Contract-Based Return-Value Commutativity

Safely Exploiting Contract-Based Commutativity for Faster Serializable Transactions

Tim Soethout
ING Bank, Amsterdam
CWI, Amsterdam
The Netherlands
Tim.Soethout@ing.com

Tijs van der Storm
CWI, Amsterdam
University of Groningen, Groningen
The Netherlands
storm@cwi.nl

Jurgen J. Vinju
CWI, Amsterdam
Eindhoven University of Technology
The Netherlands
Jurgen.Vinju@cwi.nl

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 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.

CCS Concepts: • **Information systems** → *Distributed database transactions*; • **Software and its engineering** → *Domain specific languages*; State systems; Model-driven software engineering.

Keywords: distributed systems, transactions, distributed objects, model checking, coordination

ACM Reference Format:
Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. 2021. 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 '21), October 17, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3486601.3486707

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 [28] 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 paper 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

Table 1. All contributions, concepts and abbreviations introduced and referenced in this paper.

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	3
CBC*	Optimized variant of CBC, used in the LoCA implementation.	Contribution	3
SCBC	Static CBC, an encoding in SMT that allows computing static CBC for state-machine models, including an comparison between *SIE* and SCBC.	Contribution	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	4, 6, 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. [23, 25]	5, 6, 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 [10], TLA$^+$ model [24]	5, 6, 7
IE	Independent Events, a definition of independent operation pairs, guaranteeing local internal state machine consistency, but not global serializability.	Soethout et al. [23, 25]	5
SIE	Subset of *IE*, statically determined for all possible object states using an SMT solver.	Soethout et al. [23]	5
CI	Client-Centric Isolation model, on which RV-SER is inspired, based on low-level reads and writes.	Crooks et al. [7], TLA$^+$ model [24]	4, 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 [28]	2, 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 [28]	2, 3

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: operation(arguments)/effect$\uparrow$ return value. To prevent overdraft, `Withdraws` check if enough balance is available and only then returns an updated state: `Withdraw`$(a)/\textbf{if}(b \geq a)\ b - a\ \textbf{else}\ b \uparrow b \geq a$. `Deposits` always return success (OK): `Deposit`$(a)/b+a\uparrow$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 [23, 25] 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 paper 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 [23, 25] for CBC operations. The formalization of LoCA and RV-SER in TLA$^+$ also enables validating of run-time schedules of implementations.

This paper's contributions are detailed in Table 1 including section references and the most important abbreviations used. Section 2 gives background on the grounding of RV-SER and CBC. Sections 3 to 6 describe the main contributions. Section 7 evaluates the formalizations with model checking approach in TLA$^+$ to show that interleaved processes using LoCA indeed maintain RV-SER. Discussion of the work including threats to validity are found in Section 8. Lastly we discuss related work (Section 9), and conclude in Section 10. All source code and reproducibility scripts are available on Zenodo [22].

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 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 [28] 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.`Deposit`(x) and B.`Withdraw`(x)); or commutative (A.`Deposit`(x) and A.`Deposit`(y)).

A schedule with operations, A.`Withdraw`(x) and A.`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' [7] 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 σ. p and q are SDC if schedule $pq\omega \Longrightarrow qp\omega$, where the return values of p, q and all possible later operations ω should stay the same when p and q are swapped given state σ.

RVC. Return-Value Commutativity abstracts from a concrete run-time state and looks at all possible sequences of previous operations α, instead of only to a state σ: $\alpha pq\omega \Longrightarrow \alpha qp\omega$, 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. $+20_A \uparrow$ 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 ($+50_A \uparrow$ NOK) or implicit (`GetBalance`$_A \uparrow 100$) success (OK) or failure (NOK) return value. These return values should not differ when p and q are swapped.

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 5.1) 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 respectively SDC and RVC. Dynamic CBC is valid in a specific run-time state σ from SDC, where static CBC holds in all

Table 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 $s_q)$ $\wedge$ $(s_{pq}\equiv s_{qp})$

possible run-time states, corresponding with all possible previous sequences of operations α from RVC.

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 operated arrives, it decides wether 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 runtime state s and operations p and q we look at a simple abstraction, based on operations that can only return OK and NOK. Table 2 contains the different case distinctions possible, p and q either return OK or NOK. This directly corresponds with SDC's $pq\omega \implies qp\omega$. The matching return values in ω are over-approximated by equating the post state in both orders ($s_{pq}\equiv s_{pq}$), 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. 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:

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

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 $retVal(s,o): State \times Operation \longrightarrow ReturnValue$. 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 accoun t and a later query operation (part of ω) 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 7.

Example. Consider the following example with CBC, based on money transfers between two bank accounts without overdraft (balance $\geq$ € 0). There are three transactions $T_1 : B \xrightarrow{€10} A, T_2 : B \xrightarrow{€20} A, T_3 : A \xrightarrow{€30} B$, transferring money between the accounts A to B with a respective starting balance of € 0 and € 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: $+30_B-20_B-10_B \overset{\text{CBC}(B_{+30},-20,-10)}{\Longrightarrow} +30_B-10_B-20_B \overset{\text{CBC}(B,+30,-10)}{\Longrightarrow} -10_B-20_B+30_B$, where B_{+30} represents the state of B with effects of shown operation applied. So, if CBC(€ 130,−20,−10) and CBC(€ 100,+30,−10) hold, these schedules are compatible and the original schedule is serializable.

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

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

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 € 0, results in not allowing the same swaps and therefore not being serializable, since CBC(€ 0,+30,−10) does not hold:

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

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

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.

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 $\text{CBC}(s,[o_1,..,o_n],o_i)$, where $o_1,..,o_n$ 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 unit 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:

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

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

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

where O is de sequence of in-progress operations. This optimized version of CBC, dubbed CBC*, is used in the LoCA implementation in Section 5 to achieve serializable isolation with increased concurrency.

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 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. [7], referred to as Crooks' Isolation (CI), and the formalization in TLA$^+$ builds on earlier work [24].

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 paper 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 [28], 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 of 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:

$$\forall o \uparrow o' \in T.\mathit{retVal}(s_p,o) \equiv o' \qquad (2)$$

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 $\mathit{eff}(s,o)$. This is the same contact 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: $+10_A \uparrow$ OK;$+30_B \uparrow$ OK;$+20_A \uparrow$ OK;$-20_B \uparrow$ OK;$-30_A \uparrow$ OK;$-10_B \uparrow$ 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.

$$\overset{s_0}{\begin{Bmatrix} A \mapsto 0 \\ B \mapsto 100 \end{Bmatrix}} \xrightarrow{T_1} \overset{s_1}{\begin{Bmatrix} A \mapsto 10 \\ B \mapsto 90 \end{Bmatrix}} \xrightarrow{T_2} \overset{s_{12}}{\begin{Bmatrix} A \mapsto 30 \\ B \mapsto 70 \end{Bmatrix}} \xrightarrow{T_3} \overset{s_{123}}{\begin{Bmatrix} A \mapsto 0 \\ B \mapsto 100 \end{Bmatrix}}$$

For all transactions, as per the commit test above (Equation (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: $\mathit{retVal}(s_1,+20_A) = \text{OK}$ and $\mathit{retVal}(s_1,-20_B) = \text{OK}$. This means that this execution is serializable, and therefore the original schedule is compatible and also serializable.

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.

$$\overset{s_0}{\begin{Bmatrix} A \mapsto 0 \\ B \mapsto 0 \end{Bmatrix}} \xrightarrow{T_1} \overset{s_1}{\begin{Bmatrix} A \mapsto 10 \\ B \mapsto 0 \end{Bmatrix}} \xrightarrow{T_2} \overset{s_{12}}{\begin{Bmatrix} A \mapsto 30 \\ B \mapsto 0 \end{Bmatrix}} \xrightarrow{T_3} \overset{s_{123}}{\begin{Bmatrix} A \mapsto 0 \\ B \mapsto 30 \end{Bmatrix}}$$

Now the commit test for T_2 fails: $\mathit{retVal}(s_1,+20_A) = \text{OK}$ and $\mathit{retVal}(s_1,-20_B) = \text{NOK}$, 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$^+$. RV-SER is formalized in TLA$^+$ and confirms these examples. TLA$^+$ [15] is a formal specification language for action-based modeling of programs, algorithms and (distributed) systems [6, 10, 11, 17, 18]. TLA$^+$ models states and transitions and its accompanying model checker TLC checks properties on each state, providing counter examples 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 [22]. In Section 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 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 (1). LoCA is also implemented [23, 25] using the Akka actor framework, using 2PC for atomicity. LoCA is compatible with sdifferent consensus or atomic commit algorithms, such as Raft [19] and Paxos [14].

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 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.

Table 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

5.1 LoCA with Independent Events

In earlier work LoCA was used with (Statically) Independent Events ((S)IE) [23, 25]. In this paper 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 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 [8]) in which the resource's preconditions, effects and states are modeled (or generated from another specification), similarly to *SIE*'s analysis [23].

SCBC is a subset of CBC: $\text{SCBC}(p,q) = \forall s.\text{CBC}(s,p,q)$, 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 counter examples 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.

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 counter examples in Section 7. This shows that LoCA is indeed RV-SER.

RV-SER is formalized similarly to CI [24] in TLA$^+$. The formalization of 2PL/2PC and CBC are structured similarly to the formalization of 2PL/2PC and Crooks' Isolation [7] in related work [24]. Full TLA$^+$ source code is available online [22].

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 (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 (1).

LoCA's pseudo-specification is found in Listing 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

Listing 1. LoCA formalization

```
queued = {}; inProgress = []; delayed = [];
state = InitialState; operations = {}

while true:
  # receive any of the 2PC messages
  on receive of VoteRequest:
    either # Either vote yes
      o = pick s.t. CBC*(state, inProgress, o)
      inProgress += o
      reply VoteCommit(o)
    or # vote no/abort
      reply VoteAbort(o)
    or # or delay until dependent operations finish
      o = pick s.t. ¬CBC*(state, inProgress, o)
      delayed += o
  on receive of GlobalCommit(o):
    queued += o # queue for commit
  on receive of GlobalAbort(o):
    inProgress -= o
    delayed -= o

  # Apply all applicable
      queued (ready for commit/apply) operations
  while Head(inProgress) ∈ queued:
    inP = Head(inProgress)
    # track per transaction for
        RV-SER-check, including observed return value
    operations[inP.tId] += <inP, RetVal(inP, state)>;
    state = Eff(inP, state); # Apply Eff to state
    inProgress -= inP
    queued -= inP
    # if next delayed is CBC, then start voting
    while CBC*(state, inProgress, Head(delayed)):
      o = Head(delayed)
      either
        reply VoteCommit(o)
        inProgress += o
      or
        reply VoteAbort(o)
      delayed -= o
```

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 (1). Multiple conflict relations can be configured, including *IE*, *SIE*, CBC and SCBC.

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 where 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 counter example 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 [24], 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 counter example 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 counter example 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 counter example 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 3). The model checker find an error set of transactions, containing a pair of `Withdraw` and `Deposit` operations. Thet 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 counter example shows this.

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. A change in the modeling approach can have big impact in performance, e.g. a Covid vaccination appointment could be a 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[20, 21], 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 [23], 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* [23, 25] ($IE(s,p\uparrow\text{OK},q) = retVal(s_p,q) \equiv retVal(s,q)$). *IE* however, is not serializable, since does not consider ω 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 runtime 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 [23, 25].

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* [23, 25] show that LoCA increases throughput and reduces latency in high-contention scenarios. Since SCBC has similar Go results, as shown in Table 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 [23] 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 [26, 27] 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' [7] 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 paper 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.

8.1 Threats to Validity

8.1.1 Limitations. This approach focusses 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 [13], 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.

9 Related Work

CBC is powered by contacts and models of objects. This fits well with Domain-Driven Design [9] and Command-Query Responsibility Segregation. More work [4, 5] is being done on reusing models to increase parallelism and performance.

Coordination Avoidance, Confluence and CALM. Confluence[12] 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 [3]: 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 [12] 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 [20, 21] 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 [29], related to RedBlue Consistency [16], 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.

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 paper focusses 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 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.

Acknowledgments

We thank the anonymous reviewers for their thorough reviews and feedback, and ING Bank for funding this research.

References

[1] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph. D. Dissertation. Massachusetts Institute of Technology, USA.

[2] Marcos K. Aguilera and Douglas B. Terry. 2016. The Many Faces of Consistency. *IEEE Data Eng. Bull.* 39, 1 (2016), 3–13. http://sites.computer.org/debull/A16mar/p3.pdf

[3] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[4] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*. ACM Press, 6:1–6:16. https://doi.org/10.1145/2741948.2741972

[5] Susanne Braun, Annette Bieniusa, and Frank Elberzhager. [n. d.]. 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* (Online United Kingdom, 2021-04-26). ACM, 1–12. https://doi.org/10/gjs3st

[6] Marc Brooker, Tao Chen, and Fan Ping. 2020. Millions of Tiny Databases. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 463–478.

[7] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Elad Michael Schiller and Alexander A. Schwarzmann (Eds.). ACM, 73–82. https://doi.org/10.1145/3087801.3087802

[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340.

[9] Eric Evans and Eric J Evans. 2004. *Domain-driven design - tackling complexity in the heart of software*. Addison-Wesley.

[10] Jim Gray and Leslie Lamport. 2006. Consensus on Transaction Commit. *ACM Transactions on Database Systems* 31, 1 (2006), 133–160. https://doi.org/10.1145/1132863.1132867

[11] Jason Gustafson and Guozhang Wang. 2020. Hardening Kafka Replication. https://github.com/hachikuji/kafka-specification.

[12] Joseph M. Hellerstein and Peter Alvaro. 2019. Keeping CALM: When Distributed Consistency is Easy. *CoRR* abs/1901.01930 (2019). arXiv:1901.01930 http://arxiv.org/abs/1901.01930

[13] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press.

[14] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

[15] Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

[16] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *OSDI*. USENIX Association, 265–278.

[17] Microsoft. 2020. High-Level TLA+ Specifications for the Five Consistency Levels Offered by Azure Cosmos DB. https://github.com/Azure/azure-cosmos-tla.

[18] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (March 2015), 66–73. https://doi.org/10.1145/2699417

[19] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[20] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. 2018. Conflict-free Replicated Data Types (CRDTs). *CoRR* abs/1805.06358 (2018). arXiv:1805.06358 http://arxiv.org/abs/1805.06358

[21] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *SSS (Lecture Notes in Computer Science, Vol. 6976)*. Springer, 386–400.

[22] Tim Soethout. 2021. TimSoethout/cbc-artifacts: Artifacts for AGERE'21 paper "Contract-Based Return-Value Commutativity: Safely exploiting contract-based commutativity for faster serializable transactions". https://doi.org/10.5281/zenodo.5497756

[23] Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. 2019. 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, 21–30. https://doi.org/10.1145/3358499.3361222

[24] Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. 2020. Automated Validation of State-Based Client-Centric Isolation with TLA$^+$. 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 (Lecture Notes in Computer Science, Vol. 12524)*, Loek Cleophas and Mieke Massink (Eds.). Springer, 43–57. https://doi.org/10.1007/978-3-030-67220-1_4

[25] Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. 2021. Path-Sensitive Atomic Commit - Local Coordination Avoidance for Distributed Transactions. *The Art, Science, and Engineering of Programming* 5, 1 (2021), 3. https://doi.org/10.22152/programming-journal.org/2021/5/3

[26] Jouke Stoel, Tijs van der Storm, Jurgen Vinju, and Joost Bosman. 2016. 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, 13–20. https://doi.org/10.1145/2998407.2998413

[27] Jouke Stoel, Tijs van der Storm, and Jurgen Vinju. 2021. Modeling with Mocking. In *2021 IEEE 14th International Conference on Software Testing, Validation and Verification (ICST)*. 59–70. https://doi.org/10.1109/ICST49551.2021.00018

[28] Gerhard Weikum and Gottfried Vossen. 2002. *Transactional Information Systems*. Elsevier. https://doi.org/10.1016/c2009-0-27891-3

[29] Xin Zhao and Philipp Haller. 2018. 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, 23–32. https://doi.org/10.1145/3281366.3281372