being in multiple transactions at the same time when operations are CBC\*. Variable operations tracks the observed operations per transaction, which are in turn checked to be RV-SER. The **either/or** construct denotes that any of these branches can occur when running the algorithm, which is important for defining the whole state space. The algorithm also branches at **pick** s.t., which picks a value such that the right hand side is true. Global variables are defined on top, where {} and [] respectively represent (empty) sets and sequences. Appending (+=) to sequences is at the end. Removing (-=) removes all instances of the element from the set or sequence.

LoCA's formalization is generic in the sense that it requires only two functions (Eff and RetVal) and InitialState available that capture domain knowledge. CBC\* uses these functions and follows equation (5.1). Multiple conflict relations can be configured, including *IE*, *SIE*, CBC and scBC.

**Listing 5.1** LoCA formalization

```

1  queued = {}; inProgress = []; delayed = [];
2  state = InitialState; operations = {}
3
4  while true:
5    # receive any of the 2PC messages
6    on receive of VoteRequest:
7      either # Either vote yes
8        o = pick s.t.  $CBC^*(state, inProgress, o)$ 
9        inProgress += o
10     reply VoteCommit(o)
11    or # vote no/abort
12     reply VoteAbort(o)
13    or # or delay until dependent operations finish
14     o = pick s.t.  $\neg CBC^*(state, inProgress, o)$ 
15     delayed += o
16    on receive of GlobalCommit(o):
17     queued += o # queue for commit
18    on receive of GlobalAbort(o):
19     inProgress -= o
20     delayed -= o
21
22    # Apply all applicable queued (ready for commit/apply) operations
23    while Head(inProgress)  $\in$  queued:
24     inP = Head(inProgress)
25     # track per transaction for RV-SER-check, including observed return value
26     operations[inP.tid] += <inP, RetVal(inP, state)>;
27     state = Eff(inP, state); # Apply Eff to state
28     inProgress -= inP
29     queued -= inP
30     # if next delayed is CBC, then start voting
31     while  $CBC^*(state, inProgress, Head(delayed))$ :
32       o = Head(delayed)
33       either
34         reply VoteCommit(o)
35         inProgress += o
36       or
37         reply VoteAbort(o)
38       delayed -= o

```

## 5.7 Initial Validation

**Bug seeding** In order to validate our definition of both CBC and RV-SER we introduce some bugs so that the model checker finds them in:

- the formalization of 2PL/2PC, where a resource could commit a different transaction than already voted for. The RV-SER property found an error trace, showing that it is capable of detecting non-serializability in algorithms.
- the formalization of LoCA with CBC rules for a bank account, and a bug where delayed (non-CBC) operations were not correctly aborted the model checker found an error trace where the resource processes terminate with still in-progress operations.
- the formalization of LoCA, when not checking for CBC-enabled operations, and thus allowing all operations to occur and vote instead of delay. RV-SER is violated and a counterexample is found, with a non-serializable schedule.
- the formalization of LoCA with static SIE, where it finds non-RV-SER traces for non-commutative operation pairs. This shows that SIE is not serializable and RV-SER detects this correctly.
- the formalization of CBC with  $s_{pq} \equiv s_{qp}$  left out and the account specification changed to track a list of previous transfers on Withdraw and Deposit. The checker finds a problematic example where the order of transfers is different on different accounts.

**2PL/2PC is RV-SER** To confirm the formalization of RV-SER we introduce a bug in the 2PL/2PC formalization, for which the model checker should find a problematic case. The formalization is serializable for read/write level operations [IOI], so if the same bug is also found by RV-SER, it gives us more confidence of its correctness.

The bug in the formalization allows resources to abort after voting for a different transaction. The counterexample exploits this by aborting an earlier accepted transaction, therefore violating atomicity. Eventually this leads to different resources committing to transactions in different orders. The RV-SER model check finds operations which expose values which are not observable under serializability.

**LoCA with CBC\* is RV-SER** The formalization of LoCA is model checked with a bank account instance for the RetVal and Eff functions. For small numbers of objects and transactions, it does not violate RV-SER, as designed.

When a bug is introduced where delayed operations are not correctly aborted, the model checker finds a counterexample where not all in-progress events are handled.

**LoCA without CBC\* is not RV-SER** When introducing a bug similar to the LoCA formalization, where resources can commit transactions not yet voted for, RV-SER finds a counterexample where a balance is returned by GetBalance that would not be visible in a serializable schedule. This strengthens our claim that RV-SER defines serializability and that CBC\* is sufficient for achieving serializability.

**LoCA with SIE is not RV-SER** In order to validate both CBC\* and RV-SER, LoCA is configured to use a conflict relation as defined by SIE (see table 5.3). The model checker finds an error set of transactions, containing a pair of Withdraw and Deposit operations. That are SIE, but not CBC\*, since the effect of an in-progress Withdraw never influences the acceptance of a Deposit. However, in order to be CBC it should also be possible to swap the operations without changing the return value. In this case a Deposit coming earlier can switch a Withdraw $\uparrow$ NOK to Withdraw $\uparrow$ OK, when availability of enough balance is dependent on the Deposit. The RV-SER property is sufficient to find such is problem. This gives us confidence that CBC\* indeed leads to RV-SER behavior and also that RV-SER is a sufficient for CBC, and thus serializability.

**CBC requires  $s_{pq} \equiv s_{qp}$**  The previous examples do not show the need for  $s_{pq} \equiv s_{qp}$  in CBC, since there are no later operations that read non-directly changed state. If the account operations also store the history of transfers in internal state and directly exposes this in the return value, this becomes problematic, because a future query operation can now expose this in a non-serializable fashion. In this case the history becomes ordered differently for different objects, which is not serializable. The found counterexample shows this.

## 5.8 Discussion

**CBC, RV-SER and models** CBC and RV-SER support use of higher-level operations or complete models, with as much tool support as possible. RV-SER enables automatic checking using model checkers of specific scenarios and models.

## Chapter 5 Contract-Based Return-Value Commutativity

A change in the modeling approach can have big impact in performance, e.g. a Covid vaccination appointment could be modeled as lots of separate timeslot objects, for which concurrency needs to be managed individually. Another modelling approach where timeslots are grouped per location and time and a counter of the total available slots at that time. This is similar to an AddRemove counter `CRDT`[85, 93], which does not require coordination. Tradeoffs in performance/modeling become explicit by analyses with (s)CBC. Now this tradeoff a business decision backed by data, instead of a problem of the implementers.

**Contract** CBC requires a quite strong contract on all operations with associated deterministic, side-effect-free functions. Applications have to be modelled in this sense to reap the benefits. In related work [100], a similar constraint is valid in up to 61% of operations for realistic use cases. We expect CBC to hold similarly.

**IE and RV-SER** Another conflict relation compatible with `LOCA` is *IE* [100, 103] ( $IE(s, p \uparrow \text{OK}, q) = \text{retVal}(s_p, q) \equiv \text{retVal}(s, q)$ ). *IE* however, is not serializable, since does not consider  $\omega$  or the swapped states, allowing different orderings on multiple objects, also for non-commuting operations.

`RV-SER` defines which instances of *IE* are non-serializable. However, a subset of *IE* operations, coinciding with CBC is still serializable, i.e. the commutative parts.

`LOCA` with `CBC*` computes more (just as *IE*) at run time than `2PL/2PC`, so this is really beneficial if waiting/blocking/locks become the bottleneck, and spare CPU power is available. Due to graceful degradation to `2PL/2PC` with no in-progress and non-CBC operations, performance is always on par or better than `2PL/2PC` in practice [100, 103].

All discussed conflict relations are related:  $SIE \subseteq IE$ ,  $CBC \subseteq IE$ ,  $SCBC \subseteq CBC$ ,  $SCBC \subseteq SIE$ . *IE* is the “most concurrent”. Each static variant is stricter than its dynamic counterpart.

Previous performance evaluations using (s)*IE* [100, 103] show that `LOCA` increases throughput and reduces latency in high-contention scenarios. Since `SCBC` has similar Go results, as shown in table 5.3, performance improvement of `LOCA` with (s)CBC will be on par with `LOCA` with (s)*IE*, since performance evaluation would follow the same pattern.

`LOCA` with CBC gives serializable isolation guarantees, which is closer to what modellers and business experts expect when working with modeling languages

such as Rebel. Subtle non-serializable behavior of `LoCA` with `IE` can be a problem for them. Also, one can model with serializability in mind, but swap out `CBC` for `IE` when more performance is needed and extra studying the specific behavior.

`SIE` [100] has two variants (`ACCEPT/REJECT`) in order to reduce overhead and increase parallelism at run time: either directly vote commit or abort an incoming operation. Since an abort vote directly finishes the transaction for that resource, it no longer has to consider this operation when handling other incoming operations. For some more domain specific use cases, such as Rebel [107, 108] for `SIE`, specialized static analyses based on some grouping of return values can be useful, but it is not generalizable.

Similarly to Adya's [1] and Crooks' [23] formalizations, `RV-SER` does only consider committed transactions. Transactions can abort by different functional (failing preconditions) and non-functional (deadlock, time-out, etc) reasons. One can argue that operations aborted for functional reasons, should still abort when swapped, but this is out of scope for this chapter and could be encoded in the return values of committed operations. `2PL/2PC` and `LoCA` only emit operations of committed transactions. Model checking with `TLA+` specification of return-value serializability does indeed find that all possible executions produced this way are serializable.

### 5.8.1 Threats to Validity

#### Limitations

This approach focuses on distributed objects that communicate via messages or methods and, to guarantee serializable isolation, requires that these methods are the only way to change and query the object state. This is a good fit for generating an implementation from higher level domain models, but might not be for low-level implementations, where extra care has to be taken to not break the abstraction.

**State space explosion** The model checking in the validation is run only on small model instances and on a single bank account example. The state space explosion that comes with larger model instances (more objects and transactions) make it unfeasible to model check due to time constraints. However, in line with the small scope hypothesis [56], we assume that most isolation violations can be found in small examples. Anomalies with larger error traces or complex interleaving of multiple objects, transactions and mixed use cases

might not be found with the current approach. The definition of `RV-SER` can be implemented separately to improve performance and thus increase feasibility.

### 5.9 Related Work

CBC is powered by contacts and models of objects. This fits well with Domain-Driven Design [31] and Command-Query Responsibility Segregation. More work [11, 17] is being done on reusing models to increase parallelism and performance.

**Coordination Avoidance, Confluence and CALM** Confluence[50] looks at observable behavior. A program is considered confluent if it produces the same set of outputs for all orderings of its input. Changes in the order of messages do not influence the observable outcomes, such as the return values. Invariant confluence is a necessary and sufficient condition for coordination avoidance [10]: a coordination-free execution. Non-invariant-confluent operations require coordination for correctness. This is similar to what `LoCA` with CBC guarantees: a subset of operations can be run concurrently without coordination. For non-CBC operations, coordination or delay is required to maintain correctness.

The CALM theorem [50] describes that monotonic programs only move forward, and never back. They do not need to retract any output. `LoCA` with CBC should never have to retract a value returned to a client and maintains them when operations are swapped internally. `LoCA`, `CBC*` and `SCBC`-tooling are a constructive approach towards CALM programs, including runtime performance optimizations.

Conflict-free Replicated Data Types [85, 93] are data structures that allow updates without coordinating. However, they are non-trivial to use, due to limited operations. `LoCA` with CBC allows programmers and designers to write models and code as they would normally and automatically enables high-performance where CBC allows this.

Observable Atomic Consistency [118], related to RedBlue Consistency [72], makes distinction between commutative and totally ordered operations. Commutative operations can interleave on different replicas, but as soon as total operation is requested the replicas synchronize and other (commutative) operations should wait. In a sense, `LoCA` with CBC achieves the same by allowing commuting operations to swap and interleave. It differs in providing both

run-time and static approaches to automatically detect this. *LoCA* does not focus on replicas at the moment, but could be extended to support this.

## 5.10 Conclusion

Data consistency and performance are a trade-off in many cases. Coordination is required to keep data in sync. However, commutative operations can be done without coordination, because the order of operations does not influence the resulting state and observed return values.

This chapter focuses on return-value commutativity, which looks at the client-perspective of higher level operations. Swapped operations should return the same return values when operations are executed in different order. If this is the case, non-serializable schedules of lower level read/write operations, are serializable on a higher level, because swapped operations are identical from client perspective to a serializable schedule. This insight enables using invariants from higher level operations to allow more schedules and with that improve throughput and latency.

We propose Return-Value Serializability (*RV-SER*), a definition of higher level operations, which defines when a schedule is serializable with respect to the observed return values. Next to that we define, Contract-Based Commutativity (*CBC*), an implementable definition leading to *RV-SER*. The Local-Coordination Avoidance (*LoCA*) algorithm uses an optimized constructive variant (*CBC\**) of *CBC* to allow swapping of *CBC* operations at run time, which results in improved parallelism where possible. This leads to reduction in high-contention bottlenecks, which increases performance and reduces latency. *RV-SER* and *LoCA* are formalized in  $TLA^+$  and validated by seeding bugs, which are detected by a model checking.

Static *CBC* (*SCBC*) is the subset of statically determinable *CBC* operations, e.g. depositing money can always done in parallel. An algorithm can use this information to shortcut potentially expensive dynamic *CBC* computations at run time. *SCBC* is determined for a set of operations by using an SMT solver. We compare *SCBC* for a bank account example to a non-serializable conflict relation *SIE*. The  $TLA^+$  formalization also detects non-serializability of *SIE* and confirms serializability of static and dynamic *CBC*.

Commutativity-based rescheduling of higher-level operations is often discussed, but not often used in practice, because it requires (manual) defining of the conflicting operations. Our approach enables automatically deriving

## Chapter 5 Contract-Based Return-Value Commutativity

of conflicting operations at both run and compile time and we believe this is a sweet-spot between over-specifying and error-prone manual specifying of conflicting operations. It also lowers the bar for model driven approaches for distributed objects, where modellers write intuitive model based on serializable isolation semantics, and our tools can optimize for speed when it is safe.

# 6

## Design and Architecture

**Abstract** All contributions in this thesis and evaluations thereof are backed by running software.

The generated runtime `rebel-runtime-lib` was created for Rebel. The Path-Sensitive Atomic Commit and the Local-Coordination Avoidance algorithms are part of this runtime. The generated application is reactive, event sourced and actor based. It supports high-availability, horizontal scalability, and enterprise-grade persistence.

Performance and scalability evaluations are done in a public cloud, using a fit-for-purpose, but generalizable experiment runner.

The `rebel-conflictors` toolkit was created to automatically determine both Statically Independent Events and Static Contract-Based Commutativity for Rebel specification using the `Z3` SMT solver.

And finally  $TLA^+$  formalizations of Contract-Based Commutativity and Return-Value Serializability were created to model check the isolation guarantees of the novel algorithms.

### 6.1 Introduction

This chapter introduces and explains the different software components that are used in the research conducted in the rest of the thesis.

- `rebel-runtime-lib`: A Rebel [105] runtime based on Akka [3] actors, implementing the `LOCA` algorithm
- `ing-rebel-generators`: Code generation module in Rascal for generating `rebel-runtime-lib` code from Rebel specifications. The translation is straightforward because `rebel-runtime-lib` does the heavy lifting.
- `rebel-conflictors`: Automated derivation of non-conflicting operations using SMT solver `Z3`.

- rebel-sie: Automated *SIE* derivation for Rebel specifications
- rebel-cbc: Automated CBC derivation for Rebel specifications
- experiment-runner: Experiment runner for reproducible scalability experiments on Amazon Web Services (AWS)
- isolation-specs:  $TLA^+$  formalizations of CI, CBC and RV-SER

Figure 6.1 shows how the different software components interact. External Rebel specifications are input for `ing-rebel-generators`, which generates `rebel-runtime-lib` implementations and an `experiment-runner` based on the specifications. The `experiment-runner` runs performance benchmarks based on external experiment configuration on the generated implementation and reports the results. The same specifications are analysed by the `rebel-conflictors` using SMT solving to determine which operations are non-conflicting. The resulting *SIE* and *scbc* operation pairs serve as input to the *LoCA* and *PSAC* algorithms implemented in `rebel-runtime-lib`. Output execution traces of the `rebel-runtime-lib` can be analysed with the  $TLA^+$  formalization of *RV-SER* in `isolation-specs` to analyse isolation properties.

Table 6.1 gives an indication of the size of the different components in lines of code, their licenses and where to find the source code. `ing-rebel-generators`, `rebel-sie` and `rebel-cbc` are implemented using the *RASCAL* meta-programming language [65].

## 6.2 Distributed Actors in `rebel-runtime-lib`

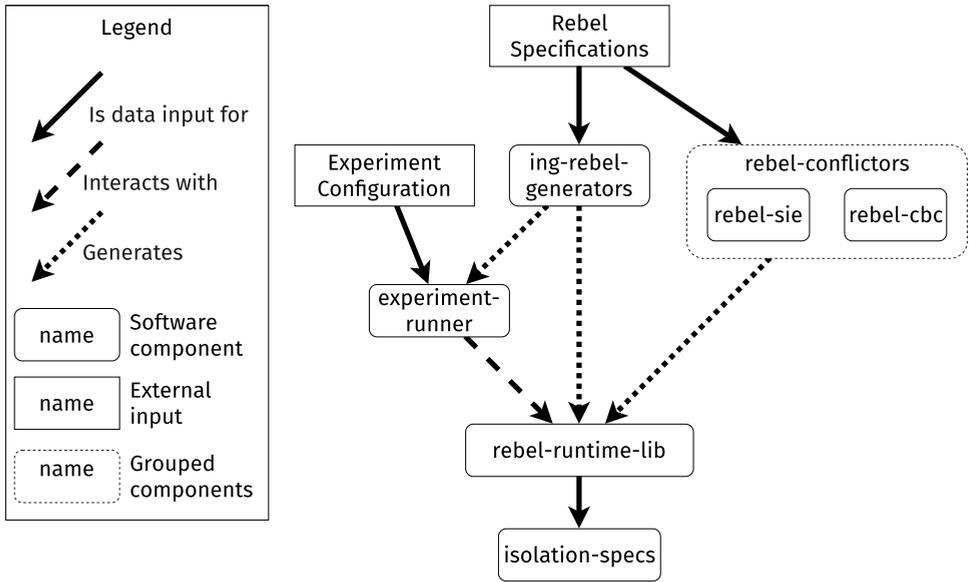
This section focuses on `rebel-runtime-lib`. This library package contains multiple important parts: a runtime for rebel specifications based on actors, build in a type-safe fashion in Scala. It also contains the *PSAC* (chapter 2) / *LoCA* (chapters 3 and 5) implementation using the domain knowledge from the Rebel specifications. *LoCA* supports multiple (and combinations of) conflict relations *IE* (chapter 2), *SIE* (chapter 3), *CBC* (chapter 5) and *scbc* (chapter 5). Part of the package is also the experiment runner and a consistency checker.

**Architecture** The implementation follows the the state machine models with events similar to Rebel specifications. Each rebel specification is a state machine class in Scala [81], which needs to specify the state machines lifecycle, the events and their guards and effects. A generic base class – capturing the

**Listing 6.1** Simplified interface trait that describes the Rebel specification and contract.

```

1 trait RebelSpecification[S <: Specification] {
2   type Spec = S
3   type RState = S#State
4   type RData = S#RData
5   type RDomainEvent = S#RDomainEvent
6   type SpecEvent = S#Event
7
8   def initialState: RState
9   def allStates: Set[RState]
10  def finalStates: Set[RState]
11  def initialData: RData = Uninitialised
12
13  /** Given current state and command, determines reachable state. */
14  def nextState: PartialFunction[(RState, SpecEvent), RState]
15
16  /** Guards of the contract */
17  def checkPreConditions(data: RData, now: DateTime)
18    : PartialFunction[SpecEvent, RebelConditionCheck]
19  /** Local effects of the contract */
20  def applyPostConditions(data: RData, domainEvent: RDomainEvent): RData
21
22  /** @return a set of other objects/specification involved in operation. */
23  def syncOperations(now: DateTime, transactionId: TransactionId)
24    : PartialFunction[(SpecEvent, RData), Set[SyncOperation]]
25
26  /** A function that takes an in-progress event and an incoming event, and returns if the incoming
    ↳ event can safely be started without checking pre-conditions. Generated by rebel-
    ↳ conflictors.
27  * @return true if incoming event can be started without checks */
28  def alwaysCompatibleEvents: PartialFunction[(S#Event, S#Event), Boolean]
29
30  /** A function that takes an in-progress event and an incoming event, and returns if the incoming
    ↳ event can be directly declined without checking pre-conditions. Generated by rebel-
    ↳ conflictors.
31  * @return true if incoming event can be declined fast without checks */
32  def failFastEvents: PartialFunction[(S#Event, S#Event), Boolean]
33 }
```



**Figure 6.1** Abstracted data flow graph of the software components presented in this chapter. Dashed boxes mark grouped components, straight edged boxes external parties and rounded boxes are software components.

contract – makes sure that runs as a persistent actor in the Akka cluster, and responds to its defined operations. Rebel runtime actors have to implement this. `ing-rebel-generators` automates this for Rebel specifications. Listing 6.1 shows a simplified interface with definitions for guards, effects and static non-conflicting operations (`alwaysCompatibleEvents` and `failFastEvents`). An implementation of this interface for a bank account is found in appendix A.2. Note that the interface is generically set up, so Rebel specifics can be factored out when needed.

The deployment and implementation follows a reactive architecture [16, 26] Instances of the state machine classes are run as actors on top of the Akka actor toolkit [3]. Each actor instance is a running lightweight – non-os – process which sends en receives messages.

Actors [53] are a useful abstraction in distributed systems, popularized by the Erlang programming language [7]. Each actor runs somewhere on a cluster of servers, and is reachable by a virtual address. This means that changes in deployments of actors are transparent to their callers. The message passing approach makes sure that delays and out of order delivery of messages (due to

**Table 6.1** Software Components with metadata: Programming Language, Source Lines of Code (SLOC) excluding comments, License, Github URL/Zenodo DOI.

Component	SLOC	Programming Language	License	Zenodo
rebel-runtime-lib		Scala	CC BY 4.0 <sup>a</sup>	[96] <sup>e</sup>
– core	2950	Scala		
– tests	2629	Scala		
– benchmarks	1827	Scala		
– performance	785	Scala		
ing-rebel-generators <sup>b</sup>	652	Rascal	Proprietary	N/A
rebel-sie	844	Rascal	CC BY 4.0 <sup>a</sup>	[96, 98] <sup>c</sup>
rebel-cbc	1336	Rascal	CC BY 4.0 <sup>a</sup>	[98] <sup>c</sup>
experiment-runner	1609	Scala	CC BY 4.0 <sup>a</sup>	[99] <sup>e</sup>
isolation-specs	1041	TLA <sup>+</sup>	CC BY 4.0 <sup>a</sup>	[97, 98] <sup>cd</sup>

<sup>a</sup> Creative Commons Attribution 4.0 International: <https://creativecommons.org/licenses/by/4.0/>

<sup>b</sup> Only code used directly for rebel-runtime-lib generation is counted.

<sup>c</sup> <https://github.com/cwi-swat/cbc-artifacts>

<sup>d</sup> <https://github.com/cwi-swat/tla-ci>

<sup>e</sup> <https://github.com/cwi-swat/rebel-runtime-lib>

the inherent underlying transport layer) are made explicit in the programming model. Due to this distributed nature, actors can be deployed over a cluster of servers – often called nodes. This enables scalability, by allowing them to spread out over a cluster of nodes.

The actors are made resilient to failure by employing an event sourced architecture: Each state changing operation is persisted to a distributed database, in order to allow seamless recovery of said actor on another node, without data loss. Our implementation makes use of builtin modules of the Akka toolkit for this.

This architecture is the foundation of a scalable, resilient, enterprise-grade implementation [3, 17, 26, 83].

**Synchronized Operations** On the Rebel level, instances of state machines communicate by synchronized events. These events are only enabled when other

referenced state machines' events are also enabled. All of these events should happen in the same atomic step or none should happen. An example is the Book operation of a **MoneyTransfer** state machine. Book synchronizes with Deposit and Withdraw on respective the receiving and paying account.

At the implementation side this requires two ingredients: Concurrency control – to make sure multiple operations do not lead to invalid state – and an atomic commitment algorithm – to make sure all involved objects do the operation or none. *Operations* are the run-time equivalent of the *events* on the specification side.

Concurrency control is handled by the Two-Phase Locking (2PL) protocol, which guarantees serializable isolation. 2PL uses locking to make sure no concurrent operations are done on an object. An object can only be locked by one operation, so this ensures that an object is only accessed by a single operation at the same moment, and no other operations access or change state that would make the guards of the in-progress operation invalid.

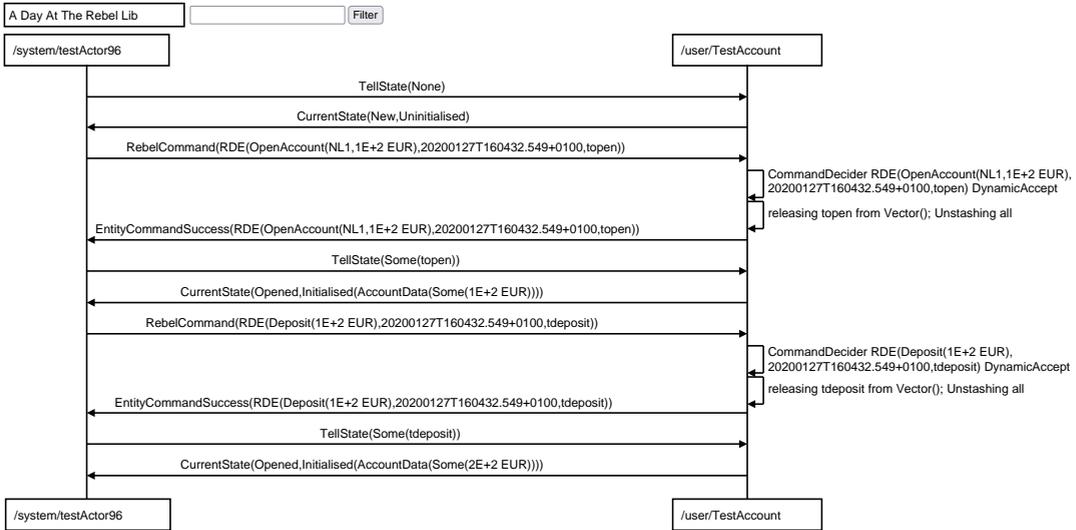
For atomic commitment the implementation uses a well-known blocking protocol, Two-Phase Commit (2PC). In 2PC all involved objects first vote to commit and later commit and persist the updates. This makes sure that none or all operations will do the operation.

The implementation combines these two protocols to implement ACID [46] transactions. 2PL to disallow multiple operations on an object and 2PC to make sure all involved objects allow and do their respective operations. When an object votes for the 2PC transaction, it locks itself via 2PL. Upon 2PC commit or abort the 2PL lock is released. For durability (D in ACID) all state changing operations are persisted to data store to make sure the internal state can be recovered in event of failure. See section 2.4.2 and appendix A.2 for more info on this.

In the case that an operation synchronizes with more objects, nested tree-based 2PC [115] is used. This adds the extra involved objects to the transaction and they should also vote before the commit is issued. The implementation of `syncOperations` (in listing 6.1) captures the involved objects.

**LoCA as concurrency control** Instead of 2PC as concurrency control, which only allows a single operations to be active at the same time in an object, this dissertation provides LoCA which enables multiple operations to be when they are non-conflicting. Multiple variants of this are presented: Independent Events (chapters 2 and 3) and Contract-Based Commutativity (chapter 5). Listings 2.1 and 5.1 presents LoCA's pseudocode and a formalization.

## 6.2 Distributed Actors in rebel-runtime-lib



**Figure 6.2** Message Flow Visualizer showing a sequence diagram of a simple unit test of rebel-runtime-lib.

LoCA checks the incoming operation against each of the in-progress operations. Then it starts a 2PC transaction participant for each operation that the non-conflict relation. Multiple different non-conflict operations can also be chained. This way first the cheaper static variants can be used, to later fall-back to the more expensive dynamic versions when conflicting.

**Consistency checker** Since the implementation uses an event sourced approach, all operations are captured in an event log. Using the Rebel semantics such a log of operations can be checked. The event log is replayed in order and guards and effects are checked. This way any violated by smartly interleaving operations differently can be checked against the local object lifecycle. In earlier days of the implementation design this gave the necessary confidence of correctness of the implementation approach.

**Message Flow Visualization** rebel-runtime-lib also includes a web based visualization of actor messages, displaying a filterable sequence diagram. An example of an open account command on an account is shown in figure 6.2. This is useful when debugging bugs and retrace origin of messages. Many unit tests contain diagram output for debugging and extra manual verification of correctness.

### 6.3 Experiment Runner

The experiment runner powers the scalability and cloud performance evaluation of chapter 2. This tool runs experiment scripts in a reproducible fashion on the Amazon Web Services cloud infrastructure and captures the relevant logs, metrics and other artifacts created.

The features of the experiment runner include:

- Experiments defined in `YAML` (using `HOCON`<sup>1</sup>).
- Inheritance of experiment definitions
- Automated deployment of application and dependencies (based on `Docker`) and running of experiment
- Overriding of properties via command line arguments
- Exponential back-off when waiting for `AWS` to catch up, for example when starting containers, or when being rate limited.
- Fetching of `AWS` logs to text files and `InfluxDB` data to `csv` files
- Fetching of file system resources after experiment, such as `Java Flight Recorder` files, and `Gatling` performance run reports
- Spreading containers over available server instances and starting the next experiment as soon as servers became available, either by finished previous experiments or newly added ones
- Headless operation, for overnight runs with error recovery
- Specific where needed (`Rebel`, `Java Flight Recorder`, `InfluxDB`, `JMS` metrics and `AWS` integration), generic where possible.
- Automated aggregation of `Gatling` performance load runs and running of `R` reports
- Usage of `Spot` instances and automated decommissioning of server instances to reduce cost.

Listing 6.2 shows an example configuration file for the experiment runner. Configuration items are inherited from included experiments. Local values override inherited ones. This experiment runs `Gatling` performance load runner with the `AllToAllSimulation` class.

The experiment runner generates multiple experiment runs from a single scenario configuration file. Two variants are defined, these are compared to each another. Everywhere where lists (`[ X ]`) are used it is possible to mix and

---

<sup>1</sup> <https://github.com/lightbend/config/blob/master/HOCON.md>

**Listing 6.2** Example experiment configuration file

```

1 include "SyncAndNoSyncXNClosed.conf"
2 batch {
3   description = "2PCvsPSACClosedContention1000"
4   default = {
5     simulation-classes = ["com.ing.corebank.rebel.simple_transaction.closedsystem.
      ↪ AllToAllSimulation"]
6     performance-throttles = [500]
7     user-counts = [250]
8     durations = [10m]
9     performance-configs = [{
10      rebel.scenario.number-of-accounts = 1000
11    }]
12  }
13  n = {
14    cluster-sizes = [1, 3, 6, 9, 12]
15  }
16  variants = [
17    {
18      description = "10002PC"
19    },
20    {
21      description = "1000PSAC"
22      app-configs = [{
23        rebel.sync.max-transactions-in-progress = 8
24      }]
25    }
26  ]
27 }

```

match variants. In this example multiple cluster-sizes are defined, so different independent experiment runs with different numbers of application servers are run.

## 6.4 rebel-conflictors: rebel-sie and rebel-cbc

Both *SIE* and *SCBC*, discussed in chapters 3 and 5, are static variants of *IE* and *CBC*. This means that operations are independent for all possible run-time

states. Our tools, based on SMT solving, use the Rascal meta-programming language and the z3 SMT solver. Part of the solution is a conversion from Rebel specifications to SMT-lib clauses.

For a specified Rebel state-machine definition, SMT code is generated and assertions for *SIE* or *SCBC* are added. This reuses code for simulations using SMT by Stoel's Rebel [105, 108]. The different non-conflicting relations are defined as SMT properties. The z3 SMT solver is iteratively queried for all pairs of operations for that state machine. The result is presented in a table or as a Scala partial function taking operation combinations to be used in the generated code.

The source code is available on Zenodo [96, 98] and on github.<sup>2</sup>

### 6.5 Verifying Isolation in TLA<sup>+</sup> with isolation-specs

The design of the TLA<sup>+</sup> specifications for *CBC* and *RV-SER* is modular. Both of course share the contracts. Different files define the object's contract, the *LOCA* and *2PL/2PC* algorithms, examples and *RV-SER* definitions. They are combined via inheritance and model checking configuration. Configuration options consists of number of participants and transactions, and algorithms and non-conflict relation used (*SIE*, *CBC*, a faulty version or combinations thereof).

The object's contract is specified in a separate file and needs to contain functions for the return values and effects given an operation and state. Optionally they also define *SIE* and *SCBC* operations pairs. Both the definitions for *RV-SER* and *LOCA* use this contract to respectively determine if return values are valid in a serial ordering, and to start concurrent processing.

TLA<sup>+</sup>'s model checker TLC can used to verify if an execution – a sequence of operations on multiple objects – is *RV-SER*. Both manually entered executions, executions generated by an implementation or executions when model checking *LOCA* can be checked.

The source code is available on Zenodo [97, 98] and github.<sup>3</sup>

---

<sup>2</sup> <https://github.com/cwi-swat/cbc-artifacts/tree/main/SCBC>

<sup>3</sup> <https://github.com/cwi-swat/tla-ci> & <https://github.com/cwi-swat/cbc-artifacts>

## 6.6 Summary

All of these software component support avoidance of coordination in a direct or indirect way. `rebel-runtime-lib` contains the implementations of `LoCA` and `PSAC` algorithms and an experiment runner for reproducible scalability and performance experiments on cloud infrastructure. `LoCA` supports different non-conflicting operations and combination thereof:

- run-time variants *IE* and `CBC`, defined in `rebel-runtime-lib`;
- compile-time variants *SIE* and `scBC`, which are determined by the `rebel-conflictors`, which determine – using SMT solving – the operations pairs that are always compatible to run concurrently. `rebel-sie` and `rebel-cbc` generate these pairs for a state machine definition containing the contract of guards, effects and return values.

`ing-rebel-generators` contains a code generator which translates Rebel specifications to a `rebel-runtime-lib` program. The runtime is based on the Akka actor toolkit, which provides building blocks for a scalable and resilient implementation. To validate the isolation guarantees of the different `LoCA` variants, `isolation-specs` contains  $TLA^+$  formalizations of `CBC` and `RV-SER`. This is used to model check one-off execution traces, or check conformance to `RV-SER` of algorithms, such as `LoCA` and `2PL/2PC`.





## Conclusion

Enterprise IT landscapes are complex and hard to maintain. Creating resilient and available applications is also non-trivial. Communicating distributed components are a prerequisite for resiliency and availability, but also for low latency for global users. Simply, because hardware or software breaks down and information takes time to travel over the globe. Model Driven Engineering and Domain Specific Languages are a way to manage this complexity, by separating business logic from implementation. Creating an implementation from such models is still hard, however correct by construction implementations can be created leveraging contracts captured in the models.

This chapter revisits the research questions from chapter 1 and consolidates the answers from the different chapters.

### 7.1 Research Questions

RQS 1 to 3 focus on similar but different aspects: Firstly, the focus is on determining how coordination can be avoided. Secondly, how this can be done at run time, and thirdly how parts of this can already be preprocessed at compile time.

### 7.1.1 RQ 1: Local Coordination Avoidance with Independent Operations

#### Research Question I (RQ I):

When can coordination between two operations be avoided without violating isolation and consistency requirements?

This question focuses on the requirements for avoiding coordination. This question has multiple answers based on two different notions of correctness: the local object consistency and the global isolation requirements. First we looked at a generic sufficient conditions.

Consistency here means that the local objects invariants are always adhered to. The objects in this research are always defined with a contract, consisting of *guards* on which operations are allowed and *effects* which side-effect-free compute the next local state. State machine formalizations, such as the domain specific language Rebel, are compatible with this contract definition, where the guards encode both the state transitions and the enabledness – whether an operation is allowed in the current state – of operations.

**Local Consistency** Chapter 3 defines the Independent Events (*IE*) property, which is a relation of two operations on an object. Two operations are independent if and only if the outcome of the the first operation does not change the validity of the second. Effectively, this means that independent of the actual commit or abort of the first, the second operation stays accepted or rejected. In other words: Does the eventual abort or commit of the in-progress operations influence the guard's outcome of a new incoming operations? If not, then the incoming operations can also already start, because it never has to be retracted, e.g. a withdrawal on a bank account can already start if there is enough balance available independent of an already in-progress withdrawal's or deposit's effect.

*IE* guarantees local linearizable consistency – separate operations adhere to a strict serial order and local objects always adhere to their invariants as defined by the contract.

**Global Isolation** For isolation guarantees ordering of operations on different objects are also taken into account, next to the single object being consistent with respect to its own lifecycle and contract. Chapter 5 defines the Contract-Based Commutativity (*CBC*) property which defines when operations on a

single object can always be reordered, without influencing observable behavior in the form of return values. A reordering never results in retraction of an observed return value. When all objects only run operations concurrently when CBC holds, the operations can always be reordered into an equivalent serializable schedule.

CBC is formalized in TLA<sup>+</sup> in chapter 5. This enables automatic verification of traces of operation schedules, either from implementations or manually entered schedules.

### 7.1.2 RQ 2: Local Coordination Avoidance at Run Time

#### Research Question 2 (RQ 2):

How can coordination avoidance between two or more operations be achieved at run time?

All non-conflicting relations *IE* and CBC, mentioned at the previous research question, are used by a run-time algorithm Local Coordination Avoidance (LoCA). If the relation holds the second operation can already start processing, since its acceptance or rejection will never have to be retracted in the future. This is exploited by the Path-Sensitive Atomic Commit (PSAC) (chapter 2) and LoCA (chapters 3 and 5) algorithms at run time. PSAC is a version of LoCA where *IE* is used for the non-conflict relation.<sup>1</sup>

At run-time, LoCA uses the property to determine if a new incoming operation is conflicting with any of the in-progress operations. Conceptually it creates a tree of possible outcomes, where each path from root to leaf represents a possible outcome, representing all possibilities of in-progress operations either aborting or committing.

<sup>1</sup> Although PSAC internally uses two-phase commit for *Atomic Commit*, this is not where the algorithm's contribution lies. PSAC (and LoCA) use two-phase locking for concurrency control, with the difference is not locking for non-conflicting operations.

### 7.1.3 RQ 3: Local Coordination Avoidance at Compile Time

#### Research Question 3 (RQ 3):

How can coordination avoidance between two operations be determined at compile time?

LOCA requires a non-conflict relation, such as *IE* and *CBC*. However, the algorithm is not concerned with the precise computation thereof.

*IE* and *CBC* follow the function signature  $State \times Operation \times Operation \rightarrow Boolean$ . However a short-cut can be made when  $Operation \times Operation \rightarrow Boolean$  returns the same for all possible states. This allows an implementation to do less computation, because it does not have to calculate *IE* or *CBC*.

**Statically Non-Conflicting Operations** Both *IE* and *CBC* have a static counterpart defined in their respective chapters 3 and 5. Chapter 3 expands on this by placing *IE* in a larger frame of independent (*IE*) events and its subset statically independent events (*SIE*). Statically Independent Events (*SIE*) are a subset of *IE*, for which the actual run-time state does not influence if operation pairs are *IE* or not, e.g. deposits can always run concurrently since their guards always hold independent of the run-time balance. Statically Contract-Based Commutativity (*SCBC*) is a subset of *CBC*, with the similar rule that run-time state does not influence adherence to *CBC*. These different properties relate as depicted in the Introduction, figure 1.3.

The *rebel-conflictors Toolkit*, based on SMT solving, computes *SIE* and *SCBC* operations for a specific object with contracts. This pre-computing at compile-time reduces the computational costs at run time.

### 7.1.4 RQ 4: Performance Benefits in High Contention Scenarios

#### Research Question 4 (RQ 4):

What are the performance benefits for local coordination avoidance and in which scenarios do they hold?

Chapter 2 contains an elaborate performance evaluation based on scalable cloud infrastructure. Details on the implementation and experiment runner are found in chapter 6. LOCA with *IE* reaches up to 1.6 times more throughput

than 2PL/2PC in high-contention scenarios. Latency and throughput are always on par or better compared to 2PL/2PC, also in low-contention scenarios.

A performance evaluation on a smaller scale for *SIE* is found in chapter 3, corroborating the larger scale results from chapter 2. Since CBC is a subset of *IE*, it is expected that performance improvements for LoCA with CBC operations are similar to the performance results found for LoCA with (*S*)*IE*. However, this remains to be confirmed in future work.

Chapter 3 contains a preliminary study of how often *SIE* operation pairs occur in real applications. It found that up to 61% of operation pairs are statically independent for an ING payments use case and the TPC-C benchmark. This is promising for all kinds of real world applications. In practice even more operations are *IE* and CBC than compile-time *SIE*.

### 7.1.5 RQ 5: Isolation Guarantees

#### Research Question 5 (RQ 5):

What are the isolation guarantees when using the different non-conflict relations, and how do they relate?

In chapter 2 we have shown that PSAC and *IE* are not serializable. The local lifecycle of the object is always consistent and correct, but non-serializable behavior can occur when considering groups of objects. Operations can arrive in different orders and due to the increased concurrency it can occur that operations of the same transaction are handled in different order on different objects, e.g. a transaction overview list differs in order or different account balances occur when interest is applied in different orders. Chapter 2 provides a non-serializable example, and chapter 5 reproduces such a counterexample using model checking with  $TLA^+$ .

We created a model and formalization of serializability based on higher-level operations based on Crooks' [23] and Weikum's [114] insights that schedules of operations are equivalent to serializability when looking respectively at the observed outcomes and higher level operations. As the first step chapter 4 we formalized Crooks' isolation model [23] in  $TLA^+$ . This model is based on low-level reads and writes. The next step in chapter 5 extends this formalization with higher-level operations from the contracts. Looking at high-level operations allows more possible orders of operations to conform to serializability as long as their observable behavior does not change, e.g. for multiple deposits which

only return success it does not matter what the exact in between balance is, as long as they stay successful and the final balance is correct. This is formalized as Return-Value Serializability (RV-SER) in  $TLA^+$  in chapter 5. CBC is a sufficient condition for RV-SER, so LoCA with CBC implements serializability, while also increasing throughput and improving latency.

## 7.2 Discussion and Further Directions

In conclusion, when contracts are available, LoCA (with CBC) is a drop in replacement for other concurrency control mechanisms. With CBC it provides serializable isolation guarantees, and with PSAC linearizability for the objects. Its performance is on par with 2PC – or when using other atomic commitment protocols, at least as good as those – and improves in high-contention scenarios with CBC operations.

LoCA itself and the statically non-conflicting operations toolkit rebel-conflictors can act as a indicator where bottlenecks occur.

The research conducted and its evaluation scripts and result data, and source code – where possible – are available as Zenodo artifacts [95, 96, 97, 98] for transparency and reproducibility purposes.

### 7.2.1 Implications for Research

One of the major challenges for current distributed applications – which almost all applications are in this cloud era – is performance and responsivity. Without coordination fast and local decisions can be made and application can scale independently. The CALM theorem [50] defines when coordination is not necessary. However it is still an open question on how to do this. This dissertation describes a path towards this. Models and contracts of applications are analyzed to (automatically) show where coordination is not required. For the places where coordination is necessary, the requirements and contracts can either be changed to not require it anymore or considered necessary evil for this problem domain. Either way, it becomes clear when programs can be sped up by avoiding coordination.

This research focuses its application on distributed objects and actors, but can also be applied in microservice architectures and other middleware solutions.

Many model driven engineering approaches may very well already capture enough information for the contracts required by `LoCA`. This means that `LoCA` can be embedded in implementations or generated code for these models, and can benefit from the improved throughput and latency.

This research presents `LoCA`, which used `2PC` for atomicity and `2PL` for concurrency control. Its novel part lies in replacing `2PL` with less strict locking when operations are compatible. An implication for Paxos [68] and other consensus protocols is that this less strict locking also be applied in their concurrency control mechanisms. The concept of non-conflicting operations not limited to `2PL` and `2PC`.

### 7.2.2 Implications for Practitioners

Coordination Avoidance formalizations, such as Return-Value Serializability and Contract-Based Commutativity, and implementations, such as `LoCA` or static analysis tools, such as the rebel-conflictors, can be used to detect potential scalability bottlenecks in applications. This gives a low-entry bar for practitioners to detect these in their applications to either change the program to remove the bottlenecks, or avoid them at run time.

Applications can partially be modelled with contracts – ideally the most communicating parts. On the borders between, more conventional approaches for concurrency control that fall back to more coordination can be used.

`LoCA` can also be embedded in various low code platforms, where contracts are often already available in some form.

Microservice architectures where each microservice has its own data store needs to guarantee isolation at the application level, either via locking with `2PC` or using the saga pattern [35]. `LoCA` can be implemented for distributed transactions involving multiple microservices and help to improve throughput by not blocking on non-conflicting operations.

The introduction section 1.2.1 discusses an example where a performance bottleneck is solved by modeling the business domain differently. A high-contention tax bank account is kept available using a shadow wash account, which does all the small transactions in a batch to stay within duration limits. These kinds of model changes are not always necessary when using an algorithm such as `LoCA`, which automatically takes non-conflicting operations off the critical path. The added benefit is that ad-hoc bugs by embedding concurrency control in the model are prevented, e.g. keeping the shadow account's blocked/unblocked

## Chapter 7 Conclusion

status in sync with the original account's status. This means less specialized changes are required in models for performance reasons and the models only concern actual business functionality.

### 7.2.3 Further Directions

Since this dissertation is conducted in the context of ING Bank, this section focuses on research outcomes and applications relevant within the financial industry.

This work can function as a stepping stone into domain expert tooling which detects potential high-contention performance bottlenecks. By analyzing the models and contracts of applications, it detects where operations are not statically independent or contract-based commutative. Together with (expected) load data, bottlenecks can be identified and a different modeling strategy can be used.

A large gain from `LOCA` lies in the run-time variant, since more information is available. This depends on the run-time state, so some bottlenecks determined at compile using the static variants, are not a problem in practice. The real bottlenecks can be detected by instrumenting the runtime or capturing metrics on how often coordination can and can not be avoided. e.g. the earlier example from section 1.2.1 where big batch transactions of a tax account are taken off the critical path by using a wash account.

**Further Research** Some important open research questions are validation on a larger set of models with contracts. Another direction is determining of other isolation levels than serializability (with `CBC`). In many use cases isolation levels with less guarantees are good enough.

Related to this is determining fail isolation guarantees on higher level operations for `PSAC` and `IE`. Strictly looking at reads and writes only, `PSAC` guarantees only read uncommitted, which is low in the hierarchy of isolation guarantees. In practice `PSAC` always guarantees that local objects never read uncommitted values on the contract's operations levels.

`LOCA`'s design is based on single objects for which local decisions are made in a single replica. In order to increase availability and scalability, and reduce response times, a multiple replicas approach should be made. This means that replicated objects need to be kept in sync. For commutative operations this

can be eventual, but for other operations not, depending on the consistency requirements. This is related to RedBlue consistency [72] and CvRDTs [118].

**Further Implementation** For adoption of `LoCA` and tooling, a small dedicated implementation library should be created. Application teams can embed this in their implementation. The contracts can then be specified as annotations on the operations or in an embedded domain specific language.

`LoCA` can be used on a small scale in a small application. This way some production application can start using this. Validation of `LoCA` at scale will be helpful in its adoptance. For this a reusable stand-alone library should be created, which only contains `LoCA` and contract abstractions. The current implementation is coupled to a Rebel runtime, but set up modularly, so extraction and reuse is possible.



# Bibliography

- [1] Atul Adya. “Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions”. en. PhD thesis. USA: Massachusetts Institute of Technology, 1999, page 198 (cited on pages 6, 10, 84, 92, 105, 111, 121).
- [2] Marcos K. Aguilera and Douglas B. Terry. “The Many Faces of Consistency”. In: *IEEE Data Eng. Bull.* 39.1 (2016), pages 3–13. URL: <http://sites.computer.org/debull/A16mar/p3.pdf> (cited on page 104).
- [3] Akka. 2018. URL: <https://akka.io> (visited on 2018-12-21) (cited on pages 37, 72, 125, 128, 129).
- [4] James C. Corbett et al. “Spanner: Google’s Globally Distributed Database”. In: *ACM Trans. Comput. Syst.* 31.3 (2013), 8:1–8:22 (cited on page 78).
- [5] Peter Alvaro. “Data-centric Programming for Distributed Systems”. PhD thesis. University of California, Santa Cruz, USA, 2015. URL: <http://www.escholarship.org/uc/item/2296w4q3> (cited on page 8).
- [6] Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*. Volume 30. AFIPS Conference Proceedings. ACM Press, 1967, pages 483–485. DOI: 10.1145/1465482.1465560 (cited on page 44).
- [7] Joe Armstrong, Robert Viriding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993. ISBN: 978-0-13-285792-5 (cited on page 128).
- [8] Peter Bailis. “Coordination Avoidance in Distributed Databases”. PhD thesis. University of California, Berkeley, USA, 2015. URL: <http://www.escholarship.org/uc/item/8k8359g2> (cited on pages 9, 54, 55, 78).
- [9] 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), pages 181–192. ISSN: 2150-8097. DOI: 10.14778/2732232.2732237 (cited on pages 3, 6, 26, 84, 98).

## Bibliography

- [10] 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), pages 185–196. ISSN: 2150-8097. DOI: 10.14778/2735508.2735509 (cited on pages 9, 24, 54, 60, 122).
- [11] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Pregoça, Mahsa Najafzadeh, and Marc Shapiro. “Putting consistency back into eventual consistency”. In: *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*. ACM Press, 2015, 6:1–6:16. ISBN: 9781450332385. DOI: 10.1145/2741948.2741972 (cited on pages 51, 54, 60, 78, 122).
- [12] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. “UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems”. In: *Hybrid Systems*. Volume 1066. Lecture Notes in Computer Science. Springer, 1995, pages 232–243. DOI: 10.1007/BFb0020949 (cited on page 98).
- [13] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 0-201-10715-5. URL: <http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx> (cited on page 27).
- [14] Ted J. Biggerstaff. *A Perspective of Generative Reuse*. Technical report MSR-TR-97-26. Dec. 2001, page 43. URL: <https://www.microsoft.com/en-us/research/publication/a-perspective-of-generative-reuse/> (cited on page 5).
- [15] Stefan Blom, Jaco van de Pol, and Michael Weber. “LTSmin: Distributed and Symbolic Reachability”. In: *Computer Aided Verification*. Edited by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 354–359. ISBN: 978-3-642-14295-6 (cited on page 98).
- [16] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. *Reactive Manifesto*. 2021. URL: <https://www.reactivemanifesto.org/> (visited on 2021-10-29) (cited on pages 3, 128).
- [17] Susanne Braun, Annette Bieniusa, and Frank Elberzhager. “Advanced Domain-Driven Design for Consistency in Distributed Data-Intensive Systems”. In: *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. EuroSys '21: Sixteenth Euro-

- pean Conference on Computer Systems. Online United Kingdom: ACM, Apr. 26, 2021, pages 1–12. ISBN: 978-1-4503-8338-7. DOI: 10/gjs3st (cited on pages 4, 122, 129).
- [18] Marc Brooker, Tao Chen, and Fan Ping. “Millions of Tiny Databases”. In: *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. Edited by Ranjita Bhagwan and George Porter. USENIX Association, 2020, pages 463–478 (cited on pages 87, 113).
- [19] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. “Serializable isolation for snapshot databases”. In: *ACM Trans. Database Syst.* 34.4 (Dec. 2009), pages 1–42. ISSN: 0362-5915, 1557-4644. DOI: 10.1145/1620585.1620587 (cited on page 51).
- [20] Cassandra. 2019. URL: <https://cassandra.apache.org/> (visited on 2019-07-31) (cited on pages 38, 78).
- [21] Steve Cook, Conrad Bock, Pete Rivett, Tom Rutt, Ed Seidewitz, Bran Selic, and Doug Tolbert. *Unified Modeling Language (UML) Version 2.5.1*. Standard. Object Management Group (OMG), Dec. 2017. URL: <https://www.omg.org/spec/UML/2.5.1> (cited on page 6).
- [22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*. Edited by Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum. ACM Press, 2010, pages 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152 (cited on page 51).
- [23] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. “Seeing is Believing. A Client-Centric Specification of Database Isolation”. In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. Edited by Elad Michael Schiller and Alexander A. Schwarzmann. ACM, July 2017, pages 73–82. ISBN: 9781450349925. DOI: 10.1145/3087801.3087802 (cited on pages 6, 7, 55, 84, 85, 88, 91, 99, 103, 105, 111, 115, 121, 141).
- [24] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. ISBN: 978-0-201-30977-5. URL: <http://www.addison-wesley.de/main/main.asp?page=englisch/bookdetails%5C&productid=99258> (cited on page 4).

## Bibliography

- [25] Jeffrey Dean and Luiz André Barroso. “The tail at scale”. In: *Commun. ACM* 56.2 (Feb. 2013), page 74. ISSN: 0001-0782. DOI: 10.1145/2408776.2408794 (cited on page 56).
- [26] Andrzej Debski, Bartłomiej Szczepanik, Maciej Malawski, Stefan Spahr, and Dirk Muthig. “A Scalable, Reactive Architecture for Cloud Applications”. In: *IEEE Softw.* 35.2 (Mar. 2018), pages 62–71. ISSN: 0740-7459. DOI: 10.1109/ms.2017.265095722 (cited on pages 4, 38, 128, 129).
- [27] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. “OLTP-Bench. an extensible testbed for benchmarking relational databases”. In: *Proc. VLDB Endow.* 7.4 (Dec. 2013), pages 277–288. ISSN: 2150-8097. DOI: 10.14778/2732240.2732246 (cited on page 51).
- [28] Renate R. Eilers. “Fine-Grained Model Slicing for Faster Verification”. Master’s thesis. Utrecht University, 2018 (cited on page 14).
- [29] Tamer Eldeeb and Phil Bernstein. *Transactions for Distributed Actors in the Cloud*. Technical report. Oct. 2016. URL: <https://www.microsoft.com/en-us/research/publication/transactions-distributed-actors-cloud-2/> (cited on page 53).
- [30] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. “The Notions of Consistency and Predicate Locks in a Database System”. In: *Commun. ACM* 19.11 (1976), pages 624–633. DOI: 10.1145/360363.360369. URL: <https://doi.org/10.1145/360363.360369> (cited on page 55).
- [31] Eric Evans and Eric J Evans. *Domain-driven design - tackling complexity in the heart of software*. Addison-Wesley, 2004. ISBN: 978-0-321-12521-7 (cited on pages 4, 122).
- [32] Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O’Neil, Patrick E. O’Neil, and Dennis E. Shasha. “Making Snapshot Isolation Serializable”. In: *ACM Transactions on Database Systems* 30.2 (2005), pages 492–528. DOI: 10.1145/1071610.1071615 (cited on page 84).
- [33] Wan Fokkink. *Distributed Algorithms, second edition: An Intuitive Approach*. The MIT Press. MIT Press, 2018. ISBN: 9780262345521. URL: <https://mitpress.mit.edu/books/distributed-algorithms-second-edition> (cited on page 3).
- [34] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2012. ISBN: 978-0-321-12742-6 (cited on page 9).

- [35] Hector Garcia-Molina and Kenneth Salem. “Sagas”. In: *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*. Edited by Umeshwar Dayal and Irving L. Traiger. ACM Press, 1987, pages 249–259. DOI: 10.1145/38713.38742. URL: <https://doi.org/10.1145/38713.38742> (cited on page 143).
- [36] Gatling. 2018. URL: <https://gatling.io/> (visited on 2018-07-23) (cited on page 41).
- [37] Seth Gilbert and Nancy A. Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”. In: *SIGACT News* 33.2 (2002), pages 51–59. DOI: 10.1145/564585.564601. URL: <https://doi.org/10.1145/564585.564601> (cited on page 7).
- [38] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Volume 1032. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996. ISBN: 9783540607618. DOI: 10.1007/3-540-60761-7 (cited on page 54).
- [39] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. “Verifying strong eventual consistency in distributed systems”. In: *PACMPL* 1.OOPSLA (2017), 109:1–109:28 (cited on pages 78, 98).
- [40] Jim Gray. “Notes on Data Base Operating Systems”. In: *Operating Systems, An Advanced Course*. Edited by Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle. Volume 60. Lecture Notes in Computer Science. Springer, 1978, pages 393–481. ISBN: 3-540-08755-9. DOI: 10.1007/3-540-08755-9\_9 (cited on pages 24, 27).
- [41] Jim Gray and Leslie Lamport. “Consensus on Transaction Commit”. In: *ACM Transactions on Database Systems* 31.1 (2006), pages 133–160. DOI: 10.1145/1132863.1132867 (cited on pages 10, 87, 92, 103, 113).
- [42] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993. ISBN: 1-55860-190-2 (cited on pages 3, 6, 55, 60).
- [43] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Pearson Education, 2014 (cited on page 11).

## Bibliography

- [44] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014. ISBN: 978-0-262-02771-7 (cited on page 98).
- [45] Jason Gustafson and Guozhang Wang. *Hardening Kafka Replication*. <https://github.com/hachikuji/kafka-specification>. 2020 (cited on pages 87, 98, 113).
- [46] Theo Haerder and Andreas Reuter. “Principles of transaction-oriented database recovery”. In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pages 287–317. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/289.291 (cited on pages 7, 25, 26, 130).
- [47] David Harel. “Statecharts: A Visual Formalism for Complex Systems”. In: *Sci. Comput. Program.* 8.3 (1987), pages 231–274. DOI: 10.1016/0167-6423(87)90035-9. URL: [https://doi.org/10.1016/0167-6423\(87\)90035-9](https://doi.org/10.1016/0167-6423(87)90035-9) (cited on page 6).
- [48] Pat Helland. “Opinion”. In: *2007 First ACM/IEEE International Conference on Distributed Smart Cameras*. IEEE, 2007, pages 132–141. ISBN: 9781424413546. DOI: 10.1109/icdsc.2007.4357494 (cited on page 24).
- [49] Pat Helland. “Don’t Get Stuck in the “Con” Game: Consistency, convergence, and confluence are not the same! Eventual consistency and eventual convergence aren’t the same as confluence, either”. In: *ACM Queue* 19.3 (2021), pages 16–35. DOI: 10.1145/3475965.3480470. URL: <https://doi.org/10.1145/3475965.3480470> (cited on pages 6, 7).
- [50] Joseph M. Hellerstein and Peter Alvaro. “Keeping CALM: When Distributed Consistency is Easy”. In: *CoRR abs/1901.01930* (2019). arXiv: 1901.01930. URL: <http://arxiv.org/abs/1901.01930> (cited on pages 8, 54, 78, 122, 142).
- [51] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. “Flat combining and the synchronization-parallelism tradeoff”. In: *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures - SPAA ’10*. ACM Press, 2010, pages 355–364. ISBN: 9781450300797. DOI: 10.1145/1810479.1810540 (cited on pages 55, 79).
- [52] Ludovic Henrio and Justine Rochas. “Multiactive objects and their applications”. In: *Logical Methods in Computer Science* 13.4 (2017) (cited on page 80).

- [53] Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Matson, and Richard Steiger. “Actor induction and meta-evaluation”. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73*. ACM Press, 1973, pages 235–245. DOI: 10.1145/512927.512942 (cited on pages 4, 37, 128).
- [54] Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. “Flat combining synchronized global data structures”. In: *7th International Conference on PGAS Programming Models*. 2013, page 76 (cited on page 79).
- [55] Gerard J. Holzmann. *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley, 2004. ISBN: 978-0-321-22862-8 (cited on page 98).
- [56] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. ISBN: 978-0-262-10114-1 (cited on pages 98, 121).
- [57] JMH. *OpenJDK: Java Microbenchmark Harness*. 2019. URL: <https://openjdk.java.net/projects/code-tools/jmh> (visited on 2019-07-30) (cited on page 73).
- [58] J.R. Jordan, J. Banerjee, and R.B. Batman. “Precision locks”. In: *Proceedings of the 1981 ACM SIGMOD international conference on Management of data - SIGMOD '81*. Edited by Y. Edmund Lien. ACM Press, 1981, pages 143–147. ISBN: 0897910400. DOI: 10.1145/582318.582340 (cited on page 55).
- [59] Kyle Kingsbury and Peter Alvaro. “Elle: Inferring Isolation Anomalies from Experimental Observations”. In: *CoRR abs/2003.10554* (2020). arXiv: 2003.10554 (cited on pages 84, 98).
- [60] Kyle Kinsbury. *Jepsen: Distributed Systems Safety Research*. <http://jepsen.io/>. 2020 (cited on page 98).
- [61] Kyle Kinsbury. *Knossos*. <https://github.com/jepsen-io/knossos>. 2020 (cited on page 84).
- [62] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O’Reilly, 2016. ISBN: 978-1-4493-7332-0. URL: <http://shop.oreilly.com/product/0636920032175.do> (cited on pages 9, 24, 26, 98).
- [63] Martin Kleppmann. *Hermitage: Testing Transaction Isolation Levels*. <https://github.com/ept/hermitage>. 2020 (cited on pages 84, 98).

## Bibliography

- [64] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. “Local-first software: you own your data, in spite of the cloud”. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*. Edited by Hidehiko Masuhara and Tomas Petricek. ACM, 2019, pages 154–178. DOI: 10.1145/3359591.3359737. URL: <https://doi.org/10.1145/3359591.3359737> (cited on page 4).
- [65] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. “RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation”. In: *SCAM*. IEEE Computer Society, 2009, pages 168–177 (cited on pages 72, 126).
- [66] Alex Kok. “Property-based testing Rebel semantics in the generated code”. Master’s thesis. University of Amsterdam, 2017 (cited on page 14).
- [67] Sebastiaan la Fleur. “Static Analysis of Symbolic Transition Systems with Goose”. Master’s thesis. University of Twente, Mar. 2018 (cited on page 14).
- [68] Leslie Lamport. “The Part-Time Parliament”. In: *ACM Trans. Comput. Syst.* 16.2 (1998), pages 133–169. DOI: 10.1145/279227.279229. URL: <https://doi.org/10.1145/279227.279229> (cited on pages 113, 143).
- [69] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN: 0-3211-4306-X (cited on pages 55, 87, 98, 113).
- [70] Leslie Lamport. *The PlusCal Algorithm Language - Microsoft Research*. <https://www.microsoft.com/en-us/research/publication/pluscal-algorithm-language/> (cited on page 87).
- [71] Costin Leau. *Spring data redis-retwis-j*. 2013. URL: <https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/> (cited on page 51).
- [72] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. “Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary”. In: *OSDI*. USENIX Association, 2012, pages 265–278 (cited on pages 78, 122, 145).

- [73] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. “Don’t Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems”. In: *OSDI*. USENIX Association, 2016, pages 383–400 (cited on page 50).
- [74] Marjan Mernik, Jan Heering, and Anthony M. Sloane. “When and how to develop domain-specific languages”. In: *ACM Comput. Surv.* 37.4 (2005), pages 316–344. DOI: 10.1145/1118890.1118892. URL: <https://doi.org/10.1145/1118890.1118892> (cited on pages 4, 6).
- [75] Microsoft. *High-Level TLA+ Specifications for the Five Consistency Levels Offered by Azure Cosmos DB*. <https://github.com/Azure/azure-cosmos-tla>. 2020 (cited on pages 87, 98, 113).
- [76] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *TACAS*. Volume 4963. Lecture Notes in Computer Science. Springer, 2008, pages 337–340 (cited on pages 60, 64, 72, 114).
- [77] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. “Extracting More Concurrency from Distributed Transactions”. In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014*. Edited by Jason Flinn and Hank Levy. ACM Press, 2014, pages 479–494. ISBN: 1880446820. DOI: 10.1145/238721. URL: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/mu> (cited on page 54).
- [78] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Tappan Morris. “Phase Reconciliation for Contended In-Memory Transactions”. In: *OSDI*. USENIX Association, 2014, pages 511–524 (cited on pages 54, 55, 79).
- [79] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardouff. “How Amazon web services uses formal methods”. In: *Commun. ACM* 58.4 (Mar. 2015), pages 66–73. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2699417 (cited on pages 87, 98, 113).
- [80] Patrick E. O’Neil. “The Escrow transactional method”. In: *ACM Trans. Database Syst.* 11.4 (Dec. 1986), pages 405–430. ISSN: 0362-5915, 1557-4644. DOI: 10.1145/7239.7265 (cited on page 54).

## Bibliography

- [81] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008. ISBN: 9780981531601. URL: <https://books.google.nl/books?id=MFjNhTjeQKkC> (cited on page 126).
- [82] Diego Ongaro and John K. Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Edited by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pages 305–319. URL: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro> (cited on page 113).
- [83] Orleans. 2018. URL: <https://dotnet.github.io/orleans/> (visited on 2018-12-21) (cited on pages 53, 129).
- [84] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer New York, 2011. ISBN: 9781441988348. DOI: 10.1007/978-1-4419-8834-8 (cited on pages 24, 27).
- [85] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. “Conflict-free Replicated Data Types (CRDTs)”. In: *CoRR abs/1805.06358* (2018). arXiv: 1805.06358. URL: <http://arxiv.org/abs/1805.06358> (cited on pages 3, 54, 120, 122).
- [86] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. “Conflict-Free Replicated Data Types CRDTs”. In: *Encyclopedia of Big Data Technologies*. Springer, 2019 (cited on pages 10, 98).
- [87] Francois Raab, Walt Kohler, and Amitabh Shah. “Overview of the TPC benchmark C: The order-entry benchmark”. In: *Transaction Processing Performance Council, Tech. Rep* (2013) (cited on pages 46, 51, 60, 69).
- [88] Reactors. 2018. URL: <http://reactors.io/> (visited on 2018-12-21) (cited on page 53).
- [89] Kun Ren, Alexander Thomson, and Daniel J. Abadi. “An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems”. In: *PVLDB 7.10* (2014), pages 821–832 (cited on page 79).
- [90] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. “Supporting the understanding and comparison of low-code development platforms”. In: *46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, Portoroz, Slovenia, August 26-28, 2020*. IEEE, 2020, pages 171–178. DOI: 10.1109/

- SEAA51224.2020.00036. URL: <https://doi.org/10.1109/SEAA51224.2020.00036> (cited on page 5).
- [91] Douglas C. Schmidt. “Guest Editor’s Introduction: Model-Driven Engineering”. In: *Computer* 39.2 (2006), pages 25–31. DOI: 10.1109/MC.2006.58. URL: <https://doi.org/10.1109/MC.2006.58> (cited on page 4).
- [92] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. “Open Versus Closed: A Cautionary Tale”. In: *3rd Symposium on Networked Systems Design and Implementation (NSDI 2006), May 8-10, 2007, San Jose, California, USA, Proceedings*. Edited by Larry L. Peterson and Timothy Roscoe. USENIX, 2006. URL: <http://www.usenix.org/events/nsdi06/tech/schroeder.html> (cited on pages 42, 45).
- [93] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. “Conflict-Free Replicated Data Types”. In: *SSS*. Volume 6976. Lecture Notes in Computer Science. Springer, 2011, pages 386–400 (cited on pages 8, 78, 120, 122).
- [94] Tim Soethout. “Exploiting models for scalable and high throughput distributed software”. In: *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2019, Athens, Greece, October 20-25, 2019*. Edited by Yannis Smaragdakis. ACM, 2019, pages 35–37. DOI: 10.1145/3359061.3361073 (cited on page 20).
- [95] Tim Soethout. *Path-Sensitive Atomic Commit: Local Coordination Avoidance for Distributed Transactions Evaluation Data*. Oct. 2019. DOI: 10.5281/zenodo.3405371 (cited on pages 19, 26, 42, 142).
- [96] Tim Soethout. *Static Local Coordination Avoidance for Distributed Objects Artifacts*. Sept. 2019. DOI: 10.5281/zenodo.3405232 (cited on pages 20, 61, 72, 75, 129, 134, 142).
- [97] Tim Soethout. *TimSoethout/TLA-CI: TLA<sup>+</sup> Specifications Used in “Automated Validation of State-Based Client-Centric Isolation with TLA<sup>+</sup>”*. Zenodo. July 2020. DOI: 10.5281/zenodo.3961617. URL: <https://github.com/TimSoethout/tla-ci> (cited on pages 20, 85, 89, 129, 134, 142).
- [98] Tim Soethout. *TimSoethout/cbc-artifacts: Artifacts for AGERE’21 paper “Contract-Based Return-Value Commutativity: Safely exploiting contract-based commutativity for faster serializable transactions”*. Zenodo. Sept. 2021. DOI: 10.5281/zenodo.5497756. URL: <https://github.com/cwswat/cbc-artifacts> (cited on pages 20, 104, 113, 115, 129, 134, 142).

## Bibliography

- [99] Tim Soethout. *cwi-swat/rebel-runtime-lib: Rebel runtime based on LoCA. Version zenodo*. Mar. 2022. DOI: 10.5281/zenodo.6381708. URL: <https://github.com/cwi-swat/rebel-runtime-lib> (cited on page 129).
- [100] Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Static local coordination avoidance for distributed objects”. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2019*. ACM Press, 2019, pages 21–30. ISBN: 9781450369824. DOI: 10.1145/3358499.3361222 (cited on pages 19, 28, 50, 54, 59, 99, 103, 104, 113–115, 120, 121).
- [101] Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Automated Validation of State-Based Client-Centric Isolation with TLA<sup>+</sup>”. In: *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops - ASYDE, CIFMA, and CoSim-CPS, Amsterdam, The Netherlands, September 14-15, 2020, Revised Selected Papers*. Edited by Loek Cleophas and Mieke Massink. Volume 12524. Lecture Notes in Computer Science. Springer, 2020, pages 43–57. DOI: 10.1007/978-3-030-67220-1\_4 (cited on pages 20, 83, 103, 111, 115, 118).
- [102] Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Contract-Based Return-Value Commutativity: Safely exploiting contract-based commutativity for faster serializable transactions”. In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2021*. ACM Press, 2021. DOI: 10.1145/3486601.3486707 (cited on pages 20, 101).
- [103] Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. “Path-Sensitive Atomic Commit - Local Coordination Avoidance for Distributed Transactions”. In: *The Art, Science, and Engineering of Programming 5.1* (2021), page 3. DOI: 10.22152/programming-journal.org/2021/5/3 (cited on pages 19, 23, 66, 97, 99, 103, 104, 113, 114, 120).
- [104] Open API Specification. 2018. URL: <https://github.com/OAI/OpenAPI-Specification> (visited on 2018-09-25) (cited on page 166).
- [105] Jouke Stoel. *Rebel*. 2020. URL: <https://github.com/cwi-swat/rebel> (visited on 2020-01-10) (cited on pages 14, 25, 35, 125, 134).
- [106] Jouke Stoel. *Rebel*. 2020. URL: <https://github.com/cwi-swat/rebel2> (visited on 2020-01-10) (cited on pages 14, 25, 35).

- [107] Jouke Stoel, Tijs van der Storm, and Jurgen Vinju. “Modeling with Mocking”. In: *2021 IEEE 14th International Conference on Software Testing, Validation and Verification (ICST)*. 2021, pages 59–70. DOI: 10.1109/ICST49551.2021.00018 (cited on pages 14, 121).
- [108] Jouke Stoel, Tijs van der Storm, Jurgen Vinju, and Joost Bosman. “Solving the bank with Rebel: On the design of the Rebel specification language and its application inside a bank”. In: *Proceedings of the 1st Industry Track on Software Language Engineering - ITSLE 2016*. ACM Press, 2016, pages 13–20. ISBN: 9781450346467. DOI: 10.1145/2998407.2998413 (cited on pages 2, 14, 25, 35, 37, 60, 72, 99, 121, 134).
- [109] Michael Stonebraker and Ariel Weisberg. “The VoltDB Main Memory DBMS”. In: *IEEE Data Eng. Bull.* 36.2 (2013), pages 21–27 (cited on page 78).
- [110] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007. ISBN: 978-0-13-239227-3 (cited on pages 4, 39, 72, 98).
- [111] Thanusijan Tharumarajah. “Runtime testing generated systems from Rebel specifications”. Master’s thesis. University of Amsterdam, 2017 (cited on page 14).
- [112] Rene van Gasteren. “Natural Language Generation from Rebel Specifications”. Master’s thesis. University of Amsterdam, 2016 (cited on page 14).
- [113] Wiebe van Geest. “Dependent Types for Invariants in Session Types”. Master’s thesis. TU Delft, 2018 (cited on page 14).
- [114] Gerhard Weikum. “Principles and Realization Strategies of Multilevel Transaction Management”. In: *ACM Transactions on Database Systems* 16.1 (Mar. 1991), pages 132–180. ISSN: 0362-5915. DOI: 10.1145/103140.103145 (cited on pages 99, 141).
- [115] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems*. Elsevier, 2002. ISBN: 9781558605084. DOI: 10.1016/c2009-0-27891-3 (cited on pages 6, 7, 39, 54, 102, 103, 105, 112, 130).
- [116] Peter D. Wessels. “Leveraging behavioural domain models in Model-Driven User Interface Development with GLUI”. Master’s thesis. University of Twente, June 2018 (cited on page 14).

## Bibliography

- [117] Jordi W. M. Wippert. “Change Impact Analysis for Rebel Specifications”. Master’s thesis. Utrecht University, 2020 (cited on page 14).
- [118] 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 - AGERE 2018*. ACM Press, 2018, pages 23–32. ISBN: 9781450360661. DOI: 10.1145/3281366.3281372 (cited on pages 54, 78, 122, 145).

# A

## Path-Sensitive Atomic Commit

### A.1 Example 2PL/2PC and PSAC diagrams with ABORT

Figures A.1 and A.2 show the same example situation as used in section 2.3.1, but in this case the first action 2PC transaction aborts. We see at figure A.1.③ and figure A.2.④ that action  $-\text{€}30$  is aborted by the 2PC coordinator. With PSAC  $-\text{€}50$  still starts, since both outcomes (commit and abort) where taken into account.

### A.2 Actor class definition

The library using Akka expects the Rebel specifications to implement a Scala trait `RebelSpec`. A simplified version of the Scala code generated from the Account example of listing 2.2 is shown in listing A.1.

The algebraic data types `AccountState` and `AccountCommand` model respectively the Rebel state machine states and actions. The methods `initialState`, `allStates`, and `finalStates` encode metadata of the life cycle of an entity. The method `nextState` encodes how transitions are performed and via which events. The preconditions and actions' effects required for PSAC are known from the Rebel specification and generated into the actor code. `checkPre` checks the preconditions for each incoming action. As a result, the method `apply` calculates the new state of the account given the current state and action.

Finally, the `syncOps` method returns a set of operations between entities that must be synchronized as per the `sync` construct. Since the `Account` class requires no synchronization it returns the empty set.

The Rebel library contains a restful HTTP API which derives endpoints for all the specifications and actions. These are used to trigger actions on the actors.

## A Path-Sensitive Atomic Commit

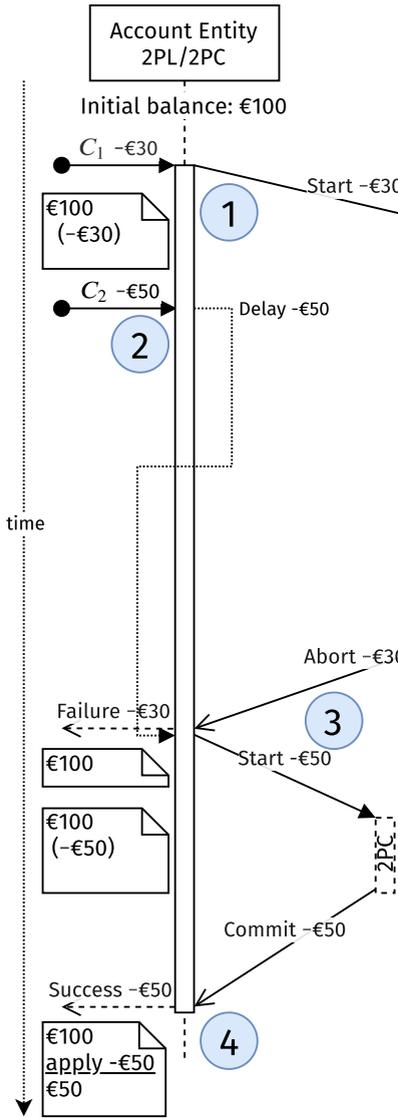


Figure A.1 Vanilla Two-Phase Commit

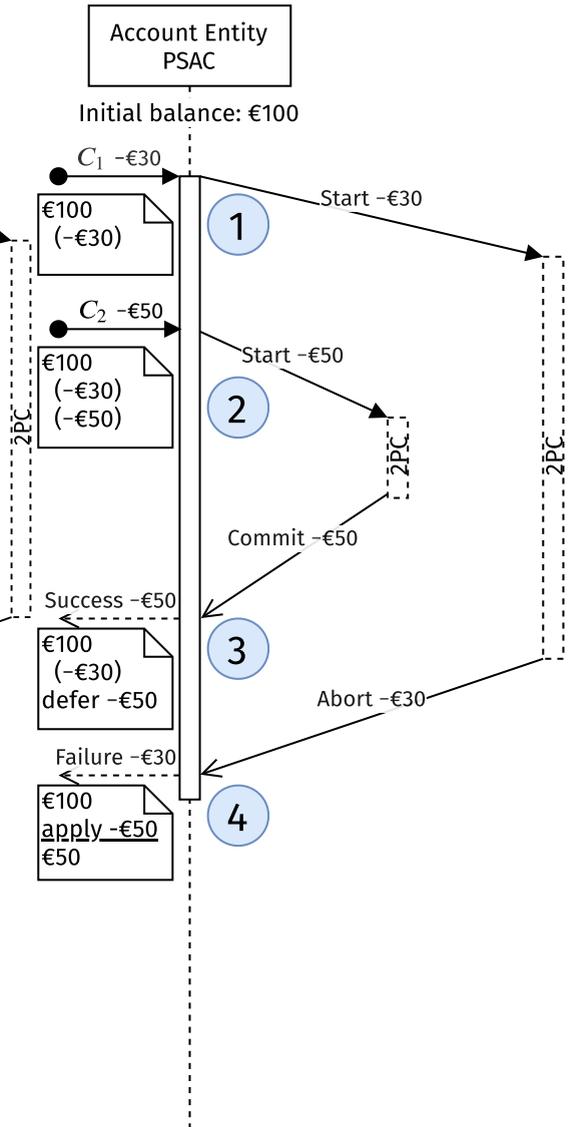


Figure A.2 Path-Sensitive Atomic Commit

**Listing A.1** Generated Scala code for the Rebel Account entity (slightly simplified)

```

1 class AccountActor extends RebelFSMACTOR
2   with RebelSpec[AccountState, AccountData, AccountCommand]{
3   val initialState: AccountState = Init
4   val allStates : Set[AccountState] = Set(Blocked, Closed, Init, Opened)
5   val finalStates : Set[AccountState] = Set(Closed)
6
7   def nextState: PartialFunction[(AccountState, AccountCommand), AccountState] = {
8     case (Uninit, _: Open) => Opened
9     case (Opened, _: Withdraw) => Opened
10    case (Opened, _: Deposit) => Opened
11    case (Opened, _: Close) => Closed
12  }
13
14  def checkPre(data: AccountData, now: DateTime)
15    : PartialFunction[AccountCommand, CheckResult] = {
16    case OpenAccount(accountNumber, initialDeposit) =>
17      checkPreCondition(initialDeposit ≥ EUR(50.00))
18    case Close() =>
19      checkPreCondition(data.get.balance.get == EUR(0.00))
20    case Withdraw(amount) =>
21      checkPre(amount > EUR(0.00)) combine
22      checkPre(data.get.balance.get - amount ≥ EUR(0.00))
23    case Deposit(amount) =>
24      checkPreCondition((amount > EUR(0.00)))
25  }
26
27  def apply(data: AccountData): PartialFunction[AccountCommand, AccountData] = {
28    case OpenAccount(accountNumber, initialDeposit) =>
29      Initialized(AccountData(accountNumber = Some(accountNumber),
30        balance = Some(initialDeposit)))
31    case Withdraw(amount) =>
32      data.map(r => r.copy(balance = r.balance.map(_ - amount)))
33    case Deposit(amount) =>
34      data.map(r => r.copy(balance = r.balance.map(_ + amount)))
35  }
36
37  def syncOps(data: RData): PartialFunction[AccountCommand, Set[SyncOp]] = Set.empty
38 }

```

## A Path-Sensitive Atomic Commit

**Listing A.2** Generated code corresponding to the `sync` action in the **MoneyTransfer**

```
1 def syncOps(data: MoneyTransferData)
2   : PartialFunction[MoneyTransferCommand, Set[SyncOp]] = {
3   case Book(amount, from, to) => Set(
4     SyncAction(ContactPoint(Account, from), Withdraw(amount)),
5     SyncAction(ContactPoint(Account, to), Deposit(amount))
6   )
7 }
```

The translation of the **MoneyTransfer** class follows the same pattern. However, in this case the method `syncOps` does not return the empty set. It is shown in listing A.2. Each `sync` action is translated to a `SyncAction` with a `ContactPoint`, which enables sending messages to the `sync` participant living somewhere in the cluster, and the action on the `sync` participant itself.

### A.3 Detailed Rebel implementation using Akka

**FSM** Each Rebel specification describes a single financial product. Each of these products can have multiple instances, which we call entities, that can be identified by their unique Rebel `@key`. This nicely maps to an actor definition per specification, where each running instance of this actor is an entity. Since a specification describes a state machine we piggyback on the Akka Domain Specific Language (DSL) for Finite State Machines (FSM). Akka FSM provides constructs for States, Data and Transitions. Notable features are Timeouts when no commands are received and batching of events to the persistence layer for improved performance.

**Cluster** The Akka cluster feature allows us to run our application on multiple servers, by supplying a mechanism to add extra nodes to the Akka cluster and location-transparently send messages to actors on remote cluster nodes. Akka takes care of the setting up of the cluster and the joining and leaving of nodes. This is what allows our application to scale horizontally in the number of nodes and therefore total the number of running actors and the amount system resources.

**Sharding** Rebel specifications allow interaction with other specification in pre-, postconditions and synchronised actions. Other entities can be accessed by using the specification name and identity (@key), e.g. **Account[this.from]**. The identity referenced, can be from a specification field or event field.

Akka Sharding allows us to distribute the running actors over the cluster nodes. Each cluster node can start a shard region, which is used to send messages to a certain type of actor somewhere in the cluster. An actor can be reached by a unique logical identifier. The sharding feature makes sure that for each unique identifier only one actor is active in the whole cluster. If a message is send to an identity that is not yet running, the corresponding actor will be started somewhere in the sharding cluster.

Rebel identity nicely maps to the logical identity of Akka Sharding. In the target application we use a sharding region per specification. This allows us to send messages to each individual actor, without knowing or caring on which node it is running in the actual Actor Cluster. If the entity is not yet created, Sharding will make sure it is started and usable. This is called *location transparency*.

**Persistence** Sharding allows us to distribute the actors over the system and makes sure only once actor is running per entity. In order to be able to durably store the data and state of an entity we use Akka Persistence.

Akka Persistence is based on Event Sourcing (ES) and Command Query Responsibility Segregation (CQRS). This means that for each event that a Rebel entity can process, a Command and an Event is defined. Respectively denoting the intention to let the event happen and the immutable proof that the event occurred.

Event sourcing means that we create an immutable log of all sequential immutable events that happened.

In our application this means that for each actor corresponding with a Rebel entity we store the incoming command after a precondition check in our persistency layer<sup>1</sup> as an intention to execute this command. After successfully persisted, the command is executed. If valid an event of the transition will be committed to the persistence layer and side effects to the internal state will

---

<sup>1</sup>We use Cassandra, but many more journaling plugins are available. The queries are configured to write and read with Quorum, so we know that each command is persisted safely before it is handled.

## A Path-Sensitive Atomic Commit

be executed. Akka Persistence makes sure other incoming commands are delayed until the in-progress command is handled completely.

The result of recording all the events in a persisted log is that we can restart an actor and replay all the events that happened and get it back into the last committed state. Because Sharding makes sure there is only one actor with the same logical identity running at a single moment, we can be sure that only a single persistent actor is writing to the log and know that its internal entity level state is consistent.<sup>2</sup>

In the event of an actor or cluster node crashes, the entity can be restarted on another node without loss of data. This also means that the persistency guarantees are heavily dependent on the guarantees of the underlying persistence journal implementation.

As persistence backend we use Cassandra 3. This is a production-ready backend for Akka Persistence and also the mostly used.

**HTTP** The Rebel events are exposed as REST endpoint for each logical identifier on which commands can be triggered to the corresponding actors.<sup>3</sup> This uses Akka HTTP for non-blocking IO and can be automatically derived from the available generated specification and event implementations. We use the *circe* JSON library to automatically derive JSON encoders and decoders for each of the events, based on the generated case classes. This means that for the entire REST interface almost no additional code has to be generated, next to the field and event definitions. A Scala worksheet file is available to manually generate example JSON-documents that the system accepts.

An Open API specification[104] definition is generated which corresponds to the generated REST interface. This allows for easy consumption of the interface.

These endpoints are used for the experiment by the load generator.

---

<sup>2</sup> Also because the actor only handles a single message and therefore a single command at the same time

<sup>3</sup> url template: `POST /SPECIFICATION-NAME/:ID/EVENT-NAME`

# B

## Posters

This appendix contains all the posters created and presented during the research for this dissertation.



# Scaling the Bank

Tim Soethout ([tim.soethout@ing.com](mailto:tim.soethout@ing.com))



## Goal: Scaling Backend for DSL

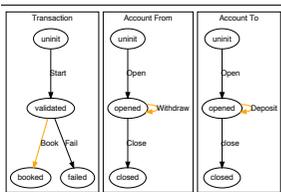
Backend for **rebel** implementing (non-)functional requirements

```

event Open!(initialDeposit: Money) {
  precondition {
    initialDeposit >= 0
  }
  postcondition {
    new this.balance == initialDeposit
  }
}

event Withdraw(amount: Money) {
  precondition {
    amount >= 0
    this.balance - amount >= 0
  }
  postcondition {
    new this.balance == this.balance - amount
  }
}

event Deposit(amount: Money) {
  precondition {
    amount >= 0
  }
  postcondition {
    new this.balance == this.balance + amount
  }
}
    
```



Classifier as StateMachine  
 event Book(amount: Money, to: Stan, from: Stan) {  
 type {  
 AccountFrom & withdraw(amount);  
 AccountTo & deposit(amount);  
 }  
 }  
 Example Rebel specifications for Account and Transaction (left) Pre- & post-conditions & sync (right) state machine of specification

## Context: Reactive Architecture

- Reactive actor-based Architecture:
- Each entity is an actor — natural mapping from/to specification
  - Actors are consistency boundaries
  - Event sourcing for persistence
  - Sharding and Location-transparency for scaling and resiliency

## Challenge: Synchronisation is hard

- Performant Synchronisation
- Atom transaction — All involved entities should step, or none.
  - Two-Phase Commit is safe, well-known implementation
    - Locking, so not strict highly-available
    - Scales reasonably for evenly distributed load over entities (embarrassingly parallel).
    - Busy entities with high contention can become bottleneck.

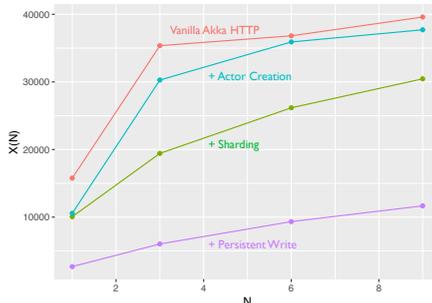
## Approach: Baseline + Improvement

- Make sure baseline infrastructure (Akka) performs as expected.
- Implement **sync** in known correct way: Two-Phase Commit (2PC)
- Improve synchronisation performance by leveraging **domain knowledge**:

Control Dependent Atomic Commit

## Preliminary Results

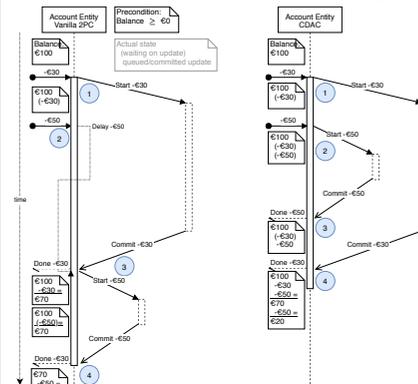
Baseline Akka is horizontally scalable?



Throughput (X(N)) vs number of nodes (N) of increasingly complex baseline Akka functionality

- Two-Phase Commit gives decent performance for many use cases:
  - Low volume or
  - Low contention

## Leveraging Domain Knowledge



### Vanilla 2PC:

- Withdraw €30 action arrives; triggers 2PC
- Withdraw €50 arrives, delayed because entity is locked
- €30 commits, effect applied; -€50 2PC starts
- €50 commits, effect applied

### CDAC:

- Withdraw €30 action arrives; triggers 2PC
- Meanwhile Withdraw €50 arrives; triggers 2PC; allowed, because not dependent on any in progress transactions' outcome
- €50 commits first, effect is queued
- €30 commits, both effect applied in order

Hypothesis: CDAC results in lower latency by dynamically detecting independent actions to run in parallel.

## Ongoing Work

- Evaluate performance improvement of CDAC
- Also on realistic ING use case
- Reordering variant of CDAC (static detection)

Figure B.1 Poster presented at the SATIS'18 summer school. CDAC is an earlier name for PSAC.



### Goal: Scaling Backend for DSL

Backend for **rebel** implementing (non-)functional requirements

```

    spec/Function Account(AccountNumber: Int, Money, Balance: Money)
    @interface DSL
    -- transfer
    transfer : Account <*> Account, Amount
    -- withdraw
    withdraw : Account, Amount
    -- close
    close : Account

    Fixed Closure
    --
    means {
    Account(AccountNumber: Money)
    preconditions: validBalance > 0, valid & amp;
    postconditions: postBalance = 0, valid & amp;
    }
    --
    means WithdrawAccount: Money
    preconditions: amount > 0 & amp; valid & amp;
    postconditions: amount > 0 & amp; preBalance - amount > 0, valid & amp;
    --
    means CloseAccount: Money
    preconditions: amount > 0 & amp; valid & amp;
    postconditions: postBalance = 0, amount > 0, valid & amp;
    }
    --
    means Book: Money
    preconditions: amount > 0 & amp; valid & amp;
    postconditions: postBalance = amount & amp;
    }
    }
    spec/Function Transaction
    @interface DSL
    -- booked
    booked : Book
    -- failed
    failed : Book

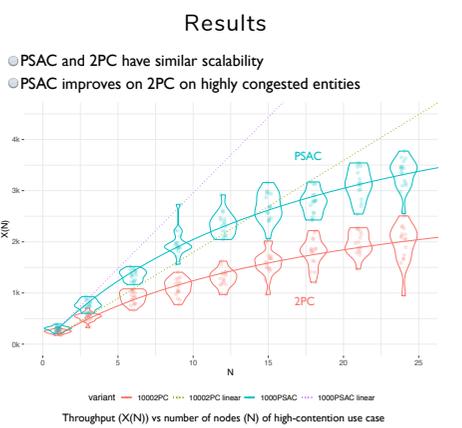
    Fixed Closure
    --
    means {
    Transaction(TransactionNumber: Int, Money, Status)
    preconditions: valid & amp;
    }
    }
    
```

Example Rebel specifications for Account and Transaction (left) Pre- & post-conditions & sync (right) state machine of specification

### Approach: Baseline + Improvement

1. Make sure baseline infrastructure (Akka) performs as expected.
2. Implement **sync** in known correct way:  
Two-Phase Commit (2PC)
3. Improve runtime synchronisation performance by leveraging **domain knowledge**:

## Path-Sensitive Atomic Commit



- Two-Phase Commit gives decent performance for many use cases:
  - Low volume or
  - Low contention

### Leveraging Domain Knowledge

**Vanilla 2PC:**

1. Withdraw €30 action arrives; triggers 2PC
2. Withdraw €50 arrives, delayed because entity is locked
3. -€30 commits, effect applied; -€50 2PC starts
4. -€50 commits, effect applied

**PSAC:**

1. Withdraw €30 action arrives; triggers 2PC
2. Meanwhile Withdraw €50 arrives; triggers 2PC; allowed, because not dependent on in any in progress transactions' outcome
3. -€50 commits first, effect is queued
4. -€30 commits, both effect applied in order

PSAC results in higher throughput by dynamically detecting independent actions to run in parallel.

- ### Future Work
- Static offline analysis of independent actions
  - Formal evaluation of PSAC

Figure B.2 Poster presented at the SEN symposium'19.

# Banking on Domain Knowledge for Faster Distributed Transactions

Tim Soethout

## Context: Domain models capture business logic

- Distributed applications are hard
- Easy to mix up implementation details and business logic
- DSLs can help: capture domain knowledge without implementation details

## Goal: From DSL to scalable application

**Source:** Synchronizing state machines domain models with guards and effects

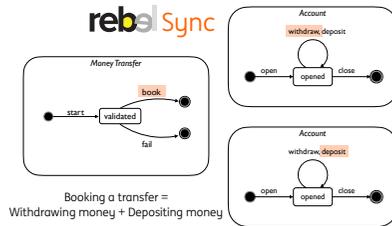


Figure 1. Synchronized operation of Money Transfer between two Bank Accounts

**Target:** Scalable and correct implementation

## Challenge: Inherent trade-off between isolation and performance

Consistency / Isolation  $\iff$  Scalability / Performance

- More coordination  $\Rightarrow$  Less room for performance
- Less coordination  $\Rightarrow$  More room for optimization

## Approach: Reduce coordination using Domain Knowledge

- Leverages semantic knowledge from domain models to safely increase parallelism
- Clear on consistency/isolation trade-off (WIP)

**Based on Independent Events [2]:**

An incoming operation is accepted or rejected, independent of an in-progress event's commit or abort

## How: Local Coordination Avoidance (LoCA)

- Safely increase parallel transactions on (distributed) resources
- Dynamic **LOCA<sup>S</sup>** [3]: run time discovery of independent events; e.g. multiple withdrawals are always safe in parallel when enough balance is available
- Static **LOCA<sup>S</sup>** [2]: offline discovery using SMT-solver, reducing run-time overhead; e.g. multiple deposits are always safe in parallel because it only adds balance

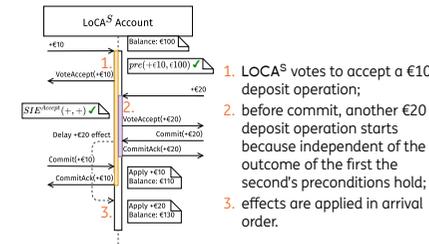


Figure 2. Parallel operations on LOCA<sup>S</sup>

Parallel operations when statically or dynamically determined they never invalidate local entity consistency

## Results

Increased throughput and latency in high-contention scenarios

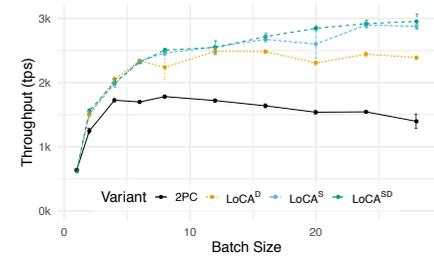


Figure 3. Scalability of LOCA variants in a high-contention scenario

## Current Work: Closing the loop

LOCA is not always serializable:

- A state-based client-centric database isolation model in TLA+ determines the isolation guarantees of algorithms [1]
- Semantic Isolation: adapt to fairly determine serializability for semantically higher-level operations

Full circle: No silver bullet, in the end we can only help business experts to make the trade-off by providing clear feedback

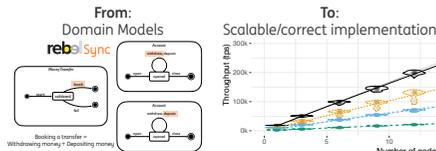
## References

- [1] Tim Soethout, Tjib van der Storm, and Jurgens J. Vrijs. Automated validation of state-based client-centric isolation with TLA+. In Software Engineering and Formal Methods: SEFM Collocated Workshop - AFDL, 2020, LNCS, doi:10.1007/978-3-030-87250-1\_4.
- [2] Tim Soethout, Tjib van der Storm, and Jurgens J. Vrijs. Static local coordination avoidance for distributed objects. In 39th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control - AGERE 2019, doi:10.1145/3384846.3384920.
- [3] Tim Soethout, Tjib van der Storm, and Jurgens J. Vrijs. Full-circle admits commit: local coordination avoidance for distributed transactions. The Art, Science, and Engineering of Programming, 5(1):1, 2021. doi:10.22552/programming-journal.org/2021/5/1.

tim.soethout@ing.com

## Goal: Domain Knowledge to Faster Transactions

- Large distributed (enterprise) software systems are complex
- Consistency / Isolation  $\iff$  Scalability / Performance
- DSLs and models capture domain knowledge without implementation details
- Scope: Distributed concurrent objects with async messaging



## Problem: Bottleneck on high-contention objects

Tax office bank account:

- Strong consistency requirements
- Strict time bounds
- Many tax and benefits money transfers
- Potential bottleneck for high contention

## Implementing Sync with 2PL/2PC

Two-Phase Locking (2PL): Concurrency Control: Single object, No concurrent access to object

Two-Phase Commit (2PC): Atomic Commitment: Multiple objects, Well-understood and Often used

Combined 2PL/2PC: Serializable Isolation guarantees

A lot of waiting, but enough balance for both, right?  $\implies$

## Approach: Reduce coordination w/ Domain Knowledge

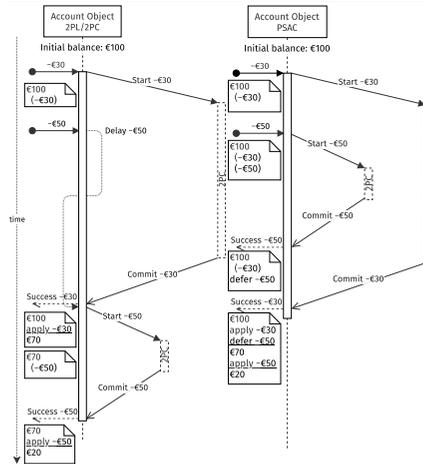
Insight: Enough balance for both withdrawals and the commit or abort of first operation does not influence second

Increase parallelism where it is safe

Enter **Path-Sensitive Atomic Commit (PSAC)**:

- Operations in parallel, when safe
- Less waiting/locking of objects
- Extra computing time vs. waiting on message IO

## 2PL/2PC vs. PSAC



## Evaluation: Performance with varying contention

Message passing actors implementation of 2PL/2PC and PSAC.

Experiment data/results available @ doi:10.5281/zenodo.3405371

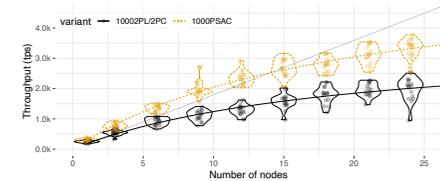
2PL/2PC is special case of PSAC with parallelism disabled

Experiments with varying contention:

- NOSYNC – Operations without synchronization
- SYNC – Uniform money transfers over 100.000 accounts
- SYNC 1000 – Uniform money transfers over 1000 accounts

## Results

Similar throughput for NOSYNC & SYNC, not enough contention



Under high-contention SYNC 1000: Up to 1.8 times higher median throughput

## Conclusion

- High contention bottleneck with 2PL/2PC
- Safe parallelism with PSAC; currently looking into isolation guarantees
- Promising for creating high-performant implementations from models



# Executive Summary

Large-scale enterprise IT systems are complex and hard to maintain. Domain models can help to manage complexity and separate implementation from business functionality.

Consistency and isolation guarantees on data are paramount in these systems, especially in the financial domain. The sector requires approaches and directions to maintain these guarantees and provide high performance, while implementing the domain models correctly.

Using contracts – consisting of operations with guards and effects of domain entities – the novel run-time algorithm Local-Coordination Avoidance runs more operations concurrently than general purpose algorithms, while maintaining consistency and isolation guarantees. This is backed by Return-Value Serializability, which is a formalization of high-level serializability, based on domain operations instead of low-level reads and writes. A (distributed) object locally computes Local-Coordination Avoidance with Contract-Based Commutativity. When all objects do this, this leads to global serializable behavior.

Performance evaluation on cloud hardware shows that throughput is increased up to 1.6 times in high-contention scenarios for non-conflicting operations and latency is on par or better.

Local-Coordination Avoidance can increase throughput for large-scale, highly scalable applications, while the underlying functional business logic does not have to change, even though more performance is gained. In the worst case it becomes clear where in the modeling these bottlenecks lie, and thus where functionality can be changed to circumvent this.

## Recommendations for ING Bank

- Isolation guarantees on transactions spanning multiple micro-services are often under-defined, e.g. should a long-running new account request be stopped when later in the process the earlier-approved customer is a fraudster or not? How does the software guard this? Historically, databases take care of this by rolling back transactions if earlier checks are invalidated, with calls to different distributed micro-services this is not automatically in place. ING can decide to implement the Local-Coordination Avoidance

## Executive Summary

algorithm inside micro-services, its middleware or software SDK to manage isolation, without blocking other operations.

- ING can decide to create an Akka-based platform, leveraging the software and research results, to replace core banking and other software.
- ING can assess where in current applications speed-up using non-conflicting operations is possible. Either by statically analyzing the code base and models to detect when operations are always non-conflicting or at run time by instrumenting existing applications to detect non-conflicting operations.
- ING could consider that a new niche of scalable applications is forming that – under different kinds of loads – still perform well, and thus that historical changes in domain logic because of performance issues (e.g. tax wash accounts, offline batch processing and shadow bookkeeping) are no longer necessary.
- ING could consider that when aiming for an always online worldwide bank, that always available and fast-response times to customer (web) requests requires local decisions operations without coordination.



## Titles in the IPA Dissertation Series since 2019

**S.M.J. de Putter.** *Verification of Concurrent Systems in a Model-Driven Engineering Workflow.* Faculty of Mathematics and Computer Science, TU/e. 2019-01

**S.M. Thaler.** *Automation for Information Security using Machine Learning.* Faculty of Mathematics and Computer Science, TU/e. 2019-02

**Ö. Babur.** *Model Analytics and Management.* Faculty of Mathematics and Computer Science, TU/e. 2019-03

**A. Afroozeh and A. Izmaylova.** *Practical General Top-down Parsers.* Faculty of Science, UvA. 2019-04

**S. Kisfaludi-Bak.** *ETH-Tight Algorithms for Geometric Network Problems.* Faculty of Mathematics and Computer Science, TU/e. 2019-05

**J. Moerman.** *Nominal Techniques and Black Box Testing for Automata Learning.* Faculty of Science, Mathematics and Computer Science, RU. 2019-06

**V. Bloemen.** *Strong Connectivity and Shortest Paths for Checking Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

**T.H.A. Castermans.** *Algorithms for Visualization in Digital Humanities.* Faculty of Mathematics and Computer Science, TU/e. 2019-08

**W.M. Sonke.** *Algorithms for River Network Analysis.* Faculty of Mathematics and Computer Science, TU/e. 2019-09

**J.J.G. Meijer.** *Efficient Learning and Analysis of System Behavior.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

**P.R. Griffioen.** *A Unit-Aware Matrix Language and its Application in Control and Auditing.* Faculty of Science, UvA. 2019-11

**A.A. Sawant.** *The impact of API evolution on API consumers and how this can be affected by API producers and language designers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

**W.H.M. Oortwijn.** *Deductive Techniques for Model-Based Concurrency Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

**M.A. Cano Grijalba.** *Session-Based Concurrency: Between Operational and Declarative Views.* Faculty of Science and Engineering, RUG. 2020-01

**T.C. Nägele.** *CoHLA: Rapid Co-simulation Construction.* Faculty of Science, Mathematics and Computer Science, RU. 2020-02

**R.A. van Rozen.** *Languages of Games and Play: Automating Game Design &*

*Enabling Live Programming*. Faculty of Science, UvA. 2020-03

**B. Changizi**. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04

**N. Naus**. *Assisting End Users in Workflow Systems*. Faculty of Science, UU. 2020-05

**J.J.H.M. Wulms**. *Stability of Geometric Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2020-06

**T.S. Neele**. *Reductions for Parity Games and Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2020-07

**P. van den Bos**. *Coverage and Games in Model-Based Testing*. Faculty of Science, RU. 2020-08

**M.F.M. Sondag**. *Algorithms for Coherent Rectangular Visualizations*. Faculty of Mathematics and Computer Science, TU/e. 2020-09

**D.Frumin**. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

**A. Bentkamp**. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VUA. 2021-02

**P. Derakhshanfar**. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineer-

ing, Mathematics, and Computer Science, TUD. 2021-03

**K. Aslam**. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

**W. Silva Torres**. *Supporting Multi-Domain Model Management*. Faculty of Mathematics and Computer Science, TU/e. 2021-05

**A. Fedotov**. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

**M.O. Mahmoud**. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

**M. Safari**. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

**M. Verano Merino**. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

**G.F.C. Dupont**. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

**T.M. Soethout**. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06