LoRe: A Programming Model for Verifiably Safe Local-First Software (Extended Abstract)

Julian Haas
Technische Universität Darmstadt, Germany

Ragnar Mogk
Technische Universität Darmstadt, Germany

Elena Yanakieva
University of Kaiserslautern-Landau, Germany

Annette Bieniusa
University of Kaiserslautern-Landau, Germany

Mira Mezini
Technische Universität Darmstadt, Germany

Abstract

Local-first software manages and processes private data locally while still enabling collaboration between multiple parties connected via partially unreliable networks. Such software typically involves interactions with users and the execution environment (the outside world). The unpredictability of such interactions paired with their decentralized nature make reasoning about the correctness of local-first software a challenging endeavor. Yet, existing solutions to develop local-first software do not provide support for automated safety guarantees and instead expect developers to reason about concurrent interactions in an environment with unreliable network conditions.

We propose *LoRe*, a programming model and compiler that automatically verifies developer-supplied safety properties for local-first applications. *LoRe* combines the declarative data flow of reactive programming with static analysis and verification techniques to precisely determine concurrent interactions that violate safety invariants and to selectively employ strong consistency through coordination where required. We propose a formalized proof principle and demonstrate how to automate the process in a prototype implementation that outputs verified executable code. Our evaluation shows that *LoRe* simplifies the development of safe local-first software when compared to state-of-the-art approaches and that verification times are acceptable.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Distributed programming languages; Software and its engineering → Data flow languages; Software and its engineering → Consistency; Theory of computation → Pre- and post-conditions; Theory of computation → Program specifications; Computer systems organization → Peer-to-peer architectures

Keywords and phrases Local-First Software, Reactive Programming, Invariants, Consistency, Automated Verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.12

Related Version *Extended Version*: `https://arxiv.org/abs/2304.07133` [13]

Supplementary Material *Software (Source Code)*: `https://github.com/stg-tud/LoRe`
Software (ECOOP 2023 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.9.2.11`

Funding This work was funded by the German Federal Ministry of Education and Research together with the Hessen State Ministry for Higher Education (ATHENE), the German Research Foundation (SFB 1053), and the German Federal Ministry for Economic Affairs and Climate Action project SafeFBDC (01MK21002K).

© Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 12; pp. 12:1–12:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Applications that enable multiple parties connected via partially unreliable networks to collaboratively process data prevail today. An illustrative example is a distributed calendar application with services to add or modify appointments, where a user may maintain multiple calendars on different devices, may share calendars with other users, back them up in a cloud; calendars must be accessible to users in a variety of scenarios, including offline periods, e.g., while traveling – yet, planning appointments may require coordination between multiple parties. The calendar application is representative for other collaborative data-driven software such as group collaboration tools, digital (cross-organizational) supply chains, multiplayer online gaming, and more.

The dominating software architecture for such applications is centralized: data is collected, managed, and processed centrally in data centers, while devices on the edge of the communication infrastructure serve primarily as interfaces to users and the outside world. This architecture simplifies the software running on edge devices since concerns like consistent data changes to ensure safety properties are managed centrally. However, this comes with issues including loss of control over data ownership and privacy, insufficient offline availability, poor latency, inefficient use of communication infrastructure, and waste of (powerful) computing resources on the edge.

To address these issues, local-first principles for software development have been formulated [17], calling for moving data management and processing to edge devices instead of confining the data to clouds. But for programming approaches that implement these principles to be viable alternatives to the centralized approach, they must support automatically verifiable safety guarantees to counter for the simplifying assumptions afforded by a centralized approach. Unfortunately, existing approaches to programming local-first applications such as *Yjs* [31] or *Automerge*[1] do not provide such guarantees. They use *conflict-free replicated data types (CRDTs)* [34] to store the parts of their state that is shared across devices and rely on callbacks for modeling and managing state that changes in both time and space. The unpredictability of the interactions triggered by the outside world, concurrently at different devices, paired with the absence of a central authority and the prevailing implicit dependencies in current callback-centred programming models, makes such reasoning without automated support a challenging, error-prone endeavour.

To close this gap, we propose a programming model for local-first applications that features explicit safety properties and automatically enforces them. The model has three core building blocks: *reactives*, *invariants*, and *interactions*. *Reactives* express values that change in time, but also in space by being replicated over multiple devices. *Invariants* are formula in first-order logic specifying safety properties that must hold at all times when the application interacts with the outside world, or values of reactives are observable. *Interactions* interface to the outside world and encapsulate changes to all reactives affected by interactions with it (state directly changed by the interactions, device-local values derived from the changed state, and shared state at remote devices). We use automatic verification with invariants as verification obligations to identify interactions that need coordination across devices, for which the compiler generates the coordination protocol; all other interactions become visible in causal order. This way, the compiler makes an application-specific availability-safety trade-off.

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

In summary, we make the following contributions[2]:

1. A programming model for local-first applications with verified safety properties (Section 2), called LoRe. While individual elements of the model, e.g., CRDTs or reactives, are not novel, they are repurposed, combined, and extended in a unique way to systematically address specific needs of local-first applications with regard to ensuring safety properties.
2. A formal definition of the model including a formal notion of invariant preservation and confluence for interactions, and a modular verification that invariants are never violated. In particular, our model enables invariants that reason about the sequential behaviour of the program. In case of potential invariant violation due to concurrent execution, LoRe automatically adds the necessary coordination logic (see the extended version of this work[2]).
3. A verifying compiler[3] that translates LoRe programs to Viper [28] for automated verification and to Scala for the application logic including synthesized synchronization to guarantee the specified safety invariants (Section 3).
4. An evaluation of LoRe in two case studies (Section 4). Our evaluation validates two claims we make about the programming model proposed, (a) It facilitates the development of safe local-first software, and (b) it enables an efficient and modular verification of safety properties. It further shows that the additional safety properties offered by our model do not come with prohibitive costs in terms of verification effort and time.

2 LoRe in a Nutshell

We introduce the concepts of LoRe along the example of a distributed calendar for tracking work meetings and vacation days. LoRe is an external DSL that compiles to Scala (for execution) and Viper IR [28] (for verification); its syntax is inspired by both. A LoRe program defines a distributed application that runs on multiple physical or virtual devices.[4] Listing 1 shows a simplified implementation of the calendar example application in LoRe. As any LoRe program, it consists of replicated state (`Source` reactives in Lines 2-3), local values derived from them (`Derived` reactives in Lines 5-6), interactions (Lines 8-15), and invariants (Lines 20-23).

2.1 Reactives

Reactives are the composition units in a LoRe program. We distinguish two types of them: *source* and *derived* reactives, declared by the keywords `Source` and `Derived`, respectively. Source reactives are values that are directly changed through interactions. Their state is modeled as *conflict-free replicated data types* (CRDTs) [34, 32] and is replicated between the different devices collaborating on the application. Derived reactives represent local values that are automatically computed by the system from the values of other reactives (source or derived). Changes to source reactives automatically (a) trigger updates of derived reactives and (b) cause devices to asynchronously send update messages to the other devices, which then merge the changes into their local state. Together, local propagations and asynchronous

[2] This is a short version of this work. The extended version is available at: `https://doi.org/10.48550/arXiv.2304.07133`.
[3] The source code of our prototype implementation is available at `https://github.com/stg-tud/LoRe`.
[4] We assume that every device is running the same application code (i.e., the same binary), and different types of devices (such as client and server) are modeled by limiting them to execute a subset of the defined interactions.

Listing 1 The distributed calendar application.

```
1  type Calendar = AWSet[Appointment]
2  val work: Source[Calendar] = Source(AWSet())
3  val vacation: Source[Calendar] = Source(AWSet())
4
5  val all_appointments: Derived[Set[Appointment]]   = Derived{ work.
        toSet.union(vacation.toSet) }
6  val remaining_vacation: Derived[Int] = Derived{ 30 - sumDays(vacation.
        toSet) }
7
8  val add_appointment : Unit = Interaction[Calendar][Appointment]
9    .requires{ cal => a => get_start(a) < get_end(a) }
10   .requires{ cal => a => !(a in cal.toSet)}
11   .executes{ cal => a => cal.add(a) }
12   .ensures { cal => a => a in cal.toSet }
13 val add_vacation : Unit = add_appointment.modifies(vacation)
14   .requires{ cal => a => remaining_vacation - a.days >= 0}
15 val add_work     : Unit = add_appointment.modifies(work)
16
17 UI.display(all_appointments, remaining_vacation)
18 UI.vacationDialog.onConfirm{a => add_vacation.apply(a)}
19
20 invariant forall a: Appointment ::
21   a in all_appointments ==> get_start(a) < get_end(a)
22
23 invariant remaining_vacation >= 0
```

cross-device update messages ensure that users always have a consistent view of the overall application state. All reactives are statically declared in the program source code. LoRe then statically extracts knowledge about the data flow for modular verification and to minimize the proof goals (cf. Sec 3.1). We discuss the technical implications of static reactives in Section 6.

Listing 1 shows two source reactives, `work` and `vacation` (Line 2 and 3), each modeling a calendar as a set of appointments. The work calendar tracks work meetings, while the vacation calendar contains registered vacation days. When defining a source reactive, programmers have to choose a CRDT for the reactive's internal state. LoRe offers a selection of predefined CRDTs including various standard data types such as sets, counters, registers and lists. Further data types can be supported by providing a Viper specification for that data type. In this case, an *add-wins-set* (a set CRDT where additions have precedence over concurrent deletions) is selected for both source reactives. Appointments from both calendars are tracked in the `all_appointments` derived reactive (Line 5), while the `remaining_vacation` reactive (Line 6) tracks the number of remaining vacation days.

2.2 Interactions

Changes to the state of the system, e.g., adding appointments to a calendar, happen through explicit *interactions*. Each interaction has two sets of type parameters: the types of source reactives that it modifies and the types of parameters that are provided when the interaction is applied. For example, the `add_appointment` interaction in Line 8 modifies a reactive of type `Calendar` and takes a parameter of type `Appointment`. The semantics of an interaction `I` are defined in four parts: (1) `requires` (Line 9) defines the preconditions that must hold for `I` to be executed, (2) `executes` (Line 11) defines the changes to source reactives, (3) `ensures` (Line 12) defines the postconditions that must hold at the end of `I`'s execution, (4) `modifies`

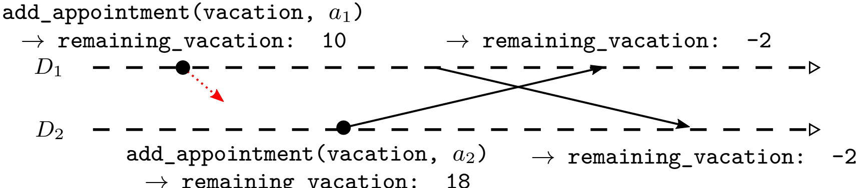

Figure 1 Concurrent execution of interactions may cause invariant violations. In this example, device D_1 adds a vacation of 20 days to the calendar, while D_2 concurrently adds a vacation of 12 days. Given a total amount of 30 available vacation days, this leads to a negative amount of remaining vacation once the devices synchronize.

(Line 13) defines the source reactives that `I` changes. The parameters of `requires`, `executes`, and `ensures` are functions that take the modified reactives and the interaction parameters as input (`cal` is of type `Calendar` and `a` is of type `Appointment`). The splitting of the definition of interactions in four parts allows for modularization and reuse. For instance, `add_appointment` is only a partial specification of an interaction, missing the `modifies` specification. Both `add_work` (Line 15) and `add_vacation` (Line 13) specify complete interactions by adding `modifies` to `add_appointment`; they are independent interactions that differ only in their modifies set.

Interactions encapsulate reactions to input from the outside world (e.g., the callback in Line 18 that is triggered by the UI and applies the arguments to `add_vacation`). Applying an interaction checks the preconditions, and – if they are fulfilled – computes and applies the changes to the source reactives, and propagates them to derived reactives – all in a "transactional" way in the sense that all changes to affected reactives become observable at-once ("atomically"). Only source reactives are replicated between devices, while derived reactives are computed by each device individually. LoRe gurantees that executing interactions does not invalidate neither postconditions nor invariants.

2.3 Invariants and Conflicts

LoRe expects the developer to use *invariants*, introduced with the keyword `invariant`, to specify application properties that should always hold. Invariants are first-order logic assertions given to a verifier based on the Viper verification infrastructure [28]. Invariants can help uncover programming bugs and reveal where the eventually-consistent replication based on CRDTs could lead to safety problems.

For illustration, consider the invariants for the calendar application in Lines 20 and 23. The invariant in Line 20 requires that all appointments must start before they end. Notice, how the invariant can be defined without knowing the amount of calendars and the actual structure of the data-flow graph by simply referring to the `all_appointments` reactive. This invariant represents a form of input validation, and is directly ensured by `add_appointment` interactions because the precondition on the arguments requires the added appointment to start before it ends (Line 9). In absence of this precondition, the LoRe compiler would reject the program and report a safety error due to a possible invariant violation. The invariant in Line 23 requires that employees do not take more vacation days than available to them. Again, this is locally enforced by the precondition of the `add_vacation` interaction, which ensures that new entries do not exceed the remaining vacation days. But there is nothing stopping two devices from concurrently adding vacation entries, which in sum violates the invariant. Figure 1 illustrates such a situation: A user plans a vacation of 20 days on the mobile phone (device D_1) and later schedules a 12-day vacation on a desktop (device D_2), at a time when D_1 was offline. Thus, both interactions happened concurrently and after merging the states the calendar contains a total of 32 days of vacation, violating the `remaining_vacation` invariant.

Figure 2 Overview of LoRe's automated compilation and verification procedure.

This example illustrates a conflict between (concurrent) execution of interactions – in this case, two executions of the `add_vacation` Interaction must be coordinated (synchronized) in order to avoid invariant violations. The LoRe compiler reports conflicting interactions to the developer and automatically synthesizes the required coordination code for the execution of such interactions (see Section 3.3). In a local-first setting, it is of paramount importance to minimize the required coordination to allow offline availability. Reporting of conflicts due to invariants helps developers to explore different situations and make informed decisions about the safety guarantees of their program. When they find that their program requires too much synchronization, they can lower the guarantees by adapting their invariants.

3 Implementation

Figure 2 depicts the architecture of LoRe's verifying compiler. The input to the compiler is a program with its specifications expressed by the invariants, e.g., the program in Listing 1. The output consists of the conflicting interactions and a safe executable program. We use the Viper program verifier to reason about invariant violations and possible conflicts between interactions. We employ an analysis of the data-flow graph to minimize proof obligations to those invariant pairs that may actually conflict.

The rest of this section describes the pipeline from Figure 2 in detail – from left to right, top to bottom.

3.1 Graph Analysis

Checking all pairs of interactions for confluence would result in an exponential amount of proof obligations. To avoid this, we employ a graph analysis to quickly detect pairs of interactions that cannot conflict, because they change completely separate parts of the data-flow graph. For illustration, consider the `add_work` interaction. It modifies the `work` reactive, and – transitively – `all_appointments`. Hence, the reachable reactives are $\{work, all_appointments\}$ and only the first but not the second invariant in Listing 1 overlaps. Thus, neither the `remaining_vacation` reactive, nor the invariant on this reactive will be part of the proof obligation for the `add_work` interaction.

3.2 Automated Verification

We use Viper to classify each interaction into one of the following three categories: 1) *Non-preserving* interactions can violate invariants during execution and are reported as bugs to the developer. 2) *Invariant-preserving* interactions preserve an invariant when executed on a single device but can violate an invariant in the presence of concurrent interactions by other devices. 3) *Invariant-confluent* [4] interactions can be executed concurrently without ever violating an invariant. Whenever two interactions in the second category must not

be executed concurrently to each other, they are *conflicting* and have to be coordinated to ensure invariant-safety. Using the proofs, we can precisely determine the sets of conflicting interactions and automatically synthesize coordination procedures which ensure safety at runtime while limiting the synchronization points to the necessary cases.

3.3 Synchronization at Runtime

Our compiler generates an executable application by converting the data-flow graph to a distributed REScala program [26, 27]. REScala supports all reactive features we require and integrates well with our CRDT-based replication, but has no mechanism for synchronization. LoRe's formal synchronization semantics (see the extended version of this work[2]) could be implemented using any existing form of coordination, such as a central server, a distributed ledger, a consensus algorithm, or a distributed locking protocol. Which choice is suitable, depends on the target application, network size, and the reliability of the chosen transport layer. We use a simple distributed locking protocol for our implementation: Each interaction has an associated lock (represented as a simple token). Whenever a device wants to execute an interaction, it acquires the tokens of all conflicting interactions. If multiple devices request the same token concurrently, the token is given to the device with the lowest ID that requested it. This ensures deadlock freedom; fairness is left for future work. After performing the interaction, the resulting state changes are synchronized with the other devices and the tokens are made available again. Timeouts ensure that whenever a device crashes or becomes unavailable for a longer period of time, its currently owned tokens are released and any unfinished interactions by the device are aborted.

4 Evaluation

Our evaluation aims to validate two claims about LoRe's programming model:
C1: It facilitates the development of safe local-first software.
C2: It enables an efficient and modular verification of safety properties.

We base our validation on two case studies. First, we implemented the standard TPC-C benchmark [36] as a local-first application in LoRe. This case study enables comparing LoRe's model with traditional database-centered development of data processing software and showcasing the benefits of LoRe's verifiable safety guarantees on standard *consistency conditions*. Second, we implemented the running calendar example (Section 2) using Yjs [31]. This case study allows comparing LoRe with an existing framework for local-first applications that we consider a representative of the state-of-the-art.

4.1 Does LoRe facilitate the development of safe local-first software?

4.1.1 Local-first TPC-C

TPC-C models an order fulfillment system with multiple warehouses in different districts, consisting of five *database transactions* alongside twelve *consistency conditions*. We implemented TPC-C in LoRe by mapping database tables to source reactives and derived database values to derived reactives. Each database transaction was modelled as a LoRe interaction.

While modelling the application using reactives might require some adaption from developers not familiar with data-flow programming, we found that using derived reactives led to a more concise and less error-prone design when compared to storing derived values in separate tables. For example, instead of storing the year to date (YTD) value of each TPC-C district in a separate table and updating it each time the payment history changes,

Listing 2 Defining source and derived variables in Yjs.

```
1  const ydoc = new Y.Doc()
2  let work = ydoc.getMap('work');
3  let vacation = ydoc.getMap('
          vacation');
4  let all_appointments;
5  let remaining_vacation = 30;
6
7  work.observe(ymapEvent => {
8   all_appointments = getMap(work,
          vacation);
9  })
10
11 vacation.observe(ymapEvent => {
12  let days_total = getTotalVacDays
          (vacation);
13  remaining_vacation = 30 -
          daysTotal;
14  all_appointments = getMap(work,
          vacation);
15 })
```

Listing 3 Adding appointments in Yjs.

```
1  function addAppointment(calendar,
          appointment) {
2   if(appointment.start <
          appointment.end){
3     calendar.set(appointment.id,
          appointment);
4   }
5  }
6
7  function addVacation(appointment)
8  {
9    let days =
10     appointment.getDays();
11   if(remainingVacation < days){
12     console.log("Sorry, no
          vacation left!");
13   }
14   else{
15     addAppointment(vacation,
          appointment)
16   }
17 }
```

we can model the district YTD as a derived reactive. Following this approach automatically guarantees 9 out of 12 consistency conditions of TPC-C that express consistency requirements between multiple related tables. We were able to phrase the remaining 3 conditions as invariants by directly translating the natural language formulations into logical specifications. To prove them, we additionally needed to specify pre- and postconditions of interactions corresponding to transactions. Other than that, LoRe relieves the TPC-C developer from any considerations of transaction interleavings that could potentially violate the conditions as well as from implementing the synchronization logic, both tedious and error-prone processes.

4.1.2 Yjs-based Calendar

We now compare the LoRe implementation of the distributed calendar to an implementation using the state of the art local-first framework Yjs [31]. Like other solutions for local-first software, Yjs uses a library of CRDTs (usually maps, sets, sequences / arrays, and counters) composed into nested trees – called a *documents* – used to model domain objects.

Source and Derived Variables. For illustration, consider Listing 2, showing how one could implement the domain model of the calendar application. Lines 2 and 3 initialize two CRDTs for the work and vacation calendar. Yjs has no abstraction for derived values and only provides callbacks for reacting to value changes, e.g., Lines 7-15 declare callback methods that update the derived variables in case the Yjs document changes.

Safety Guarantees. Using callbacks to model and manage complex state that changes both in time and in space has issues. It requires that developers programmatically update the derived values once the sources get updated, via local interactions or on receiving updates from other devices, with no guarantees that they do so consistently. It yields a complex control-flow and requires intricate knowledge of the execution semantics to ensure atomicity of updates, let alone to enforce application-level safety properties. Frameworks like Yjs do not offer support for application invariants and thus force developers to integrate custom safety measures at each possible source of safety violations.

Table 1 Seconds to verify combinations of interactions and invariants of the two example applications. Each entry represents the mean verification time over 5 runs with the deviation shown in parentheses.

Distributed Calendar			
Interaction	**Invariant**		
	1	2	
Add vacation	3.32 (± 0.05)	2.97 (± 0.03)	
Remove vacation	3.28 (± 0.06)	3.00 (± 0.02)	
Change vacation	3.32 (± 0.05)	3.04 (± 0.03)	
Add work	3.31 (± 0.04)	–	
Remove work	3.30 (± 0.06)	–	
Change work	3.34 (± 0.06)	–	
TPC-C			
Interaction	**Consistency Condition**		
	3	5	7
New Order	45.4 (± 63.69)	7.63 (± 0.11)	14.49 (± 7.31)
Delivery	5.78 (± 0.03)	5.74 (± 0.07)	5.76 (± 0.09)

In summary, while the replication capabilities of systems like Yjs are valuable for local-first applications, these systems still require the developer to do state management manually. The prevailing use of callbacks and implicit dependencies makes reasoning about the code challenging for both developers and automatic analyses. In contrast, LoRe allows declarative definitions of derived values, with positive effects on reasoning [33, 10]. Moreover, LoRe integrates application invariants as explicit language constructs, which allows for a modular specification and verification and relieves developers from having to consider every involved interaction whenever the specification changes.

4.2 Does LoRe enable efficient and modular verification of safety properties?

To empirically evaluate the performance of LoRe's verifier, we quantify how long it takes to verify different combinations of interactions and invariants of our two case studies. The results are shown in Table 1. The calendar example has two additional types of interactions, which we have not shown in Section 2: removing and changing calendar entries. This leads to a total of 6 interactions (3 per calendar reactive). For TPC-C we only had to verify consistency conditions 3, 5, and 7 because the others were already ensured by the respective derived reactives. The benchmarks were performed on a desktop PC with an AMD Ryzen 7 5700G CPU and 32 GB RAM using Viper's *silicon* verification backend (release v.23.01) [37].

Results. In summary, every interaction/invariant combination in our case studies could be verified in less than a minute. Verification times differed depending mainly on the complexity and length of the interactions and invariants under consideration. Differences become apparent especially when looking at the results for TPC-C. Proofs involving the *New Order* interaction, which is the most "write-heavy" interaction of TPC-C that changes many source reactives at once, generally took longer to verify than others. For *New Order*, we also observe a much higher deviation of up to 64 seconds which we assume to be caused

by internal Z3 heuristics[5]. When interpreting the results, it is important to note that each interaction/invariant combination has to be verified only once and independently of other combinations. Large-scale applications can be verified step-by-step by splitting them into smaller pieces. Furthermore, we limit the need for verification to potential conflicts that we derive from the reactive data-flow graph. Programmers can add new functionality to the application (i.e., specify interactions) and only have to reason about the properties of that new functionality (i.e., specify its invariants) and the system ensures global safety – at only the cost of the amount of overlap with existing functionality. This allows for an incremental development style, where only certain parts of programs have to be (re-)verified, when they have been changed or added.

5 Related Work

Our work relates to three areas: distributed datatypes, formal reasoning, and language-based approaches. Sections below relate work from each area to respective aspects of our approach.

5.1 Consistency Through Distributed Data Types

Conflict-Free Replicated Datatypes (CRDTs)[34, 32] are a building block for constructing systems and applications that guarantee eventual consistency. CRDTs are used in distributed database systems such as *Riak* [18] and *AntidoteDB* [1]. These databases make it possible to construct applications that behave under mixed consistency, but unlike our approach, they leave reasoning about application guarantees to the programmer. Several works [12, 30] propose frameworks for formally verifying the correctness of CRTDs, while others [16, 20] focus on synthesizing correct-by construction CRDTs from specifications.

De Porre et al. [9, 8] suggest *strong eventually consistent replicated objects (SECROs)* relying on a replication protocol that tries to find a valid total order of all operations. Similar to LoRe, *SECROs* [9, 8] and *Hamsaz* [14] extend upon the eventually consistent replication of CRDTs by automatically choosing the right consistency level based on application invariants. Both approaches tie consistency and safety properties to specific datatypes/objects. This is not sufficient to guarantee end-to-end correctness of an entire local-first application - consistency bugs can still manifest in derived information (e.g., in the user interface).

5.2 Automated Reasoning about Consistency Levels

Our formalization is in part inspired by the work of Balegas et al. [6, 7] on *Indigo*. The work introduces a database middleware consisting of transactions and invariants to determine the ideal consistency level – called *explicit consistency*. They build on the notion of *invariant-confluence* for transactions that cannot harm an invariant which was first introduced by Bailis et al. [4]. While they work on a database level, we show how to integrate this reasoning approach into a programming language. An important difference between our *invariant-confluence* and the one by Balegas et al. [6] is that our approach also verifies local preservation of invariants, whereas their reasoning principle assumes invariants to always hold in a local context. In a more recent work called *IPA*, Balegas et al. [5] propose a static analysis technique that aims at automatically repairing transaction/invariant conflicts without adding synchronization between devices. We consider this latter work complementary to ours.

[5] These could likely be improved by annotating quantifiers in invariants and pre-/postconditions with hand-crafted trigger expressions [28].

Whittaker and Hellerstein [38] also build on the idea of invariant-confluence and extend it to the concept of *segmented invariant-confluence*. Under segmented invariant-confluence, programs are separated into segments that can operate without coordination and coordination only happens in between the segments. The idea is similar to our definition of *conflicting interactions*, however, their procedure cannot suggest a suitable program segmentation, but requires developers to supply them.

The *SIEVE* framework [22] builds on the previous work on *Red/Blue-Consistency* [23] and uses invariants and program annotations to infer where a Java program can safely operate under CRDT-based replication (*blue*) and where strong consistency is necessary (*red*). They do so by relying on a combination of static and dynamic analysis techniques. Compared to *SIEVE*, our formal reasoning does not require any form of dynamic analysis. *Blazes* [2] is another analysis framework that uses programmer supplied specifications to determine where synchronization is necessary to ensure eventual consistency. Contrary to *Blazes*, LoRe ensures that programs are "by design" at least eventually consistent, while also allowing the expression and analysis of programs that need stronger consistency. *Q9* [15] is a bounded symbolic execution system, which identifies invariant violations caused by weak consistency guarantees. Similar to our work, *Q9* can determine where exactly stronger consistency guarantees are needed to maintain certain application invariants. However, its verification technique is bound by the number of possible concurrent operations. LoRe can provide guarantees for an unlimited amount of devices with an unlimited amount of concurrent operations.

5.3 Language Abstractions for Data Consistency

We categorize language-based approaches based on how they achieve consistency and on the level of programmer involvement.

Manual Choice of Consistency Levels. Li et al. [23] propose *RedBlue Consistency* where programmers manually label their operations to be either blue (eventually consistent) or red (strongly consistent). In MixT [25], programmers annotate classes with different consistency levels and the system uses an information-flow type system to ensure that the requested guarantees are maintained. However, this still requires expert knowledge about each consistency level, and wrong choices can violate the intended program semantics. Other approaches [29, 19] expect programmers to choose between *consistency* and *availability*, again leaving the reasoning duty about consistency levels to the programmer. Compared to LoRe, languages in this category place higher burden on programmers: They decide which operation needs which consistency level, a non-trivial and error-prone selection.

Automatically Deriving Consistency from Application Invariants. *CAROL* [21] uses CRDTs to replicate data and features a refinement typing discipline for expressing safety properties similar to our *invariants*. Carol makes use of pre-defined datatypes with *consistency guards* used by the type system to check for invariant violations. The compatibility of datatype operations and consistency guards is verified ahead of time using an algorithm for the Z3 SMT solver. This approach hides much of the complexity from the programmer, but the abstraction breaks once functionality that is not covered by a pre-defined datatype is needed. Unlike Carol, LoRe does not rely on predefined consistency guards, but allows the expression of safety properties as arbitrary logical formulae. Additionally, *CAROL* only checks the concurrent interactions of a program for invariant violations, whereas LoRe verifies the overall application including non-distributed parts. Sivaramakrishnan et al. [35] propose *QUELEA*, a declarative language for programming on top of eventually consistent datastores. It features a contract-language to express application-level invariants and automatically

generates coordination strategies in cases where invariants could be violated by concurrent operations. *QUELEA*'s contract-language requires programmers to express the desired properties using low-level visibility relations, which can be challenging to get right for non-experts. LoRe avoids this intermediate reasoning and automatically derives the right level of consistency for satisfying high-level safety invariants to enable end-to-end correctness.

Automating Consistency by Prescribing the Programming Model. Languages in this category seek to automate consistency decisions by prescribing a certain programming model such that certain consistency problems are impossible to occur. In *Lasp* [24], programmers model the data flow of their applications using combinator functions on CRDTs. Programs written in *Lasp* always provide eventual consistency but contrary to LoRe, *Lasp* does not allow arbitrary compositions of distributed datatypes. *Bloom* [3] provides programmers with ways to write programs that are *logically monotonic* and therefore offer automatic eventual consistency. Both *Lasp* and *Bloom*, however, are not meant to formulate programs that need stronger consistency guarantees. LoRe is similar to *Lasp* and *Bloom* in the sense that we also prescribe a specific – reactive – programming style. However, our programming model is less restrictive and allows arbitrary compositions of distributed datatypes. This is enabled by leveraging the composability properties of reactive data-flow graphs. Secondly, LoRe provides a principled way to express hybrid consistency applications with guarantees stronger than eventual consistency. Drechsler et al. [11] and Mogk et al. [26, 27] also use a reactive programming model similar to ours to automate consistency in presence of multi-threading respectively of a distributed execution setting. However, they do not support a hybrid consistency model. Drechsler et al. [11] enable strong consistency (serializability) only, while Mogk et al. [26, 27] support only eventual consistency.

6 Conclusion and Future Work

In this paper, we proposed LoRe, a language for local-first software with verified safety guarantees. *LoRe* combines the declarative data flow of reactive programming with static analysis and verification techniques to precisely determine concurrent interactions that could violate programmer-specified safety properties. We presented a formal definition of the programming model and a modular verification that detects concurrent executions that may violate application invariants. In case of invariant violation due to concurrent execution, LoRe automatically enforces the necessary amount of coordination. LoRe's verifying compiler translates LoRe programs to Viper [28] for automated verification and to Scala for the application logic including synthesized synchronization to guarantee the specified safety invariants. An evaluation of LoRe's programming model in two case studies confirms that it facilitates the development of safe local-first applications and enables efficient and modular automated reasoning about an application's safety properties. Our evaluation shows that verification times are acceptable and that the verification effort required from developers is reasonable.

In the future, it would be desirable to integrate existing libraries of verified CRDTs [12] or even solutions that allow ad-hoc verification of CRDT-like datatypes [30, 20]. This would enable us to support a wider range of data types or even allow programmers to use custom distributed datatypes, which can be verified to be eventually consistent. Furthermore, our current data-flow analysis is limited to static data-flow graphs. While static reasoning about dynamic graphs is impossible in the general case, most applications make systematic use of dynamic dependencies, and we believe it would be feasible to support common cases.

References

1. Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016. doi:10.1109/ICDCS.2016.98.
2. Peter Alvaro, Neil Conway, Joseph M Hellerstein, and David Maier. Blazes: Coordination analysis for distributed programs. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, March 2014. doi:10.1109/ICDE.2014.6816639.
3. Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency Analysis in Bloom: A CALM and Collected Approach. In *CIDR*. Citeseer, 2011. URL: http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf.
4. Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, November 2014. doi:10.14778/2735508.2735509.
5. Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases. *Proceedings of the VLDB Endowment*, 12(4):404–418, December 2018. doi:10.14778/3297753.3297760.
6. Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*, pages 1–16, 2015. doi:10.1145/2741948.2741972.
7. Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Towards Fast Invariant Preservation in Geo-replicated Systems. *ACM SIGOPS Operating Systems Review*, 49(1):121–125, January 2015. doi:10.1145/2723872.2723889.
8. Kevin De Porre, Carla Ferreira, Nuno M. Preguiça, and Elisa Gonzalez Boix. Ecros: building global scale systems from sequential code. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485484.
9. Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. Putting order in strong eventual consistency. In *Distributed Applications and Interoperable Systems*, pages 36–56, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-22496-7_3.
10. Moritz Dinser. An empirical study on reactive programming. Master's thesis, Technische Universität Darmstadt, 2021. URL: http://tubama.ulb.tu-darmstadt.de/id/eprint/30079.
11. Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Thread-Safe Reactive Programming. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), October 2018. doi:10.1145/3276477.
12. Victor B F Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. Verifying Strong Eventual Consistency in Distributed Systems. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):109:1–109:28, October 2017. doi:10.1145/3133933.
13. Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini. Lore: A programming model for verifiably safe local-first software, 2023. arXiv:2304.07133.
14. Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication Coordination Analysis and Synthesis. *Proceedings of the ACM on Programming Languages*, 3(POPL):74:1–74:32, January 2019. doi:10.1145/3290387.
15. Gowtham Kaki, Kapil Earanky, K C Sivaramakrishnan, and Suresh Jagannathan. Safe Replication through Bounded Concurrency Verification. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), October 2018. doi:10.1145/3276534.
16. Gowtham Kaki, Swarn Priya, K C Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):154:1–154:29, October 2019. doi:10.1145/3360580.

17 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, New York, NY, USA, October 2019. Association for Computing Machinery. doi:10.1145/3359591.3359737.

18 Rusty Klophaus. Riak Core: Building Distributed Applications without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming*, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1900160.1900176.

19 Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. Rethinking Safe Consistency in Distributed Object-Oriented Programming. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):188:1–188:30, 2020. doi:10.1145/3428256.

20 Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: Synthesizing CRDTs with verified lifting. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):173:1349–173:1377, October 2022. doi:10.1145/3563336.

21 Nicholas V Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černý. Sequential Programming for Replicated Data Stores. *Proceedings of the ACM on Programming Languages*, 3(ICFP):106:1–106:28, July 2019. doi:10.1145/3341710.

22 Cheng Li, João Leitão, Allen Clement, Nuno M. Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, 2014. USENIX Association. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2.

23 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguic, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 265–278, 2012. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

24 Christopher Meiklejohn and Peter Van Roy. Lasp: A Language for Distributed, Coordination-free Programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 184–195, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2790449.2790525.

25 Mae Milano and Andrew C Myers. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192375.

26 Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2018.1.

27 Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A Fault-tolerant Programming Model for Distributed Interactive Applications. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):144:1–144:29, October 2019. doi:10.1145/3360570.

28 P Müller, M Schwerhoff, and A J Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583, Berlin, Heidelberg, 2016. Springer-Verlag. doi:10.1007/978-3-662-49122-5_2.

29 Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. A CAPable Distributed Programming Model. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3276954.3276957.

30 Sreeja S Nair, Gustavo Petri, and Marc Shapiro. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems*, pages 544–571, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-44914-8_20.

31 Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 19th International Conference on Supporting Group Work, Sanibel Island, FL, USA, November 13 - 16, 2016*. Association for Computing Machinery, 2016. doi:10.1145/2957276.2957310.

32 Nuno Preguiça. Conflict-free Replicated Data Types: An Overview. *ArXiv*, June 2018. doi:10.48550/arXiv.1806.10254.

33 Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering*, 43(12), 2017. doi:10.1109/TSE.2017.2655524.

34 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011. URL: https://hal.inria.fr/inria-00555588.

35 KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, June 2015. Association for Computing Machinery. doi:10.1145/2737924.2737981.

36 TPC. TPC-C Specification 5.11.0, 2021. URL: http://tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf.

37 Viper. Viperproject/silicon Github Repository. Viper Project, April 2021. URL: https://github.com/viperproject/silicon.

38 Michael Whittaker and Joseph M Hellerstein. Interactive Checks for Coordination Avoidance. *Proceedings of the VLDB Endowment*, 12(1):14–27, September 2018. doi:10.14778/3275536.3275538.