Consistent Local-First Software: Enforcing Safety and Invariants for Local-First Applications

Mirko Köhler, George Zakhour, Pascal Weisenburger, and Guido Salvaneschi

***Abstract*—Local-first software embraces data replication as a means to achieve scalability and offline availability. A crucial ingredient of local-first software are mergeable data types, like conflict-free replicated data types (CRDTs), which feature eventual consistency by enabling processes to access data locally and later merge it with other replicas in an asynchronous manner. Notably, the merging process needs to adhere to application constraints for correctness. Ensuring such application-level invariants poses a challenge, as developers must reason about the replicated program state and resort to manual synchronization of specific application components to enforce the invariant.**

This paper introduces ConLoc (Consistent Local-First Software), a novel system designed to automatically enforce safety and maintain invariants in local-first applications. ConLoc effectively addresses the issue of preserving invariants in the execution of programs with replicated data types, including CRDTs. Our approach is able to verify the correctness of many CRDTs examined in the literature and in implementations, such the ones used in the Riak database. ConLoc ensures that applications are automatically synchronized correctly, resulting in substantial latency and throughput improvements when compared to sequential execution, while upholding the same set of invariants.

***Index Terms*—replication, consistency, verification, CRDT, Java.**

I. Introduction

DISTRIBUTED applications consist of multiple processes running in parallel on possibly geo-distributed machines. These processes communicate changes to coordinate and achieve a common global view of the application state. For the coordination strategy, applications have the choice between two options. First, they may coordinate often using a consensus algorithms like Paxos [1], [2] for example, to agree on an order of operations at the expense of increasing latency and impeding efficient execution. Second, they may coordinate occasionally and operate locally on replicas, allowing for rapid progress at the expense of encountering and handling temporary data inconsistencies or in some cases, permanent conflicts.

Local-first software [3] is software where processes interact solely with local replicas and *eventually* merge their modifications and resolve conflicts. This solution enhances first *latency* since access to local replicas is fast, second *scalability* since clients access replicas in parallel, and third *offline availability* since processes can function solely on local data [4].

Mirko Köhler is with Technische Universität Darmstadt, Germany. E-mail: koehler@cs.tu-darmstadt.de

George Zakhour is with the Universität St. Gallen, Switzerland. E-mail: george.zakhour@unisg.ch

Pascal Weisenburger is a post-doctoral researcher at the Universität St. Gallen, Switzerland. E-mail: pascal.weisenburger@unisg.ch

Guido Salvaneschi is a professor at the Universität St. Gallen, Switzerland. E-mail: guido.salvaneschi@unisg.ch

The design of local-first applications, however, poses fundamental challenges compared to centralized systems where data is stored in a single place. A major problem is resolving state modification conflicts; when multiple processes concurrently modify their local replicas. Over time, numerous approaches [5], [6] have emerged that use *mergeable replicated data types* to ensure convergence *by design*. Most notably, *conflict-free replicated data types* (CRDTs) [7], [8].

CRDTs enable automatic conflict resolution. Implementations across a broad range of data structures exist: lists and sets [9], [10], trees [11], JSON data [12], and rich text documents [13]. In practice, they have been adopted to ensure scalability by Facebook in the Apollo low-latency “consistency at scale” database [14], by PayPal to manage compliance statuses between data centers with dynamic infrastructures [15], by TomTom to propagate navigation data among a user’s devices [16], and by numerous databases [17], [18], [19].

These data types implement *Eventual Consistency* [20]: replicas can be temporarily inconsistent but eventually converge when the replicas communicate all the changes. We refer to Eventually Consistent operations as *Weak*.

While Eventual Consistency boosts availability and performance, it might compromise correctness. For example, in an online shopping scenario, a buyer may locally record a successful payment, while the seller locally records a cancellation. Only after merging, the buyer will see the cancellation, i.e., they may temporarily see the approval before the system eventually reaches consistency.

To ensure correctness, such situations require a stronger consistency level where operations are executed as if they were on a single centralized system regardless of data distribution, i.e., requiring immediate coordination between the buyer and seller. We refer to operations that require that all nodes have a consistent view of the data at all times [2] through *strict serializability* [21] as *Strong*.

Sadly, Strong consistency sacrifices several advantages of local-first software such as offline availability. Hence, it is crucial that only as few operations as possible operate with Strong consistency. For this reason, when using mergeable data types, developers carefully choose the operations that need Strong consistency – for those, they manually implement coordination on top of the default Eventual Consistency.

A common approach to reason about software correctness is to consider *application invariants*. Yet, enforcing invariants on replicated data as required by local-first software is complex since modifications can be executed concurrently on different replicas. For example, a bank account should always have a positive balance. An implementation with only Weak operations cannot consistently ensure a non-negative balance, as concurrent withdrawals may result in a negative balance when merged everywhere. Thus, it is not enough that invariants are satisfied for the local state, instead they must also hold when states are merged.

In summary, when enforcing invariants with local-first software, developers must reason about the local state of a process and about concurrent modifications. To enforce such invariants, additional synchronization is needed.

The decision to employ Strong or Weak, however, is a balancing act: specifying the wrong consistency to operations may introduce bugs, and excessive synchronization reduces the application's availability; a Strong operation labeled Weak will be buggy, and a Weak operation labeled Strong will sacrifice the ability to provide continuous operations [22].

In this paper, we introduce a technique to automate the correct labeling of Strong and Weak operations. We introduce ConLoc, a system to enforce correct execution of local-first software with mergeable data types, such as CRDTs, ensuring that application invariants hold when executing operations concurrently. Our system uses techniques from software verification to deduce the correct consistency level for each method, enforcing the application-level invariants.

Our approach is the first to provide the combination of the following crucial features: (1) Unlike functional data structures [23], [24], [25], ConLoc targets imperative languages with mutable data. (2) Unlike approaches that only provide a verification framework for Weak execution [25], [26], [27], [28], ConLoc not only provides a verification framework, but also a code generation system and a middleware to ensure the correct execution of applications mixing Weak and Strong operations. (3) In contrast to other approaches [29], [30], ConLoc focuses on state-based replication, which requires a different treatment than operation-based replication (Section VI-C).

To this end, the verification capitalizes on object encapsulation provided by object-oriented languages, such as the widely adopted C++, C#, Java, and Kotlin. For the purposes of this work, our focus is primarily directed towards Java due to its prevalence [31]. In summary, our contributions are as follows:

- We design ConLoc, a system to ensure application-level invariants (i.e., expressed in the domain of the application, like *"the bank account should be positive"*) to provide *safe local-first* software with mergeable data types. ConLoc automatically synthesizes correct synchronization of operations, thereby guaranteeing safe execution.
- We implement ConLoc for Java, and develop the verification of invariants on top of the Z3 SMT solver.
- We evaluate ConLoc on case studies from literature, implement a new CRDT library, and migrate the Riak CRDT library to ConLoc. The results demonstrate that adopting ConLoc in existing applications requires limited effort, the verification is efficient and ConLoc improves the performance of such applications compared to a correct but conservative baseline using Strong operations only.

II. Towards Safe Local-First Apps

We motivate our approach with a running example of a distributed account that holds client credits for an online shop.

```java
1  public class CreditAccount {
2    public final ReplicatedCounter credits;
3
4    public CreditAccount() {
5      credits = new Counter(/*initial value*/ 0);
6    }
7
8    public int getValue() {
9      return credits.getValue();
10   }
11
12   public void deposit(int val) {
13     if (val < 0) throw new Exception();
14     credits.increment(val);
15   }
16
17   public void withdraw(int val) {
18     if (val < 0 || val > getValue())
19       throw new Exception();
20     credits.decrement(val);
21   }
22
23   public void merge(CreditAccount other) {
24     credits.merge(other.credits);
25   }
26 }
```

Fig. 1: Distributed implementation of CreditAccount.

A. Version 1: Distributed Implementation

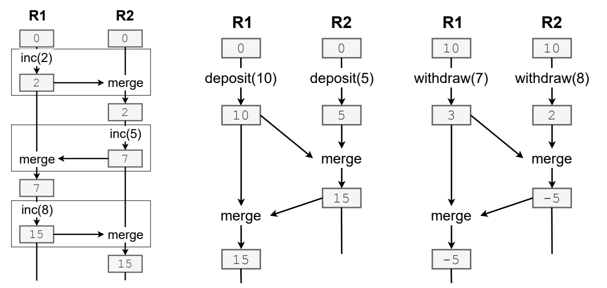
Figure 1 shows the account that stores credits. The `CreditAccount` class represents a customer's credit account with three operations: retrieving (Line 8), depositing (Line 12), and withdrawing credits (Line 17). Internally, `CreditAccount` utilizes a mutable counter (Line 2) with increment, decrement, and get operations. Depositing and withdrawing mutates the object's state as opposed to returning a new (modified) counter as one might expect in a functional implementation.

Both operations check that the argument (`val`) adheres to the precondition; depositing or withdrawing a negative amount is disallowed, and withdrawing is possible when enough credits exist (Line 19). Therefore the *application invariant* is that the `CreditAccount` balance is non-negative, i.e., `getValue() >= 0`.

Clients, logically single-threaded, connect to the application and perform operations on `CreditAccount`. Each client accesses a single *replica*, i.e., an instance of `CreditAccount` hosted on its own server. Replicas accept *operations* and apply them locally. However, replicas have to remain consistent with each other. One solution is to immediately coordinate the states at every client access (Figure 2a). For example, when replica R1 deposits 2 credits, R2 must wait until the operation is finished. In this case, whenever an operation is requested, the replicas coordinate until the operation is executed on all replicas, resulting in a sequential execution of operations (*Strong* consistency). While this solution satisfies the application invariant, it compromises availability.

Instead, to ensure availability, we drop immediate coordination and allow replicas to asynchronously propagate their state to other replicas (*Weak* consistency). Upon receiving a state, a replica merges it into its own using some strategy which is called by the middleware as part of the state propagation.

Rather than devising a merge strategy from the ground up, we use a CRDT as `ReplicatedCounter`. By design, CRDTs support *Eventual Consistency*: local states can be immediately modified and sent later, still ensuring that replicas eventually converge to a consistent state. The CRDT counter's merge strategy retains a mutable data structure that associates each



(a) Sequential. (b) Asynchronous. (c) Unsafe execution.

Fig. 2: Execution of replicated data types.

individual replica with the number of increments applied to it. The `merge` method of the `CreditAccount` can then simply call the underlying counter's `merge` function (Line 23).

Figure 2b shows a potential execution of an account: Both replicas start with state 0. R1 deposits 10 credits; R2 concurrently deposits 5. As updates change the local state, R1 has 10 credits and R2 has 5. Eventually, R1 sends its state to R2, which merges it with its own, resulting in 15. After R2 sends its state to R1, both replicas hold the (correct) state 15.

While a CRDT provides the required availability and merge logic, ensuring that the credit account cannot be negative poses a challenge. As with Strong consistency, one could implement `withdraw` to call the respective decrement operation on the CRDT only when the withdrawn amount is within limits of the account. However, with Weak consistency, this approach leads to a flawed application. Consider Figure 2c, in which both replicas are initially in state 10. R1 withdraws 7 while R2 withdraws 8. Both operations pass the check in the withdraw method since there are sufficient credits. Then R1 sends its state to R2. However, when R2 merges that state the result is -5, violating the invariant. So, checking the credits amount before withdrawal (Figure 1, Line 19) is only valid with Strong consistency where all operations are applied sequentially.

The solution is to limit the `withdraw` method – which is the one with the potential to violate the invariant – to Strong consistency, i.e., to prevent it from being called concurrently and avoiding withdrawals by two processes *at the same time*. While there exist solutions that can reduce the amount of coordination such as Bounded Counters [32], the need for coordination can not be removed completely.

Overall, the goal is to adopt Weak consistency as much as possible without violating any application invariants. Yet, deciding the appropriate consistency level is not straightforward and requires knowledge about the merge strategy. To demonstrate, consider the case of a CRDT with a `reset` operation, in addition to the operations already discussed, which sets the credits back to 0. The `reset` method cannot violate the invariant by itself as it can never yield a negative balance. A developer may thus conclude that `reset` can always be executed concurrently with other operations and assign it Weak consistency. Yet, this is wrong. When `reset` occurs concurrently with `withdraw` the balance can become negative. Consequently, the concurrent execution violates the invariant even if a sequential execution (`withdraw` followed by `reset`) does not. In a correct solution, to preserve the invariant, `reset` executes with Strong consistency.

```
1  public class CreditAccount {
2    public final ReplicatedCounter credits;
3
4    //@ invariant getValue() >= 0;
5
6    /* Initial Precondition on Constructor */
7    //@ ensures getValue() == 0;
8    public CreditAccount() { /* ... */ }
9
10   //@ ensures \result == \old(credits.getValue());
11   public int getValue() { /* ... */ }
12
13   //@ requires val >= 0;
14   //@ assignable credits;
15   //@ ensures stateful( credits.increment(val) );
16   public void deposit(int val) { /* ... */ }
17
18   //@ requires 0 <= val && val <= getValue();
19   //@ assignable credits;
20   //@ ensures stateful( credits.decrement(val) );
21   public void withdraw(int val) { /* ... */ }
22
23   //@ requires (\sum int i; i >= 0 && i < numOfReplicas();
           Math.max(credits.incs[i], o.credits.incs[i]) −
           Math.max(credits.decs[i], o.credits.decs[i])) >= 0;
24   //@ ensures stateful( credits.merge(other.credits) );
25   public void merge(CreditAccount o) { /*...*/ }
26 }
```

Fig. 3: Credit account with invariants.

In summary, CRDTs enhance availability and ensure that replicas eventually converge *by design*. However, maintaining application invariants requires additional synchronization logic. Too many synchorizations eliminates the benefits of CRDTs, while too few synchronizations leads to buggy applications.

B. Version 2: Safe Local-First Applications

We design ConLoc, a system to ensure the safety of replicated operations with mergeable data types *and* application invariants. ConLoc automatically verifies which methods require Strong consistency and where Weak consistency suffices. Thus, ConLoc enhances an application to run as a local-first application (i.e. Weak) where possible but imposes synchronization (i.e. Strong) when safety requires it.

Similar to Version 1, developers employ a CRDT counter. With ConLoc they further provide the invariant as well as pre- and postconditions of operations and the merge method as *annotations*. Figure 3 shows the `CreditAccount` example with additional annotations (based on JML [33]).

Initially, developers define the invariant, asserting that the account balance is not negative (Line 4) which ConLoc ensures is upheld even under concurrent execution. As the `getValue` method is pure – deterministic and non-mutating – it can freely appear in invariants, pre-, and postconditions.

For each method, developers specify pre- and postconditions. Preconditions (`requires` keyword) define when a method can be execute. For instance, `deposit` requires that the amount is non-negative (Line 13). Postconditions (`ensures` keyword) define the effect of a method on the return value and state. In the postcondition of `getValue` (Line 10), `\result` is the return value and `\old` refers to the object before method execution. The `stateful` construct captures the effect of calling another method. In the example, we assert that the state of `credits` after `deposit` is the same as executing `increment` on `credits` (Line 15).

Developers also specify which fields are mutated (`assignable` keyword), e.g., the `credit` field in `deposit` (Line 14), and no field

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathsf{x}, \mathsf{f}, \mathsf{m}, \mathsf{C} \in \mathit{Identifiers},\ \mathsf{lit} \in \mathit{Literals}} \\
\mathit{ClassDecl} & ::= & \texttt{class}\ \mathsf{C}\ \{@\mathit{inv}\ \mathit{FieldDecl}^*\ \mathit{MthdDecl}^*\} \\
@\mathit{inv} & ::= & \textbf{invariant}\ e; \\
\mathit{FieldDecl} & ::= & T\ \mathsf{f}; \\
\mathit{MthdDecl} & ::= & @\mathit{mthd}\ T_0\ \mathsf{m}(T_1\ \mathsf{x})\ \{\ \mathit{java\_stmt}^*\ \} \\
@\mathit{mthd} & ::= & \textbf{requires}\ e;\ \textbf{assignable}\ \mathit{assign}^*;\ \textbf{ensures}\ e \\
\mathit{assign} & ::= & \mathsf{f} \mid \mathsf{f}[\mathsf{x}] \mid \mathsf{f}[\star] \\
e & ::= & \mathsf{lit} \mid \mathsf{x} \mid e == e \mid e \oplus e \mid e[e] \\
& & \mid \texttt{this} \mid e.\mathsf{f} \mid e.\mathsf{m}(e) \mid \textbf{forall}(T\ \mathsf{x}; e; e) \\
& & \mid \textbf{exists}(T\ \mathsf{x}; e; e) \mid \textbf{sum}(\texttt{int}\ \mathsf{x}; e; e) \\
& & \mid \texttt{replicaId} \mid \texttt{numOfReplicas} \mid \textbf{old}(e) \mid \textbf{result} \\
& & \mid \textbf{stateful}(e.\mathsf{m}(e)) \\
T & ::= & \texttt{int} \mid \texttt{boolean} \mid \ldots \mid T[] \mid \mathsf{C}
\end{array}
$$

Fig. 4: Core language of ConLoc.

in `getValue`. The initial state is constrained by preconditions on the constructor, e.g., that the initial value of the counter is 0 (Line 7). Since invariants must always hold, including before and after one of the class methods are invoked, all method pre- and post-conditions are conjoined with the class invariants.

For `merge` (Line 25), the precondition specifies when two states are *mergeable*. If the precondition (Line 23) is satisfied, then merging does not result in a negative counter. The postcondition uses **`stateful`**, indicating that merging two `CreditAccount` instances involves merging the underlying counters.

ConLoc enables *composing* replicated data types by referring to method calls in specifications, which is crucial as operations on one data type (e.g., `CreditAccount`) are often expressed with operations on another (e.g., counter CRDT).

ConLoc identifies the set of operations that can potentially violate the invariant if executed concurrently. The corresponding methods are assigned Strong consistency resulting in a sequential execution that requires coordination. In the example, `withdraw` executes with Strong consistency while the other methods execute with Weak consistency, enhancing availability.

III. Design of ConLoc

Based on the observations in II-B, in this section, we systematically present ConLoc, our system for safe local-first software with application invariants.

A. Core Language

We introduce a core language for ConLoc, which provides a systematic treatment of the constructs informally introduced in Figure 3, based on a Java-like language with extended syntax for method pre- and postconditions. A program is a set of classes, (Figure 4), which consist of an invariant and a sequence of field and method declarations. An invariant $@\mathit{inv}$ consists of an annotation expression e. Fields have a type T and a name f. Methods have a name m, a return type T_0, and a (single, for simplicity) parameter x of type T_1. Methods have annotations $@\mathit{mthd}$ for preconditions (**requires**), postconditions (**ensures**), and fields that are mutated (**assignable**). For arrays, a developer can specify which element is mutated, either one ($\mathsf{f}[\mathsf{x}]$) or all ($\mathsf{f}[\star]$).

Annotations use expressions e, which include literals, e.g., integers and booleans. Variables x refer to local variables.

$$
\begin{array}{rcl}
[\![\mathsf{lit}]\!] & \equiv & \mathsf{lit} \\
[\![e_1 == e_2]\!] & \equiv & [\![e_1]\!] = [\![e_2]\!] \\
[\![e_1 \oplus e_2]\!] & \equiv & [\![e_1]\!] \hat{\oplus} [\![e_2]\!] \\
[\![\mathsf{x}]\!]^{\Theta} & \equiv & \Theta(\mathsf{x}) \\
[\![e_1[e_2]]\!] & \equiv & [\![e_1]\!]([\![e_2]\!]) \\
[\![\texttt{this}]\!]_{\mathit{this},\cdot,\cdot} & \equiv & \mathit{this} \\
[\![e.\mathsf{f}]\!] & \equiv & \mathit{fld}_{\mathsf{f}}([\![e]\!]) \\
[\![e_0.\mathsf{m}(e_1)]\!] & \equiv & \mathsf{m}_{\mathit{result}}([\![e_0]\!], [\![e_1]\!]),\ \text{if } \mathsf{m} \text{ is pure} \\
[\![\textbf{stateful}(e_0.\mathsf{m}(e_1))]\!]_{\cdot,\mathit{old},\cdot} & \equiv & [\![e_0]\!] = \mathsf{m}_{\mathit{state}}([\![\textbf{old}(e_0)]\!], [\![\textbf{old}(e_1)]\!]) \\
& & \text{if } \mathit{old} \neq \bot \\
[\![\textbf{forall}(T\ \mathsf{x}; e_0; e_1)]\!]^{\Theta} & \equiv & \forall \gamma \in T.\ [\![e_0]\!]^{[\Theta, \mathsf{x} \to \gamma]} \Rightarrow [\![e_1]\!]^{[\Theta, \mathsf{x} \to \gamma]} \\
[\![\textbf{exists}(T\ \mathsf{x}; e_0; e_1)]\!]^{\Theta} & \equiv & \exists \gamma \in T.\ [\![e_0]\!]^{[\Theta, \mathsf{x} \to \gamma]} \wedge [\![e_1]\!]^{[\Theta, \mathsf{x} \to \gamma]} \\
[\![\textbf{sum}(\texttt{int}\ \mathsf{x}; e_0; e_1)]\!]^{\Theta} & \equiv & \sum_{n \in \{n \in \mathbb{Z} \mid [\![e_0]\!]^{[\Theta, \mathsf{x} \to n]}\}} [\![e_1]\!]^{[\Theta, \mathsf{x} \to n]} \\
[\![\texttt{numOfReplicas}]\!] & \equiv & \text{some } \mathsf{c}_0 \in \mathbb{N} \\
[\![\texttt{replicaId}]\!] & \equiv & \text{some } \mathsf{c}_1 \in \mathbb{N} \text{ with } \mathsf{c}_1 < \mathsf{c}_0 \\
[\![\textbf{old}(e)]\!]_{\mathit{this},\mathit{old},\mathit{res}} & \equiv & [\![e]\!]_{\mathit{old},\mathit{old},\mathit{res}},\ \text{if } \mathit{old} \neq \bot \\
[\![\textbf{result}]\!]_{\cdot,\cdot,\mathit{res}} & \equiv & \mathit{res},\ \text{if } \mathit{res} \neq \bot
\end{array}
$$

Fig. 5: Translation function for expressions.

$e == e$ checks for equality, and operators $\oplus$ include, for example, `+`, `–`, `&&` and `<`. The expression $e[e]$ selects an element of an array. The current object is accessed by `this`. Fields are accessed by $e.\mathsf{f}$. Methods are called with $e.\mathsf{m}(e)$. The **forall** or **exists** constructs quantify over types T and take boolean expressions for the range and for the body. Summation **sum** is restricted to range over integers as the underlying solver Z3 requires a fixed range. `replicaId` is an identifier of the current replica, and `numOfReplicas` is the total number of replicas. $\textbf{old}(e)$ is the state before executing the method by replacing `this` in e with old. **result** refers to the return value. **stateful** turns impure method calls into constraints by relating the states before and after a method call. For example, in Figure 3, in Line 20, the **`stateful`** keyword refers to the `credits` object itself after the mutation rather than to the return value. Types T supported by constraints are all base types, such as `int` and `boolean`, array types $T[]$, and class types C.

B. From Annotations to Constraints

ConLoc employs the annotations and class specifications to define data types $\mathcal{D}$. For every class, ConLoc defines a mathematical data type that consists of 1) an internal (local) state S, i.e., a tuple of its fields, 2) a set of methods, and 3) a set of constraints, i.e., logical formulas that specify the behavior of the data type generated from the annotations.

1) Expression Translation: We first define translation from annotations to mathematical constraints V for expressions (Figure 5). The translation function $[\![e]\!]^{\Theta}_{\mathit{this},\mathit{old},\mathit{res}}$ takes the expression e and three additional parameters. The parameter $\mathit{this} \in V$ refers to state of the object which contains e. The two parameters $\mathit{old}, \mathit{res} \in V \cup \{\bot\}$ is the state of this object before executing a method (old), and the return value (res). If e is not in a method, old and res are $\bot$. Store $\Theta : \mathsf{x} \to V$ maps local variables to the respective constraints. We write $[\![e]\!]$ when the other parameters are clear. For literals, equality and operators $\oplus$, the translation is straightforward. Variables x are looked up in the local store Θ. Arrays with elements of type T are partial maps $\mathbb{Z} \mapsto T$ in constraints: $e_1[e_2]$ accesses the element at index e_2 in the array e_1. `this` is the *this* argument of the translation function.

We introduce three auxiliary functions, defined later: 1) $\mathit{fld}_\mathsf{f} : S \to T$ selects the value of a field with type T from a state S. 2) $\mathsf{m}_{result} : S \times T_0 \to T_1$ takes the state of an receiver object S and argument T_0 of a call to m, and gives the return value. 3) $\mathsf{m}_{state} : S \times T \to S$ returns the state of the receiver after the method call. Fields are selected from an object by applying fld_f. Method calls $e_0.\mathsf{m}(e_1)$ translate as m_{result}. This can only be applied when m is pure, i.e., when its **assignable** annotation is empty. **stateful** turns impure method calls into constraints by relating the states before and after a method call using m_{state}. **forall**, **exists**, and **sum** are translated easily to constraints. Sums are unrolled, which is possible as the range is fixed at compile time. `replicaId` and `numOfReplicas` are chosen arbitrary but fixed. $\mathbf{old}(e)$ translates e with `this` changed to *old*. **result** is the return value.

2) Invariants: We define how to built invariants from classes. An invariant constraint $\mathcal{I}_\mathsf{C}$ of class C is a constraint on the state s of an object, i.e., $\mathcal{I}_\mathsf{C}(s)$ means that the object in state s satisfies the predicate $\mathcal{I}_\mathsf{C}$. We write $\mathcal{I}$ if C is clear. From annotations **invariant** e, we generate invariant constraints $\mathcal{I}$ by translating e.

$$\mathcal{I}_\mathsf{C}(s) \equiv [\![e]\!]^{[]}_{s,\bot,\bot}$$

We apply the translation with an empty initial store, as invariants do not contain free variables. *old* and *res* are $\bot$, as invariants have no return values or states before execution.

3) Pre- and Post-Constraints: From every method's annotations, we derive pre- and post-constraints. In the following, let method m be declared as $@mthd\ T_0\ \mathsf{m}(T_1\ \mathsf{x})$ where $@mthd$ is **requires** e_{req}; **assignable** fs; **ensures** e_{ens}. Method pre-constraints $\mathsf{m}_{pre}(s, e)$ are constraints on the object state s and the method argument e. Method m has pre-constraint:

$$\mathsf{m}_{pre}(s, e) \equiv [\![e_{req}]\!]^{[\mathsf{x}\mapsto[\![e]\!]]}_{s,\bot,\bot}$$

We evaluate the annotation e_{req} with the current state of the object s as `this`. The store maps the parameter x to the translated argument e. As with invariants, *old* and *res* are $\bot$.

Method post-constraints $\mathsf{m}_{post}(s, s', r, e)$ refer to the object state *before* (s) and *after* (s') executing the method, to the return value r, as well as to the argument e. Post-constraints are derived from the **ensures** and **assignable** annotations. The former are translated similar to pre-constraints, and the latter are translated such that the states s and s' are equal for every field f that is *not* assigned.

$$\mathsf{m}_{post}(s, s', r, e) \equiv [\![e_{ens}]\!]^{[\mathsf{x}\mapsto[\![e]\!]]}_{s',s,r} \wedge \bigwedge_{\mathsf{f}\notin \mathit{fs}} (\mathit{fld}_\mathsf{f}(s) = \mathit{fld}_\mathsf{f}(s'))$$

A sequential precondition of a method $\mathsf{m}_{pre}(s,e)$ is the weakest precondition [34], [35], [36] for that method such that the postcondition $\mathsf{m}_{post}(s, s', r, e)$ holds.

4) Initial Constraints: We define a constraint *init* for the initial state, i.e., $\mathit{init}(s)$ is satisfied for the initial state. Initial constraints are derived from postconditions of constructors.

5) Class Types: We show how class types are represented in constraints. Let class C have the declaration `class` C $\{\mathit{inv\ fields\ methods}\}$ with $\mathit{fields} = T_0\ \mathsf{f}_0; ...T_n\ \mathsf{f}_n;$. We model the state $s \in S$ of an object (of class C) as a tuple of the field values $(v_0, ..., v_n) \in T_0 \times ... \times T_n$. For every field f, we additionally define the function fld_f to select the field f from a tuple. For every method m of C, we define two functions for computing 1) the result of the method, and 2) the state of C after executing the method. Let $@mthd\ T_0\ \mathsf{m}(T_1\ \mathsf{x})\ \{\ java\_stmt^*\ \}$ be the method declaration of m in C, then these functions are:

- $\mathsf{m}_{result} : S \times T \to T$ with
  $\mathsf{m}_{result}(s, e) = r \ \Leftrightarrow\ \mathsf{m}_{post}(s, s', r, e)$
- $\mathsf{m}_{state} : S \times T \to S$ with
  $\mathsf{m}_{state}(s, e) = e' \ \Leftrightarrow\ \mathsf{m}_{post}(s, s', r, e')$

For merge, we have a similar state function that takes two states instead: $\mathit{merge}_{state} : S \times S \to S$.

6) Merge Constraints: The merge method is treated analogously to other methods. Thus, its pre-constraints, post-constraints, and state are accessible through the merge_{pre}, merge_{post}, and merge_{state} functions which instantiate m in m_{pre}, m_{post}, and m_{state} with *merge* respectively. Additionally, we verify that *merge* satisfies the constraints of being commutative, idempotent, and associative – in line with CRDTs [7].

7) Example: We demonstrate the translation procedure for some constraints in the running example. The **invariant** of `CreditAccount` (Figure 3, Line 4) results in the constraint:

$$\mathcal{I}(s) \ \equiv\ \mathit{getValue}_{result}(s) \geq 0 \tag{1}$$

We can call *getValue* in the invariant, because it is a pure method. For `getValue` (Figure 3, Line 10) the field `credits` is not assigned, generating the following post-constraint:

$$\begin{aligned}\mathit{getValue}_{post}(s, s', r) \ \equiv\ & r = \mathit{getValue}^{CRDT}_{result}(\mathit{fld}_{credits}(s)) \\ & \wedge \mathit{fld}_{credits}(s) = \mathit{fld}_{credits}(s')\end{aligned} \tag{2}$$

Here, $\mathit{getValue}^{CRDT}_{result}$ refers to the *getValue* method of the underlying CRDT. The precondition for `withdraw` (Figure 3, Line 18) is:

$$\mathit{withdraw}_{pre}(s, \mathsf{x}) \ \equiv\ 0 \leq \mathsf{x} \wedge \mathsf{x} \leq \mathit{getValue}_{result}(s) \tag{3}$$

The **ensures** annotation for `withdraw` (Figure 3, Line 20) has an impure method call:

$$\begin{aligned}&\mathit{withdraw}_{post}(s, s', r, \mathsf{x}) \ \equiv \\ &\qquad \mathit{fld}_{credits}(s') = \mathit{decrement}^{CRDT}_{state}(\mathit{fld}_{credits}(s), \mathsf{x})\end{aligned} \tag{4}$$

As above, $\mathit{decrement}^{CRDT}_{state}$ is the *decrement* method in the underlying CRDT. The initial constraint for `CreditAccount` is:

$$\mathit{init}(s) \ \equiv\ \mathit{getValue}_{result}(s) = 0 \tag{5}$$

C. From Constraints to Properties

From each class's constraints, we derive *properties* which, when proven, (1) guarantee that the class's operations always satisfy its invariant and (2) imply a consistency specification for which methods can be safely executed with Weak consistency.

First, we recap the safety framework by Nair et al. [37]. Then, we define the execution model of ConLoc. Finally, we show that the execution model satisfies the safety framework.

1) Safety Framework: We base our work on a verification technique that introduces two groups of properties: *invariant sufficiency* and *mergeability* [37] for a transition system (TS).

Invariant sufficiency states that the invariant is satisfied under sequential execution. If it is not satisfied, even sequentially, it can never be (also not under concurrent execution). Mergeability specifies that an object can be merged with another (possibly concurrently modified) object. To prove a program *safe*, i.e., the (distributed) execution cannot violate the invariant, it is key to prove invariant sufficiency and mergeability.

a) Invariant Sufficiency: For the formal specification we first define the global invariant $\hat{\mathcal{I}}$ which holds for a global state $\hat{s}$ when the local invariant $\mathcal{I}$ holds for all of its local states, i.e., $\hat{\mathcal{I}}(\hat{s}) = \forall i \in \mathcal{P}.\ \mathcal{I}(s_i)$.

Definition 1 (Invariant sufficiency)**.** *The TS* $(\hat{S}, \rightarrow)$ *is called* invariant-sufficient w.r.t. $\mathcal{I}$ *iff for all global states* $\hat{s}_1$ *and* $\hat{s}_2$ *holds* $\hat{\mathcal{I}}(\hat{s}_1) \wedge (\hat{s}_1 \rightarrow \hat{s}_2) \Rightarrow \hat{\mathcal{I}}(\hat{s}_2)$.

We will now define properties for mergeable data types that enable invariant-sufficient executions. First, the initial state of the data type has to satisfy the invariant.

$$init(s) \Rightarrow \mathcal{I}(s) \qquad \text{(i0)}$$

For each method of the data type, if the preconditions are satisfied, then the invariant holds after execution of the method.

$$\mathcal{I}(s) \wedge \mathsf{m}_{pre}(s, e) \Rightarrow \mathcal{I}(\mathsf{m}_{state}(s, e)) \qquad \text{(i1)}$$

These two properties ensure safety in a sequential execution but they do not ensure safety in a *concurrent* program where states are merged and may no longer satisfy the invariant afterwards.

Invariant violation example. The invariant of `ReplicatedCounter` states that the value must be positive, i.e., $\mathcal{I}(s) = getValue_{result}(s) \geq 0$. A decrement method dec that decrements the counter by 1 has the precondition $dec_{pre}(s) = getValue_{result}(s) \geq 1$, ensuring dec to not violate the invariant. Yet, when merging, the following execution with the states replicated to two replicas (s_1, s_2) is possible:

$$(1,1) \xrightarrow{dec(s_1)} (0,1) \xrightarrow{dec(s_2)} (0,0) \xrightarrow{s_1 \leftarrow merge(s_1,s_2)} (-1,0)$$

In the last state, the invariant does not hold for process p_1 despite *dec*'s precondition being satisfied in every step.

b) Mergeability: Two states s_1 and s_2 are *mergeable*, denoted by the symmetric and reflexive relation $\mathcal{M}(s_1, s_2)$ if they do not violate the invariant after merging them. $\mathcal{M}$ is the *weakest precondition* for $merge$ w.r.t. the invariant $\mathcal{I}$, i.e., $merge$ preserves the invariant if $\mathcal{M}$ is satisfied:

$$\mathcal{I}(s_1) \wedge \mathcal{I}(s_2) \wedge \mathcal{M}(s_1, s_2) \Rightarrow \mathcal{I}(merge_{state}(s_1, s_2)) \qquad \text{(i2)}$$

Further, we define global mergeability $\hat{\mathcal{M}}$ that holds for a global state $\hat{s}$ if all local states in $\hat{s}$ are mergeable, i.e., $\hat{\mathcal{M}}(\hat{s}) = \forall i, j \in \mathcal{P}.\ \mathcal{M}(s_i, s_j)$. As merging can happen at any point in time, we have to ensure that mergeability always holds, thus making mergeability another invariant of the system.

Definition 2 (Mergeability)**.** *The TS* $(\hat{S}, \rightarrow)$ *is called* mergeable w.r.t. $\mathcal{M}$ *iff for all global states* $\hat{s}_1$ *and* $\hat{s}_2$ *holds* $\hat{\mathcal{M}}(\hat{s}_1) \wedge (\hat{s}_1 \rightarrow \hat{s}_2) \Rightarrow \hat{\mathcal{M}}(\hat{s}_2)$.

To prove mergeability, we introduce the following two properties for mergeable data types. First, we define *mergeability of initial states*, i.e., the intial state of a mergeable data type has to be mergeable:

$$\frac{\tau_i = \mathsf{m}(e); \tau_i' \qquad \mathsf{m} \in Weak \qquad \mathsf{m}_{pre}(s_i, e)}{\hat{p} \rightarrow \hat{p}[p_i \mapsto (\mathsf{m}_{state}(s_i, e), \tau_i')]} \ \textsc{Tr-Weak}$$

$$\frac{\begin{array}{c} p_i = (s_i, \mathsf{m}(e); \tau_i) \qquad \mathsf{m} \in Strong \\ \hat{\mathcal{M}}(s_1, \ldots, s_n) \qquad \mathsf{m}_{pre}(merge_{i \in \mathcal{P}}(s_i), e) \\ s = \mathsf{m}_{state}(merge_{i \in \mathcal{P}}(s_i), e) \end{array}}{\hat{p} \rightarrow ((s, \tau_1), \ldots, (s, \tau_n))} \ \textsc{Tr-Strong}$$

$$\frac{\tau_i = merge_j; \tau_i' \qquad \mathcal{M}(s_i, s_j)}{\hat{p} \rightarrow \hat{p}[p_i \mapsto (s_i, merge(s_j); \tau_i')]} \ \textsc{Tr-Merge}$$

Fig. 6: The transition relation

$$init(s_1) \wedge init(s_2) \Rightarrow \mathcal{M}(s_1, s_2) \qquad \text{(m0)}$$

Second, we define *meargeability of merge*, i.e., if two mergeable states are merged then the resulting state is mergeable:

$$\mathcal{M}(s_1, s_2) \Rightarrow \mathcal{M}(merge_{state}(s_1, s_2), s_2) \qquad \text{(m1)}$$

c) Safe System: A safe system is one that enjoys invariant sufficiency and mergeability as defined above.

2) System Model: The model assumes a fixed number n of processes each uniquely identified by an id $i \in \{1, \ldots, n\} = \mathcal{P}$. Every process i stores the local state $s_i \in State$ of a mergeable data type and a trace τ_i, defined in Definition 3, of the program the process wishes to execute. We use $p_i = (s_i, \tau_i)$ to refer to the state and the trace of process i. We use $\hat{p} = (p_1, \ldots, p_n)$ to refer to all the processes and use $\hat{p}[p_i \rightarrow p_i']$ to modify the state of p_i to p_i' and leave all other processes unchanged.

Definition 3 (Traces and actions)**.** *Traces* τ *are either empty, denoted with* $\cdot$*, or an action* α *followed by another trace* τ'*, denoted with* $\alpha; \tau'$*. An action* α *is either a method call of the form* $\mathsf{m}(e)$ *or* $merge_j$ *that instructs the process to start merging its state with the state of process* j*.*

The model is parametrized by two disjoint sets of methods, $Weak$ and $Strong$, with the assumption that $merge \in Weak$. We denote by $\hat{S} = (S \times \tau)^n$ the global state of all processes, i.e. their local state and the trace of their program.

Figure 6 defines a *transition system* $(\hat{S}, \rightarrow)$ for global states [38] of a mergeable data type and their traces which models the execution of a ConLoc program. Tr-Weak models Weak execution where a method is executed locally by effecting only the local state. Tr-Strong models strong execution where the local states of all[1] processes are merged before the method is executed. The notable difference between these two rules is that Tr-Weak depends on and modifies the local state of a single process whereas Tr-Strong depends on and modifies the state of all processes. As such, methods are always executed atomically on a single replica and – in the case of Strong consistency – atomically in the entire distributed system across all replicas w.r.t. the accessed objects.

[1]The rule can be easily adapted to require only a majority to step into s to support a majority quorum, e.g., allowing for failing nodes.

TR-MERGE defines the semantics of $merge_j$. For mergeable processes i and j, the $merge_j$ action is replaced by a call to the $merge$ method with a copy of the state s_j. This rule allows process j to modify its state concurrently with the merge.

Example. With $n = 2$ processes, the global state of `ReplicatedCounter` is $(s_1, s_2) = (0, 0)$ i.e., the counter is 0 on both replicas, with the traces $\tau_1 = inc(); merge_2; \cdot$ and $\tau_2 = inc(); \cdot$ with $inc, merge \in Weak$.

The following diagram exemplifies the execution:

$$\begin{aligned} &((0, inc(); merge_2; \cdot), (0, inc(); \cdot)) \\ &\quad \to \quad ((1, merge_2; \cdot), (0; inc(); \cdot)) \quad \to \quad ((1, merge_2; \cdot), (1, \cdot)) \\ &\quad \to \quad ((1, merge(1); \cdot), (1, \cdot)) \qquad \to \quad ((2, \cdot), (1, \cdot)) \end{aligned}$$

3) Safety of CONLOC*:* As CONLOC's TS is parametrized by the $Weak$ and $Strong$ sets, we now present the properties that Weak and Strong methods must satisfy.

a) Consistency Specification: The separating criterion between Weak and Strong methods is whether mergeability is retained after a local execution or not. Weak methods retain it while Strong methods require coordination.

We define *mergeability for Weak methods*:

$$\mathcal{M}(s_1, s_2) \wedge \mathsf{m}_{pre}(s_1, e) \Rightarrow \mathcal{M}(\mathsf{m}_{state}(s_1, e), s_2) \qquad \text{(m2)}$$

Classifying Strong methods solely as those that do not satisfy m2 is insufficient, as this could result in a non-mergeable state after executing the designated Strong method m. Since a non-mergeable state can never be merged with any other state (as m1 requires mergeability for merging), diverged states in the distributed program cannot be reconciled.

Thus we define *mergeability for Strong methods* as follows:

$$\begin{aligned} &\mathcal{M}(s_1, s_2) \wedge \mathsf{m}_{pre}(merge(s_1, s_2), e) \\ &\quad \Rightarrow \mathcal{M}(\mathsf{m}_{state}(merge(s_1, s_2), e), s_2) \end{aligned} \qquad \text{(m3)}$$

The property states that, if a state is mergeable with another, it must remain mergeable with the other state after merging and applying the method to the resulting state. Crucially, this precondition is checked not only on a local state but also on the merge of any two states, necessitating coordination.

The m3 property ensures that merging after executing a Strong method remains possible. The two properties m2 and m3 determine whether a method can be safely executed Weakly or Strongly. The following lemma states the expected result that all Weak methods can also be safely executed in Strongly:

Lemma 1. *If a method* m *satisfies m2 then* m *satisfies m3.*

Proof. Assume m2 holds for m for all states s_1 and s_2. Let $s' = merge(s_1, s_2)$, then from $\mathcal{M}(s_1, s_2)$ and m1, we know $\mathcal{M}(merge(s_1, s_2), s_2)$. Thus, from m2, we conclude $\mathcal{M}(\mathsf{m}_{state}(merge(s_1, s_2)), s_2)$. □

b) Safety: We now prove that CONLOC's transition system is safe when the methods of the $Weak$ and $Strong$ sets obey m2 and m3, respectively. First, we instantiate the states of the safety framework with the domain of the TS, i.e, with pairs of process state and trace. Second, we introduce valid data types which are those that satisfy the properties of the safety framework and the condition stated in the previous sentence.

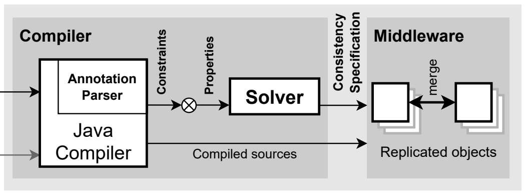


Fig. 7: The architecture of CONLOC.

Definition 4 (Valid data type)**.** *Let the set of methods of a mergeable data type be divided into two disjoint sets Weak and Strong. A mergeable data type is called* valid *if i0, i1, i2, m0, and m1 hold, and m2 hold for all methods in Weak, and m3 holds for all methods in Strong.*

The validity of data types is required for safe transition systems that satisfy invariant sufficiency and mergeability.

Theorem 1 (Safety)**.** *The TS for a valid mergeable data type is invariant-sufficient w.r.t. $\mathcal{I}$ and mergeable w.r.t. $\mathcal{M}$.*

Proof. By case analysis on $\hat{p}_1 \to \hat{p}_2$. We write $\hat{s}$ for the projection of $\hat{p}$ to the process states. Case TR-WEAK. The premise gives (a) $\mathsf{m}_{pre}(s_i, e)$. $\hat{\mathcal{I}}(\hat{s}_2)$ follows from (a), $\mathcal{I}(\hat{s}_1)$, and i1. $\hat{\mathcal{M}}(\hat{s}_2)$ follows from (a), $\hat{\mathcal{M}}(\hat{s}_1)$, and m2. Case TR-STRONG. The premise gives (b) $\mathsf{m}_{pre}(merge_{p\in\mathcal{P}}(s_i))$. (c) $\mathcal{I}(merge_{i\in\mathcal{P}}(s_i))$ follows from $\mathcal{I}(\hat{s}_1)$, $\hat{\mathcal{M}}(\hat{s}_1)$, and i2. $\hat{\mathcal{I}}(\hat{s}_2)$ follows $\mathcal{I}(\mathsf{m}(merge_{i\in\mathcal{P}}(s_i)))$ which follows from (c), (b), and i1. $\hat{\mathcal{M}}(\hat{s}_2)$ follows from (b), $\hat{\mathcal{M}}(\hat{s}_1)$, and m3. Case TR-MERGE. The process states do not change. □

We formally define program executions, with *safe* executions defined as those where the invariant holds at every step.

Definition 5 (Execution)**.** *Given a global state $\hat{p}_0$ an execution from $\hat{p}_0$ is a sequence of global states $\hat{p}_0, \hat{p}_1, \ldots$ such that $\hat{p}_k \to \hat{p}_{k+1}$.*

Definition 6. *An execution from $\hat{p}_0$ is* safe *if for all states $\hat{p}_k$ in the execution $\hat{\mathcal{I}}(\hat{p}_k)$ and $\hat{\mathcal{M}}(\hat{p}_k)$ hold.*

The last step is to show that transition systems for valid mergeable data types induce safe executions.

Theorem 2. *For every $\hat{p}_0$ such that $init(\hat{p}_0)$ and a TS for a valid mergeable data type, all executions from $\hat{p}_0$ are safe.*

Proof. Follows from i0, m0 and Theorem 1. □

4) Verification: The CONLOC middleware ensures safe executions given that a mergeable data type is valid (Definition 4). To check safety, we employ a solver that first checks that properties i0 to i2 and m0 and m1 are valid. Then, for every method, we check if it satisfies m2 and, in that case, label the method as Weak. Otherwise, the method is checked for m3 and labeled Strong if the property holds. If a method is not Strong (an thus not Weak), then the program is not valid. The solver generates the consistency specification which maps each method to Weak or Strong consistency, depending whether m2 is satisfied (Weak) or not (Strong).

IV. Technical Realization

In this section, we discuss the implementation of ConLoc.[2] The architecture (Figure 7) consists of the ConLoc compiler, which implements the verification procedure, and the ConLoc middleware which operates at run time. The compiler consists of $\sim 4\,K$ LoC and the middleware $\sim 2.5\,K$ LoC.

1) ConLoc Compiler: The compiler is an extension to the Eclipse Java compiler [39] that takes Java sources annotated with JML [33]. The compiler uses an *annotation parser* to obtain the invariants and Z3 [40] to verify them. It produces a *consistency specification* for valid data types (cf. Section III-C) when all invariants are verified, otherwise it rejects the program.

2) ConLoc Middleware: The middleware is given the compiled sources and the consistency specification. It distributes objects according to said specification. The middleware is implemented on top of Akka actors [41], which provide asynchronous message-passing with at-most-once-delivery guarantees. We use the widely deployed Zookeeper Atomic Broadcast protocol (ZAB) [42] for coordination required by Strong consistency. Zookeeper can be configured to use quorum reads and writes to allow for some nodes to go down even with Strong consistency. This introduces some flexibility when desired.

V. Evaluation

In this section, we present the evaluation of ConLoc. We aim to answer the following research questions.

(RQ1 Expressivity and applicability): *Does* ConLoc *support safe implementations of real-world applications?* This research question ensures that it is possible to represent the desired logic of distributed applications with ConLoc and that ConLoc provides correct synchronization.

(RQ2 Support for existing CRDTs): *Is it possible to ensure safety for existing mergeable data types with* ConLoc*?* Here, we determine whether ConLoc can be easily adopted by implementors of replicated data types libraries, for verification of data types that can be composed in replicated applications.

(RQ3 Efficiency): *Is the compiler and verification procedure fast enough to ensure that our approach is usable in practice?* With this research question we want to ensure that ConLoc can be used on real-world, replicated applications without incurring a performance penalty that hinders its adoption.

(RQ4 Application speedup): *Does* ConLoc *help speeding up of distributed, replicated applications?* This research question assesses whether ConLoc's analysis has a concrete performance impact on distributed systems.

A. Case Studies: ConLoc *in Practice*

We reimplemented in ConLoc various case studies of local-first applications that are composed of CRDTs from literature. Since such applications are built upon standard CRDTs, we first implemented and verified a small annotated CRDT library. An overview of all case studies is in Table I. For each, the table lists the lines of code (*LOC*), the number of methods (*#Methods*), for total (*all*) and *Strong* methods. *#Fields* shows the total number of fields (*all*) and the number of fields with a CRDT, collection or array type (*coll*). *Complexity* is the number of inner nodes of the AST of the formula as produced by the rewriting simplification of Z3 [43] for invariants (*inv*), pre- (*pre*) and postconditions (*post*).

TABLE I: Metrics for the case studies.

	Case study	**LOC**		**#Methods**		**#Fields**		**Complexity**		
		code	*JML*	*all*	*Strong*	*all*	*coll*	*inv*	*pre*	*post*
CRDTs	GCounter	38	11	5	0	1	1	0	1	68
	PNCounter	63	19	8	0	2	2	0	2	133
	Bounded Counter	71	33	7	0	3	2	10	2	215
	Multi-Value Register	31	6	3	0	1	1	5	0	2
	GSet	31	12	5	0	1	1	0	0	40
	2P-Set	36	19	6	0	2	0	0	3	171
	GGraph	37	12	4	0	2	2	14	0	11
	2P-Graph	48	17	6	0	2	2	14	0	23
Applications	Account	33	13	4	1	1	0	2	7	146
	Account No-CRDT	55	29	6	1	2	2	2	5	85
	Account LWW	39	16	4	0	2	0	2	6	20
	Joint Account	40	24	6	0	3	0	4	15	184
	Resettable Counter	32	11	4	0	1	1	15	0	69
	Consensus	46	15	4	0	3	1	20	14	73
	Distributed Lock	38	20	2	0	2	1	37	8	31
	Message Groups	75	26	6	1	2	2	31	11	185
	Tournament	243	117	38	0	13	9	39	27	211
Riak CRDTs	GCounter	71	21	10	0	2	1	0	1	58
	PNCounter	64	27	10	0	3	2	0	3	129
	GSet	65	65	14	0	1	1	0	0	121
	2P-Set	110	27	20	0	2	2	0	23	495
	ORSet	119	55	20	0	2	2	0	0	700

1) CRDT Library: The library comprises eight general CRDTs from literature [44]. It consists of **GCounter**, a grow-only counter, **PNCounter**, a counter that can be incremented and decremented, **Bounded Counter**, a counter that can be incremented up to a maximum bound, **Multi-Value Register**, a register that can hold multiple values, **GSet**, a grow-only set, **2P-Set**, a set that supports adding and removing elements such that once an element was removed it cannot be added again, **GGraph**, a grow-only graph that nodes and edges can be added to, and **2P-Graph**, a graph that supports adding and removing nodes and edges such that once removed they cannot be added again. All CRDTs in the library are checked for program safety (cf. Section III), and are ready to use.

2) Applications: We implemented nine applications leveraging the CRDT library, most of which are from scientific literature. These applications cover a variety of different scenarios. **Account** is the application from which the example in Section II is derived. The account uses the `PNCounter` from the CRDT library. The invariant states that the counter cannot be negative. **Account No-CRDT** is a reimplementation of `CreditAccount` without the CRDT library. **Account LWW** uses a last-writer-wins (LWW) register where increments and decrements are applied to the value of the register. **Joint Account** only allows withdraw operations that have to be granted by another party before it is performed [45] and does allow the account to be negative. **Resettable Counter** [46] implements reset by using rounds, i.e., increments are always applied to the current round, whereas `reset` sets the counter to zero and advances to the next round. The eventually consistent **Consensus** protocol [37] is correct even with Weak consistency. Replicas agree to a value by setting a flag in a boolean array.

[2]Source code available at https://github.com/consysT-project/consysT-code/.

The invariant ensures that the agreement occurs only when all replicas set their respective flag. In the **Distributed Lock** [37], ownership of the lock is passed around replicas, and the invariant ensures that only one replica can own the lock at any time. The **Message Groups** application [47] implements a message group with an upper bound on the number of users that must not be violated. Users can be added to the message group and messages can be broadcasted to all users. The addition of users is a strong operation as the upper bound can be violated after merging replicas of groups while broadcasting is weak. The **Tournament** management system [48] handles matches and players in a game. The invariants ensure matches do not exceed player capacity, players maintain a positive budget, active matches have players, and only players can enroll in matches. The operations include adding and removing players and matches, managing player enrollment, and adding funds. The application employs multiple 2P-Sets. Referential integrity constraints are satisfied as operations are transactional which ensures that certain objects are created sequentially.

These case studies answer RQ1, demonstrating that ConLoc is effective in supporting the implementation and verification of existing applications composed of multiple CRDTs.

B. Safety of the Riak CDRT Library in ConLoc

We verified the popular CRDT library [49] originally developed for the database Riak [17] by taking the library's five data types and reimplementing them with ConLoc. The CRDTs in the library are general-purpose, similar to our CRDT library (Section V-A), but employ different data structures and are designed for Riak. Thus the implementations differ from our standalone library. The Riak library contains implementations of **GCounter**, **PNCounter**, **GSet**, **2P-Set**, and Observed-Remove Sets (**ORSet**), the latter allowing addition and removal of an element more than once. We annotated the library methods and ensured program safety by verifying the properties i0–i2 and m0–m3 (Section III). Compared to our CRDT library, which uses rudimentary underlying collections like arrays and sets, the Riak library adopts more sophisticated collection types, such as maps and multimaps, which increase the complexity of annotations. Further, The Riak library contains more methods such as `addAll` or `containsAll`. For compatibility with ConLoc, we applied some minor code changes (unrelated to the library semantics) to the CRDTs, e.g., changing the return type of merge to `java.lang.Void`, and setting the number of replicas to a fixed value. The latter enables a well-defined domain of internal maps, to work around Z3's lack of partial maps [40]. We believe these changes are minimal and do not hinder expressivity.

These results positively answer RQ2 indicating that a third-party library, not designed for ConLoc, can be verified. Annotating took ~ 4 working days, once ConLoc was fully functional, which we consider a reasonably modest effort.

C. Efficiency of the Compilation Process

We executed ConLoc on the 22 case studies and CRDT implementations augmented with ConLoc annotations and measured the time taken by the compiler. Figure 8 presents the results for each data type averaged over 10 measure runs

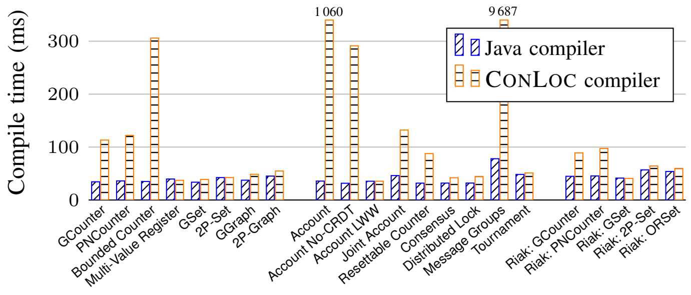
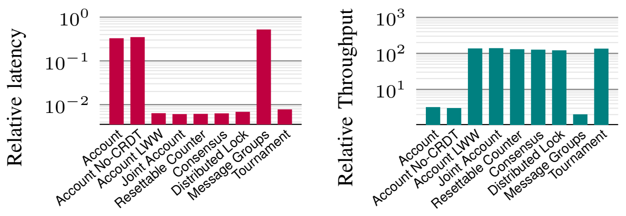


Fig. 8: Compiler Benchmarks.



(a) Latency, smaller is better. (b) Throughput, higher is better.

Fig. 9: Performance Benchmarks.

preceded by 10 warmup runs. For comparison, we also report the compile time of the Java compiler executed on the same class. The compiler takes nine seconds for Message Groups, one second for Account and substantially less for the other cases. Highest values of compilation time are due to two factors: 1) The more Strong methods the more time is required for verification; for Strong methods the verifier must check that mergeability does not hold for all states, thus Z3 explores a large state space. 2) The complexity of object states also contributes, as Z3 demands more time for verification of classes with more fields. The existence of fields with a collection type factor into the verification time due to their complex constraints.

This result answers RQ3, demonstrating that the compiler implemented by ConLoc is sufficiently fast to be executed on common CRDT implementations.

D. Speeding up Replicated Applications

To evaluate the effect of ConLoc, we run the applications from Section V-A as performance benchmark. The benchmarks are executed in the AWS cloud on 4 processes each running on a `t2.large` machine (2 vCPUs and 8 GiB memory) in the same cluster. Every process executes 1 K random method calls, uniformly distributed, on all replicated data types, with randomly generated method arguments. We measured latency as the time required to execute a single method, while throughput is the number of operations a process executes per second.

We consider two configurations. The first, All-Strong, contains only Strong operations. This configuration corresponds to a replicated application where developers cannot come up with a better design and hence default to serializing operations to satisfy invariants, representing a correct but conservative baseline. The second configuration, Mixed, adopts our approach of using both Strong and Weak operations as inferred by ConLoc. Both configurations satisfy Invariant Sufficiency, meaning the invariant is never violated. In a concurrent execution, All-Strong satisfies the invariant because Strong consistency restricts concurrency. For MIXED, while Weak operations may occur, the invariant remains satisfied, as the application is verified with CONLOC.

The results in Figure 9a show the ratio between the latency of MIXED and of ALL-STRONG; similarly for throughput in Figure 9b. For instance, the latency of Account in MIXED is a third of that of ALL-STRONG, and the throughput is $\times 2.94$ compared to ALL-STRONG. Overall, the results show that MIXED improves the latency and throughput for *all* case studies compared to ALL-STRONG. The smallest improvement in latency and throughput is for Account, Account No-CRDT and Message Groups due to their Strong operations. The best performance improvements are in the case studies with only Weak operations in MIXED, resulting in a latency that is only approx. $\sim 0.6\,\%$ compared to ALL-STRONG and a throughput that is approx. ~ 125 times higher. The reason for such an improvement is that all Weak operations can be executed locally without contacting other replicas thus completely removing the overhead of synchronization of Strong operations.

These results answer RQ4 by demonstrating that CONLOC improves the efficiency of replicated applications by permitting Weak consistency wherever feasible, while still correctly enforcing application invariants.

VI. Related Work

A. *Conflict-Free Replicated Data Types*

Local-first software [3] allows processes to own their data locally and synchronize on demand. Distributing data with CRDTs [7] fulfills these principles as data synchronization occurs eventually. CRDTs come in two flavors, operation-based and state-based, which can be defined in terms of the other. In CONLOC, we utilize state-based CRDTs for their robustness against network message duplication and their inherent causality [50]. Shapiro et al. [44] provide a comprehensive overview of several CRDTs, such as counters, sets, maps, etc. Delta-based CRDTs [51] improve the efficiency of state-based CRDTs by not forwarding the entire state between replicas but only incremental changes (deltas).

Nair et al. [37] propose a technique to verify replicated objects. Developers define a replicated data type – state and operations – and application-specific invariants. The proof system checks whether this object can be replicated with Weak consistency without violating any invariants. CONLOC's invariant sufficiency and mergeability are based on this technique. Yet, adding the necessary synchronization is not supported, while CONLOC does this automatically. Also, CONLOC can verify replicated data types build upon other replicated data types – e.g., the `CreditAccount` in Section II uses a Counter CRDT – using **`stateful`** expressions that reference mutating operations of the underlying data type. Further, CONLOC ensures safety not only for Weak data types but also *generates* consistency specifications for data types with both Strong and Weak methods, ensuring efficient and correct execution.

VeriFx [25] is a language for CRDT verification. This approach is similar to ours in that developers program data types and the system automatically validates certain properties. Yet, VeriFx verifies inherent properties of the data types, while CONLOC determines which methods of a class need execution with Strong consistency to align with the *user-imposed* constraints. Also, CONLOC supports mutable data structures, while VeriFx is restricted to immutable, functional data structures. Yet, in VeriFx, one implementation forms the basis of both executable code and formal verification, whereas in CONLOC developers provide a separate specification.

Nieto et al. [28] offer libraries that enable the implementation and verification of operation-based CRDTs using separation logic, enabling programmers to write Coq [52] specifications and prove the correctness of the implementations.

B. *CRDTs with Strong Consistency*

Several works aim to augment CRDTs with Strong consistent operations. Conflict-aware Replicated Data Types (CARDs) [45] extend the concept of CRDTs to accommodate conflicting operations. CARDs use consistency guards, i.e., invariants on local and global state, to identify conflicting operations. Carol [53] is a language for mixed consistency built upon CARDs. Developers provide invariants on the replicated store and the system derives the consistency for each operation. Carol features a refinement type system by lifting consistency guards of CARDs to the type level. Instead of relating a global and local state as with CARDs, in CONLOC, developers define constraints directly on the application state. Observable atomic consistency (OAC) [46] integrates operation-based CRDTs with Strong operations. Consistency types for replicated data (CTRD) [54] utilizes OAC to reason about Strong and Weak consistent data. Consistency is defined by assigning a consistency level to types.

C. *Invariants for Consistency*

Numerous works use invariants to determine the consistency of operations. In RedBlue consistency [55], operations can be either Blue (i.e., Weak) or Red (i.e., Strong), and developers manually define which operations are Blue or Red. SIEVE [56] employs static analysis to distinguish between Red and Blue operations – finding commutative operation pairs (hence Weak) – and additionally uses dynamic enforcement. Explicit Consistency [57] infers consistency of operations by applying invariants. Indigo [48] is a middleware to find conflicting operations by stating class invariants. Gotsman et al. [58] propose a proof rule for identifying operations that satisfy invariants when executed weakly, which is used by CISE [59], a static analysis tool that identifies the weakest consistency for satisfying invariants. Compared to RedBlue consistency, CONLOC uses state-based replication, which is crucial for real-world applications to take advantage of robustness and causality.

Invariant confluence ($\mathcal{I}$-confluence) [60] is a criterion specifying when applications need immediate coordination (i.e., Strong consistency). A set of operations satisfies $\mathcal{I}$-confluence if all states reachable during the execution of these operations satisfy the invariant. CONLOC's invariant sufficiency implies $\mathcal{I}$-confluence, as every state satisfies the invariant – the initial state and every state that is the result of an operation. IPA [61] applies $\mathcal{I}$-confluence to check that concurrent operations are invariant preserving. IPA marks operations that violate invariants and proposes code changes to implement synchronization. Similar to IPA, ConLoc finds operations that do not preserve the invariant when executed concurrently. Yet, while IPA does not consider the underlying replicated data types, ConLoc ensures safety of the application *including* the underlying data type. Thus, the safety encompasses the whole application, and does not, as in IPA, require to trust that the developer uses the correct data types. This solution enables the middleware to directly generate replication logic based on the consistency specification, instead of merely proposing changes.

In Quelea [30], invariants are defined on histories of operations. The system infers the required consistency for operations. Q9 [62] is a language and verification procedure for replicated data types. Developers define data types and invariants. If Q9 detects that the data type cannot be replicated with Weak consistency, an automatic repair mechanism selects a suitable stronger consistency model not violating any invariant. Hamsaz [29] is a reasoning system to generate coordination protocols for (operation-based) replicated objects based on class invariants. Current research of inferring consistency from application invariants mostly targets operation-based replication strategies, reducing verification effort to, e.g., commutativity of operations. However unlike ConLoc, these approaches do not have the practical advantages of state-based replication.

State-based and operation-based replication require different treatments. While preserving invariants in an operation-based system requires reasoning about the commutativity of operations, preserving invariants in a state-based system requires reasoning about state merges. Merges are by definition commutative, but can still violate invariants [37]. Thus, ConLoc provides a framework that is fitted for state-based CRDTs.

LASP [23] enables the safe combination of existing CRDTs by utilizing an ad-hoc dataflow language to specify dependencies between them. In LoRe [24], like LASP, uses a dataflow language for CRDT combination, but also ensures the satisfaction of user-defined invariants. Invariant validation relies on the analysis of the program's dataflow – thanks to the explicitly defined dependencies between inputs, which are protected by constraints, and the outputs. Similar to ConLoc, LoRe can employ additional synchronization mechanisms when Weak consistency cannot guarantee invariants. The functional dataflow approach simplifies the verification while ConLoc uses a mainstream imperative language and allows developers to define correctness requirements using invariants.

Ma et al. [63] proposel Noctua, a framework for consistency analysis in web applications informs the developer where synchronization is needed to ensure consistency. Similar to ConLoc, Noctua identifies pairs of operations that could lead to state divergence or correctness violations if executed concurrently. In contrast to ConLoc, Noctua does not require user-defined invariants but infers them via symbolic execution in a custom interpreter. As a result, Noctua does not support language abstractions like unrestricted loops or recursion.

D. Mixed Consistency

Another line of work does not infer consistency levels based on application invariants and program verification but offers different levels to the developers via dedicated abstractions.

CAPtain.js [64] is a library with *consistent* and *available* data types that ensure strong consistency by sacrificing availability, and availability but with eventual consistency. Conflicts among consistency levels generate runtime exceptions. In a similar vein, CScript [65] is a language for replicated objects that are available or consistent and allows one to mix such objects within the same service in the distributed system.

Type systems have been employed to prevent errors in mixing different consistency levels. The IPA system [66] prevents that values with high consistency flow into values with low consistency using subtyping. MixT [47] is a C++ DSL for transactions over multiple data stores with different consistency levels and an information-flow type system that avoids mixing consistency levels. Similarly, ConSysT [67] is a Java extension that combines consistency types and object structure.

VII. Limitations and Future Work

In ConLoc, as in other approaches [25], [29], [61], annotations are limited to first-order logic. While not complete, solvers for first-order formulas such as Z3 provide a high level of verification automation but they require additional manual aid when verifying properties of recursive data types.

In this work we made two simplifying assumptions on the execution of Java programs: (1) numeric values do not overflow; (2) programs do not run out of memory. The assumption on numeric values can be addressed by using `BigInteger` where an `int` is used, or by adding overflow assertions to the underlying solver. The goal of ConLoc is to provide a verification of replicated data types and not for Java specific code.

Another limitation is that pre- and post-constraints are not checked against the source code. The use of reasoning frameworks for Java and JML annotations, such as KeY [68] (not fully automatic) or OpenJML [69] (requires additional runtime checks), can be employed alongside ConLoc to improve the confidence that the JML annotations reflect the Java code.

VIII. Conclusion

Local-first software adopts data replication for scalability and availability of distributed applications. As designing correct replicated data types is challenging, developers often use off-the-shelf solutions like CRDTs. Yet, application invariants do not always transfer to the replicated case, forcing developers to reason about the interplay between concurrency and replication – hence defeating the purpose of off-the-shelf solutions.

We introduced ConLoc, a system that processes invariants in code annotations of mergeable data types and generates a safe distributed application with correct synchronization logic. We show that ConLoc can be applied to real-world applications, that the automatic verification of invariants demands low time overhead, and that applications coordinated based on the verification results exhibit better latency and throughput compared to a correct but conservative sequential baseline.

Acknowledgments

This work is supported by the Swiss National Science Foundation (SNSF) under grant 200429, and the LOEWE initiative (Hesse, Germany) within the emergenCITY centre.

References

[1] L. Lamport, “Paxos made simple,” *ACM SIGACT News 32, 4*, 2001.

[2] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Pearson Prentice Hall, 2007.

[3] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, “Local-first software: You own your data, in spite of the cloud,” in *Onward!*, 2019.

[4] M. Kleppmann, *Designing Data-Intensive Applications*. O’Reilly, 2017.

[5] G. Kaki, S. Priya, K. Sivaramakrishnan, and S. Jagannathan, “Mergeable replicated data types,” *PACMPL*, vol. 3, no. OOPSLA, 2019.

[6] K. D. Porre, F. Myter, C. D. Troyer, C. Scholliers, W. Meuter, and E. G. Boix, “A generic replicated data type for strong eventual consistency,” in *PaPoC*, 2019.

[7] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *SSS*, 2011.

[8] P. S. Almeida, A. Shoker, and C. Baquero, “Delta state replicated data types,” *JPDC*, vol. 111, 2018.

[9] N. M. Preguiça, “Conflict-free replicated data types: An overview,” *arXiv*, vol. abs/1806.10254, 2018.

[10] Yjs Contributors, “Yjs: A CRDT framework with a powerful abstraction of shared data,” 2023, https://github.com/yjs/yjs.

[11] M. Kleppmann, D. P. Mulligan, V. B. F. Gomes, and A. R. Beresford, “A highly-available move operation for replicated trees,” *TPDS*, vol. 33, 2022.

[12] Automerge Contributors, “Automerge: A JSON-like data structure (a CRDT) for building collaborative applications,” 2023, https://github.com/automerge/automerge.

[13] W. Yu, L. André, and C.-L. Ignat, “A CRDT supporting selective undo for collaborative text editing,” in *DAIS*, 2015.

[14] J. Johnson, “How facebook scales big data systems,” *QCon*, 2014, https://www.infoq.com/presentations/scale-facebook-big-data.

[15] D. Martyanov, “CRDTs in production,” *QCon*, 2018, https://www.infoq.com/presentations/crdt-production/.

[16] D. Ivanov, “Practical demystification of CRDTs,” *Lambda Days*, 2016, https://www.lambdadays.org/lambdadays2016/dmitry-ivanov.

[17] Riak. (2021) Riak – enterprise NoSQL database. https://riak.com/.

[18] Antidote DB. (2021) Antidote DB. https://www.antidotedb.eu/.

[19] Amazon. (2021) DynamoDB. https://aws.amazon.com/dynamodb/.

[20] W. Vogels, “Eventually consistent,” *CACM*, vol. 52, no. 1, 2009.

[21] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley, 1986.

[22] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Highly available transactions: Virtues and limitations,” *Proceedings of the VLDB Endowment*, vol. 7, no. 3, 2013.

[23] C. Meiklejohn and P. Van Roy, “Lasp: A language for distributed, coordination-free programming,” in *PPDP*, 2015.

[24] J. Haas, R. Mogk, E. Yanakieva, A. Bieniusa, and M. Mezini, “LoRe: A programming model for verifiably safe local-first software,” *TOPLAS*, vol. 46, no. 1, 2023.

[25] K. De Porre, C. Ferreira, and E. Gonzalez Boix, “VeriFx: Correct replicated data types for the masses,” in *ECOOP*, vol. 263, 2023.

[26] G. Zakhour, P. Weisenburger, and G. Salvaneschi, “Type-checking CRDT convergence,” *PACMPL*, vol. 7, no. PLDI, 2023.

[27] ——, “Automated verification of fundamental algebraic laws,” *PACMPL*, vol. 8, no. PLDI.

[28] A. Nieto, L. Gondelman, A. Reynaud, A. Timany, and L. Birkedal, “Modular verification of op-based CRDTs in separation logic,” vol. 6, no. OOPSLA2, 2022.

[29] F. Houshmand and M. Lesani, “Hamsaz: Replication coordination analysis and synthesis,” *PACMPL*, vol. 3, no. POPL, 2019.

[30] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan, “Declarative programming over eventually consistent data stores,” in *PLDI*, 2015.

[31] TIOBE. (2023) TIOBE index for april 2023. https://www.tiobe.com/tiobe-index/.

[32] V. Balegas, “Bounded counters: Maintaining numeric invariants with high availability,” 2016, https://pages.lip6.fr/syncfree/attachments/article/59/boundedCounter-white-paper.pdf.

[33] Gary Leavens. (2017) JML. https://www.cs.ucf.edu/~leavens/JML.

[34] E. W. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs,” *Commun. ACM*, vol. 18, no. 8, 1975.

[35] C. Flanagan and J. B. Saxe, “Avoiding exponential explosion: Generating compact verification conditions,” in *POPL*, 2001.

[36] M. Barnett and K. R. M. Leino, “Weakest-precondition of unstructured programs,” in *PASTE*, 2005.

[37] S. S. Nair, G. Petri, and M. Shapiro, “Proving the safety of highly-available distributed objects,” in *ESOP*, 2020.

[38] R. M. Keller, “Formal verification of parallel programs,” *Commun. ACM*, vol. 19, no. 7, 1976.

[39] Eclipse. (2021) Eclipse JDT Core. https://www.eclipse.org/jdt/core.

[40] Microsoft. (2020) Z3 Prover. https://github.com/Z3Prover/z3.

[41] Akka. (2022) Akka Documentation. https://doc.akka.io/docs/akka.

[42] F. P. Junqueira, B. C. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” in *DSN*, 2011.

[43] Z3 Guide, “Formula Simplification,” 2023, https://microsoft.github.io/z3guide/programming/Example Programs/Formula Simplification.

[44] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “A comprehensive study of convergent and commutative replicated data types,” INRIA, Tech. Rep. 7506, 2011.

[45] N. V. Lewchenko, A. Radhakrishna, and P. Černý, “Conflict-aware replicated data types,” *arXiv*, vol. abs/1802.08733, 2018.

[46] X. Zhao and P. Haller, “Observable atomic consistency for CvRDTs,” in *AGERE*, 2018.

[47] M. Milano and A. C. Myers, “MixT: A language for mixing consistency in geodistributed transactions,” in *PLDI*, 2018.

[48] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, “Putting consistency back into eventual consistency,” in *EuroSys*, 2015.

[49] David Clements. (2013) Java CRDT Library. https://github.com/dclements/riak-java-crdt.

[50] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, “Replicated data types: Specification, verification, optimality,” in *POPL*, 2014.

[51] V. Enes, P. S. Almeida, C. Baquero, and J. Leitão, “Efficient synchronization of state-based CRDTs,” in *ICDE*, 2019.

[52] The Coq Development Team, “The Coq reference manual – release 8.19.0,” https://coq.inria.fr/doc/V8.19.0/refman, 2024.

[53] N. V. Lewchenko, A. Radhakrishna, A. Gaonkar, and P. Černý, “Sequential programming for replicated data stores,” *PACMPL*, vol. 3, no. ICFP, 2019.

[54] X. Zhao and P. Haller, “Consistency types for replicated data in a higher-order distributed programming language,” *Programming*, vol. 5, no. 2, 2021.

[55] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, “Making geo-replicated systems fast as possible, consistent when necessary,” in *OSDI*, 2012.

[56] C. Li, J. Leitão, A. Clement, N. M. Preguiça, R. Rodrigues, and V. Vafeiadis, “Automating the choice of consistency levels in replicated systems,” in *USENIX ATC*, 2014.

[57] V. Balegas, C. Li, M. Najafzadeh, D. Porto, A. Clement, S. Duarte, C. Ferreira, J. Gehrke, J. Leitão, N. M. Preguiça, R. Rodrigues, M. Shapiro, and V. Vafeiadis, “Geo-replication: Fast if possible, consistent if necessary,” *IEEE Data Eng. Bull.*, vol. 39, no. 1, 2016.

[58] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro, “’cause i’m strong enough: Reasoning about consistency choices in distributed systems,” in *POPL*, 2016.

[59] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro, “The CISE tool: Proving weakly-consistent applications correct,” in *PaPoC*, 2016.

[60] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Coordination avoidance in database systems,” *Proceedings of the VLDB Endowment*, vol. 8, no. 3, 2014.

[61] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, and N. Preguiça, “IPA: Invariant-preserving applications for weakly consistent replicated databases,” *Proceedings of the VLDB Endowment*, vol. 12, no. 4, 2018.

[62] G. Kaki, K. Earanky, K. Sivaramakrishnan, and S. Jagannathan, “Safe replication through bounded concurrency verification,” *PACMPL*, vol. 2, no. OOPSLA, 2018.

[63] K. Ma, C. Li, E. Zhu, R. Chen, F. Yan, and K. Chen, “Noctua: Towards automated and practical fine-grained consistency analysis,” in *EuroSys*, 2024.

[64] F. Myter, C. Scholliers, and W. De Meuter, “A CAPable distributed programming model,” in *Onward!*, 2018.

[65] K. D. Porre, F. Myter, C. Scholliers, and E. G. Boix, “CScript: A distributed programming language for building mixed-consistency applications,” *JPDC*, vol. 144, 2020.

[66] B. Holt, J. Bornholt, I. Zhang, D. Ports, M. Oskin, and L. Ceze, “Disciplined inconsistency with consistency types,” in *SoCC*, 2016.

[67] M. Köhler, N. Eskandani, P. Weisenburger, A. Margara, and G. Salvaneschi, “Rethinking safe consistency in distributed object-oriented programming,” *PACMPL*, vol. 4, no. OOPSLA, 2020.

[68] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification – The KeY Book – From Theory to Practice*. Springer.

[69] OpenJML.org, “OpenJML,” 2023, https://www.openjml.org/.