SPLASH: G: LoRe: Local-First Reactive Programming with Verified Safety Guarantees

Julian Haas
Technische Universität Darmstadt
Germany
haas@cs.tu-darmstadt.de

1 Problem & Motivation

Nowadays, the dominant design approach in distributed software is cloud-centric. This comes at the cost of several issues including loss of control over data ownership and privacy, lack of offline availability, poor latency, inefficient use of communication infrastructure, and waste of (powerful) computing resources on the edge. *Local-first software* [15] presents an alternative approach where data is stored and managed locally and devices primarily communicate in a peer-to-peer manner. The approach advocates making as much as possible of the application functionality available locally – even when the device is offline – while still offering the benefits that users expect from modern cloud software such as multi-device synchronization and live collaboration.

Unfortunately, local-first software is hard to implement. The decentralized nature of local-first software forces developers to reason about the interleavings of dataflow of interactions through local data dependency chains to remote devices and back – a highly complex task even for small applications. As a further complication, existing solutions [2, 14] for local-first software usually employ weaker consistency models such as causal consistency in order to maximize the availability of offline-operations [17]. Despite these challenges, existing solutions to develop local-first software do not provide safety guarantees and instead expect developers to manually reason about concurrent interactions in an environment with unreliable network conditions.

As an example, consider a distributed local-first calendar application that is used for organizing work meetings and vacation days. Users can modify the calendar concurrently from multiple devices and can also access it during offline periods. Now, imagine we wanted to enforce the following application invariant: *"Employees are not allowed to enter more than the available vacation days."*. This simple example is illustrative of invariants that are impossible to maintain using the previously described weakly consistent replication strategies. Figure 1 illustrates the conflict that would occur if two devices concurrently try to perform a conflicting "add_appointment" interaction: Starting with 30 days of vacation, device D_1 adds a vacation of 20 days to the calendar while being offline. Device D_2 concurrently adds 18 days which leads to a negative amount of remaining vacation, once both devices are synchronized again. In order to prevent such conflicts, developers currently have to implement custom coordination logic on top of the existing weakly-consistent local-first frameworks which is a challenging and error-prone endeavour. Introducing too much coordination harms the availability of the application while too little coordination can violate application correctness [7].

In this work, we propose *LoRe*: A local-first reactive programming language which guarantees developer-specified safety-invariants by design. The LoRe compiler builds on automatic deductive verification tools [20] to prove invariant-safety of a given program. By default, LoRe programs offer causal consistency through the use of Conflict-Free Replicated Datatypes (CRDTs) [21, 24] in order to maximize offline availability. Whenever our verification procedure detects invariant violations due to the weak consistency level offered by CRDTs, we automatically synthesize coordination code for these cases and thus lift the offered consistency level to strong consistency (serializability). LoRe is implemented as a Scala DSL and compiles to executable Scala code.

2 Background & Related Work

2.1 Local-First Programming

The current approach to develop local-first applications [15, 23] combines two techniques: *Conflict-free replicated data types (CRDTs)* and *Reactive programming (RP)*.

CRDTs [6, 8, 21, 24] provide an easy and accessible way for developers to replicate data between different devices in a distributed application: On the surface, they offer a familiar programming interface that resembles traditional data types such as counters, registers, sets and lists. Internally, however, CRDTs allow merging remote updates into the local state while ensuring that these merges always *converge*, i.e. two devices that have seen the same updates will always end up in the same state. While CRDTs are a convenient way for achieving eventual/weak consistency in a distributed application, they are unsuited to maintain application invariants that require coordination among the participants such as the one seen in the previous example.

Reactive programming (RP) is a declarative, dataflow-based programming model incarnated in popular web frameworks such as Facebook's React [13] or the web programming language Elm [9]. RP provides a principled way to react to events coming from different sources, such as user interactions or sensor readings which makes the approach

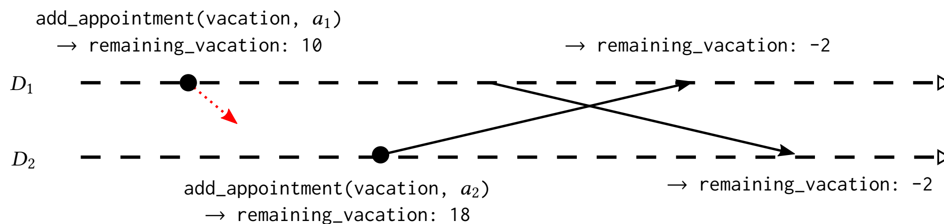

Figure 1. Concurrent execution of interactions may cause invariant violations. In this example, device D_1 adds a vacation of 20 days to the calendar, while D_2 concurrently adds a vacation of 12 days. Given a total amount of 30 available vacation days, this leads to a negative amount of remaining vacation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

particularly suited for interactive, user-focused applications such as local first-software. When combined with CRDTs, reactive-programming enables unified handling of local and remote updates, as we can simply integrate CRDT updates with the already existing event pipeline.

Prior work has provided evidence for the effectiveness of combining RP with CRDTs independently in academic [18] and industrial [27] contexts. However, LoRe is – to the best of our knowledge – the first work to explore the synergies of the two concepts for formal program verification.

2.2 Coordination Avoidance and Mixed-Consistency Objects

The availability-safety trade-off that LoRe addresses for local-first software, has previously been explored in the systems and database community under the term *coordination avoidance* [3, 4]. Coordination avoidance captures the idea that coordination between different devices should be limited to those cases where it is necessary to ensure application specific correctness criteria. In this regard, *invariant-confluence* [3, 28] and *logical-monotonicity* [1, 11] provide formal definitions for determining if a certain operation is safe to perform without coordination.

Related approaches from the area of mixed-consistency objects address the problem by extending the capabilities of CRDTs with additional non-convergent operations that require coordination [10, 12, 16, 25].

Our work builds on the aforementioned ideas but differs in two significant ways: 1) Unlike related work that assumes a replicated database, we target a local-first setting without any centralized authority. 2) To the best of our knowledge, we provide the first language-based solution to mixed-consistency with whole program guarantees. Related works mainly target the programming interface to a distributed datastore/object but leave derived values out of scope. Following this approach, data can be safe and consistent at the datastore level, but inconsistencies can still occur at the application level. In contrast, we can control the consistency of derived data “all the way down” by leveraging the reactive data flow.

3 Uniqueness of the Approach

This work presents LoRe, the first dedicated programming model for verified mixed-consistency local-first applications. We present our approach in two subsections: The first subsection introduces LoRe’s programming model by example while the second focuses on the verification procedure employed by the LoRe compiler.

3.1 Programming Model

LoRe is a distributed functional reactive programming language [19, 22] which is built around three main programming abstractions: *reactives*, *interactions* and *invariants*. Listing 1 demonstrates how the distributed calendar application introduced in Section 1 could be implemented in LoRe.

Reactives (line 2-8) allow the expression of time-changing values and model the data flow of a LoRe program. They come in two flavours: *Source reactives* and *derived reactives*. *Source reactives* are based on *conflict-free replicated data types* (CRDTs) [24]. They allow programmers to replicate state between the different devices running a LoRe program. In this case, we define two source reactives (line 2 and 3): one representing a calendar for work appointments and one representing a calendar for vacation entries.

While the value of source reactives can be manipulated directly through *interactions*, the values of *derived reactives* depend on the values of other reactives called *inputs*. The calendar application contains two derived reactives (line 5-8): `all_appointments` tracks appointments from both calendars and `remaining_vacation` tracks the amount of remaining vacation days. Whenever one of their inputs changes, the value of the derived reactives is updated automatically.

Interactions (line 10-19) explicitly model interactions with the user or the outside world. Examples would be button-presses, sensor-readings or more complex interactions like booking a ticket. Interactions can cause the values of source reactives to change and their effects are applied transactionally, i.e. either all effects of an interaction are visible to the user or none.

The calendar application defines two interactions called `add_vacation` (line 15) and `add_work` (line 18) which both build on the same base interaction `add_appointment` (line 10). A LoRe interaction is defined in four parts: 1) `requires` (line 11-12)

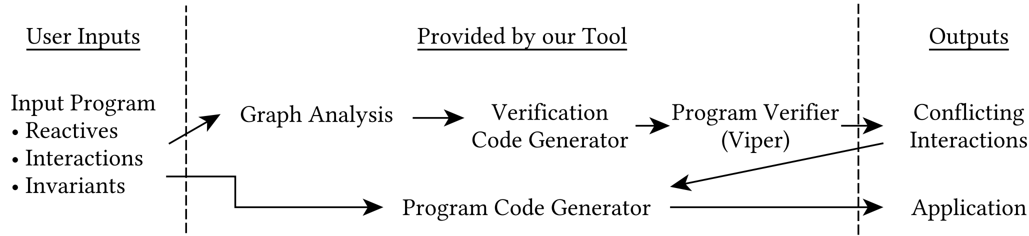

Figure 2. Overview of *LoRe*'s compilation and verification process. Conflicts reported by the program verifier are used to synthesize coordination code for the affected interactions.

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

Listing 1. Excerpt of a distributed calendar application implemented in LoRe. The application models two calendars, one for work meetings and one for vacation days. Both calendars can be modified via the respective `add_work` and `add_vacation` interactions. The two invariants ensure that appointments start before they end and that the amount of available vacation days is never exceeded.

defines the preconditions that must hold for the interaction to be executed, 2) `executes` (line 13) defines the changes to source reactives, 3) `ensures` (line 14) defines the postconditions that must hold after the interaction and 4) `modifies` (line 16 and 19) defines the source reactives that this interaction changes.

Invariants (line 21-23) allow programmers to formulate custom safety requirements of their program. They are expressed as first-order logical formulae which restrict the allowed values of reactives. The calendar application defines two invariants: The first invariant (line 21-22) states that valid appointments must start before they end. The second invariant (line 23) states that the amount of available vacation days must never be exceeded.

3.2 Verification of LoRe Programs

Safety of LoRe programs is automatically verified during the compilation process using the Viper program verifier [20]. Figure 2 gives an overview of the process: Our tool starts by analyzing the data-flow graph of the reactive input program to determine which interactions affect which safety invariants. Afterwards, we synthesize proofs for the relevant interactions in Viper's intermediate verification language (Viper IR). Viper IR is a simple language with support for permission-based reasoning, designed for expressing programs along with their proofs. The Viper proofs allow us 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* [3] 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, we call them *conflicting*. They 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 to the necessary cases.

4 Results & Contributions

4.1 Evaluation

We evaluated LoRe along two research questions:

- RQ1) Is LoRe expressive enough to implement real-world local-first software?
- RQ2) What is the overhead of the automated verification procedure?

To answer these questions, we implemented two case studies: An extended version of the local-first calendar presented

Table 1. Seconds to verify combinations of interactions and invariants of the two example applications.

Distributed Calendar			
Interaction		**Invariant**	
		1	2
Add vacation		3.27	5.15
Remove vacation		4.86	4.13
Change vacation		4.61	4.99
Add work		3.92	–
Remove work		4.37	–
Change work		5.61	–
TPC-C			
Interaction		**Consistency Condition**	
	3	5	7
New Order	11.75	7.63	6.36
Delivery	3.99	3.92	3.87

in Listing 1 and the standardized TPC-C transaction processing benchmark [26]. TPC-C models an order fulfillment system with multiple warehouses in different districts, consisting of five database transactions alongside twelve consistency conditions. While TPC-C is not a typical local-first application, it allowed us to test the generality of our approach and provided us with pre-defined safety invariants of realistic size in order to test the performance of our verifier.

Results. *RQ1)* Our case studies show that LoRe is expressive enough even for implementing more complex business applications like TPC-C. Perhaps surprisingly, we found that database focussed software like TPC-C still benefits from the reactive programming model: Modelling data dependencies explicitly via *derived reactives* eliminates several classes of errors that can occur in a standard relational database model (e.g. loss of referential integrity). As a result, we only had to model 3 out of 12 consistency conditions as invariants. The remaining 9 consistency conditions express dependencies between database tables and derived data that could be modeled via derived reactives, and thus are always fulfilled by design.

RQ2) Table 1 shows the results of our verifier performance experiments. The verification tasks were performed on a standard desktop PC with an AMD Ryzen 5 3600 CPU. A "–" in the table denotes that the graph analysis step (see Figure 2) could already determine that the given interaction never affects the invariant in question. For example, the "add work" interaction never affects invariant 2 which restricts the available vacation days. For TPC-C, we only had to verify two interactions because the other interactions did not affect any of the remaining three consistency conditions.

Overall, we were able to verify every invariant/interaction combination in less than 15s with most combinations needing around 5s to complete. We consider this overhead acceptable in relation to the provided safety guarantees.

4.2 Contributions

In summary, we make the following contributions:

- We presented the first programming model for local-first applications with verified safety properties 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.
- We implemented a verifying compiler that translates LoRe programs to Viper [20] for automated verification and to Scala for the application logic including synthesized synchronization to guarantee the specified safety invariants.
- We evaluated LoRe in two case studies. Our evaluation shows that LoRe is expressive enough to implement real-world local-first software and that the additional safety properties offered by our model do not come with prohibitive costs in terms of verification effort and time.

The research presented in this paper is also part of an upcoming ECOOP 2023 paper of which I am the first author.

5 Conclusion & Future Work

One of the motivations behind this work was to empower developers with a principled and automated model to reason about consistency choices in local-first software. While LoRe is a first step in systematically exploring these trade-offs, selecting the right consistency level for a local-first application and balancing safety with e.g. offline-support remains a challenge. Future work could provide a better understanding by collecting more empirical evidence, and integrating other conflict resolution strategies such as reordering operations or using bounded CRDTs [5]. In addition, LoRe's verification and graph analysis is currently limited to static data-flow graphs. In the future, we would like to explore possibilities for allowing certain dynamic changes to the data-flow while still offering static and verifiable guarantees.

References

[1] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and David Maier. 2014. Blazes: Coordination Analysis for Distributed Programs. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 52–63. https://doi.org/10.1109/ICDE.2014.6816639

[2] Automerge contributors. 2023. Automerge: Build Local-First Software. https://automerge.org/ (Accessed on 2023-04-19).

[3] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proceedings of the VLDB Endowment* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[4] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. 2018. IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases. *Proceedings of the VLDB Endowment* 12, 4 (Dec. 2018), 404–418. https://doi.org/10.14778/3297753.3297760

[5] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. 2015. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. 31–36. https://doi.org/10.1109/SRDS.2015.32

[6] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. An Optimized Conflict-Free Replicated Set. https://doi.org/10.48550/arXiv.1210.3368 arXiv:arXiv:1210.3368

[7] Eric A. Brewer. 2012. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer* 45, 2 (2012), 23–29. https://doi.org/10.1109/MC.2012.37

[8] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 271–284. https://doi.org/10.1145/2535838.2535848

[9] Evan Czaplicki. 2021. Elm - Delightful Language for Reliable Web Applications. https://elm-lang.org/ (Accessed on 2023-04-19).

[10] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. 2021. ECROs: Building Global Scale Systems from Sequential Code. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 107:1–107:30. https://doi.org/10.1145/3485484

[11] Joseph M Hellerstein and Peter Alvaro. 2019. Keeping CALM: When Distributed Consistency Is Easy. *CoRR* abs/1901.0 (2019). arXiv:1901.01930 http://arxiv.org/abs/1901.01930

[12] Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: Replication Coordination Analysis and Synthesis. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 74:1–74:32. https://doi.org/10.1145/3290387

[13] Meta Platforms Inc. 2023. React: The Library for Web and Native User Interfaces. https://reactjs.org/ (Accessed on 2023-04-19).

[14] Kevin Jahns. 2023. Yjs: Modular Building Blocks for Building Collaborative Applications like Google Docs and Figma. https://docs.yjs.dev/ (Accessed on 2023-04-19).

[15] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. 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)*. Association for Computing Machinery, New York, NY, USA, 154–178. https://doi.org/10.1145/3359591.3359737

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

[17] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. 2011. Consistency, Availability, and Convergence. In *University of Texas at Austin Tech Report*. https://www.cs.utexas.edu/users/dahlin/papers/cac-tr.pdf

[18] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-Tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:26. https://doi.org/10.4230/LIPIcs.ECOOP.2018.1

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

[20] P Müller, M Schwerhoff, and A J Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, B Jobstmann and K R M Leino (Eds.), Vol. 9583. Springer-Verlag, Berlin, Heidelberg, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[21] Nuno Preguiça. 2018. Conflict-Free Replicated Data Types: An Overview. *ArXiv* (June 2018). https://doi.org/10.48550/ARXIV.1806.10254

[22] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging between Object-Oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity - MODULARITY '14*. ACM Press, New York, New York, USA, 25–36. https://doi.org/10.1145/2577080.2577083

[23] Nicholas Schiefer, Geoffrey Litt, Johannes Schickling, and Daniel Jackson. 2022. Building Data-Centric Apps with a Reactive Relational Database. https://riffle.systems/essays/prelude/ (Accessed on 2023-04-19).

[24] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A Comprehensive Study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

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

[26] TPC. 2010. TPC-C Specification 5.11.0. http://tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf (Accessed on 2023-04-19).

[27] Peter van Hardenberg and Martin Kleppmann. 2020. PushPin: Towards Production-Quality Peer-to-Peer Collaboration. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3380787.3393683

[28] Michael Whittaker and Joseph M. Hellerstein. 2020. Checking Invariant Confluence, in Whole or in Parts. *SIGMOD Rec.* 49, 1 (Sept. 2020), 7–14. https://doi.org/10.1145/3422648.3422651