Invariant Safety for Distributed Applications

Sreeja S Nair
Sorbonne Université—LIP6 & Inria, Paris, France
sreeja.nair@lip6.fr

Gustavo Petri
ARM Research, Cambridge, UK
gustavo.petri@arm.com

Marc Shapiro
Sorbonne Université—LIP6 & Inria, Paris, France
marc.shapiro@acm.org

ABSTRACT

We study a proof methodology for verifying the safety of data invariants of highly-available distributed applications that replicate state. The proof is (1) modular: one can reason about each individual operation separately, and (2) sequential: one can reason about a distributed application as if it were sequential. We automate the methodology and illustrate the use of the tool with a representative example.

KEYWORDS

Replicated data, Consistency, Automatic verification, Distributed application design, Tool support

ACM Reference Format:
Sreeja S Nair, Gustavo Petri, and Marc Shapiro. 2019. Invariant Safety for Distributed Applications. In *Proceedings of Principles and Practice of Consistency for Distributed Data (PaPoC'19).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A distributed application often replicates its data to several locations, and accesses the closest available replica. Examples include social networks, multi-user games, co-operative engineering tools, collaborative editors, source control repositories, or distributed file systems. To ensure availability, an update must not synchronise across replicas; otherwise, when a network partition occurs, the system will block. Asynchronous updates may cause replicas to diverge or to violate the data invariants of the application.

To address the first problem, Conflict-free Replicated Data Types (CRDTs)[13] have mathematical properties to ensure that all replicas that have received the same set of updates converge to the same state [13]. To ensure availability, a CRDT replica executes both queries and updates locally and immediately, without remote synchronisation. It propagates its updates to the other replicas asynchronously.

There are two basic approaches to update propagation: to propagate operations, or to propagate states. In the former approach, an update is first applied to some origin replica, then sent as an operation to remote replicas, which in turn apply it to update their local state. Operation-based CRDTs require the the message delivery layer to deliver messages in causal order, exactly once; the set of replicas must be known.

In the latter approach, an update is applied to some origin replica. Occasionally, one replica sends its full state to some other replica, which merges the received state into its own. In turn, this replica will later send its own state to yet another replica. As long as every update eventually reaches every replica transitively, messages may be dropped, re-ordered or duplicated, and the set of replicas may be unknown. Replicas are guaranteed to converge if the set of states, as a result of updates and merge, forms a monotonic semi-lattice [13]. Due to these relaxed requirements, state-based CRDTs have better adoption [1]. They are the focus of this work.

As a running example, consider a simple auction system. The state of an auction consists of status, a set of bids, and a winner. This state is replicated at multiple servers; CRDTs ensures that all replicas eventually converge. Users at different locations can start an auction, place bids, close the auction, declare a winner, inspect the local replica, and observe if a winner is declared and who it is. All replicas will eventually agree on the same auction status, same set of bids and the same winner.

However, the application may also require to maintain a correctness property or *invariant* over the data. An invariant is an assertion on application data that must evaluate to true in every state of every replica. For instance, the auction's invariant is that: when the auction is closed, there is a winner; there is a single winner; and the winner's bid is the highest.

Such an invariant is easy to ensure in a sequential system, but concurrent updates might violate it. In this case, the application would need to synchronise some updates between replicas, in order to maintain the invariant. For instance, in the absence of sufficient synchronisation, a replica might close the auction and declare a winner, while concurrently a user at a different replica is placing a higher bid.

PaPoC'19, March 25, Dresden, Germany

ACM ISBN 978-1-4503-6276-4…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This problem has been addressed before, by stating correctness rules and proof obligations; however, previous work considers only the operation-based approach [6, 8, 10].

In this paper, we propose a proof methodology for applications that use state-based CRDTs. We exploit the properties of state-based CRDTs to reason about a concurrent system in a sequential manner. We have also developed a tool named Soteria, to automate our proof rule. Soteria detects concurrency bugs and provides counterexamples.

2 SYSTEM MODEL

An application consists of state, some operations, a merge function, and an invariant. The state is replicated at any number of replicas. A client chooses any arbitrary replica for its next operation, called the *origin replica* for that operation. A replica occasionally sends its state to some other replica, which the receiving replica *merges* into its own state. In summary, the state of any given replica changes, either by executing an update operation for which it is the origin, or by merging the state received from a remote replica. Each replica is sequential. A merge is the only point where a replica observes concurrent operations submitted to other replicas. Each replica executes sequentially, one local update or merge at a time; equivalently, an update or merge operation executes atomically, even if it updates multiple data items.

An application invariant is an assertion over state. The invariant must evaluate true in every state of every replica. Despite being evaluated against local state, an invariant is in effect global, since it must be true at all replicas, and replicas eventually converge.

Figure 1 depicts the evolution of state in our auction application. Each line represents a replica, time progressing from left to right. A box represents local state. A curved arrow represents an update operation, labelled with the operation name. A diagonal arrow shows propagation (labelled with the propagated state), merged at the receiving replica.

We assume here that application state is a composition of CRDTs. This is not a limitation, since many basic CRDT types have been proposed, which extend familiar sequential data types with a concurrency semantics [12].

The CRDT convergence rules, or *lattice rules*, are the following. The state of a replica progresses monotonically with time. The set of states forms a semi-lattice, i.e., is equipped with a partial order and a least-upper-bound function. A state transition represents the execution of either an update operation or merge. An update is an inflation, i.e., the resulting state is no less (in the partial order) than the previous one. *Merge* computes a state which is the least-upper-bound of the current local state and the received remote state.

We write σ for the current state of a replica, σ_{new} for the new state after an operation or merge, and σ' for the incoming state from a remote replica. In the Boogie specification language (used by our tool, described in Section 4), we denote operation execution with the keyword `call`, an assertion with `assert`, and assumptions with `assume`. An operation op executes only if its `precondition` is true. Thus, we can write the above lattice conditions as follows:

- An update operation op is an inflation.

  `call` $\sigma_{new} = \mathsf{op}(\sigma)$
  `assert` $\sigma_{new} \geq \sigma$

- Merge is a least upper bound

  `call` $\sigma_{new} = \mathsf{merge}(\sigma, \sigma')$
  `assert` $\sigma_{new} \geq \sigma \wedge \sigma_{new} \geq \sigma'$ `#upper bound`
  `assert` $\forall \sigma^*, \sigma^* \geq \sigma \wedge \sigma^* \geq \sigma' \implies \sigma^* \geq \sigma_{new}$ `#least`

Let us illustrate with the auction example (for simplicity we consider a single auction). Its state is as follows:

- `Status`: the status of an auction can move from its initial state, `INVALID` (under preparation), to `ACTIVE` (can receive bids) and `CLOSED` (no more bids accepted), such that `INVALID` < `ACTIVE` < `CLOSED`.
- `Winner`: The winner of the auction. It is either $\perp$, initially, or the bid with the highest amount. In case of multiple bids with same amount, the bid with the lowest id wins. It is ordered such that $\forall b \in Bids, \perp \leq b$.
- `Bids`: Set of bids placed (initially empty)
  - `BidId`: A unique identifier for each bid placed
  - `Placed`: A boolean flag to indicate whether the bid has been placed or not, ordered `TRUE` > `FALSE`. It is enabled once when the bid is placed. Once placed, a bid cannot be withdrawn.
  - `Amount`: An integer representing the amount of the bid; this cannot be modified once the bid is created.

Figure 1 illustrates how the auction state evolves over time; status, bids and winner are represented by a circle, rectangle and a star respectively. The application state is geo-replicated at data centres in Australia, Belgium and China.

We now specify the `merge` operation for an auction. We denote the receiving replica state σ = (status, winner, Bids); the received state is denoted σ' = (status′, winner′, Bids′).

```
merge((status,winner,Bids),
      (status′,winner′,Bids′)):
  status_new := max(status,status′)
  if winner′≠⊥ then winner_new := winner′
      else winner_new := winner
  ∀b, if b ∈ Bids ∩ Bids′ then
      Bids_new.b.placed := Bids.b.placed
                         ∨ Bids′.b.placed
    else if b ∈ Bids′ then
      Bids_new.b.placed := Bids′.b.placed
    else Bids_new.b.placed := Bids.b.placed
```

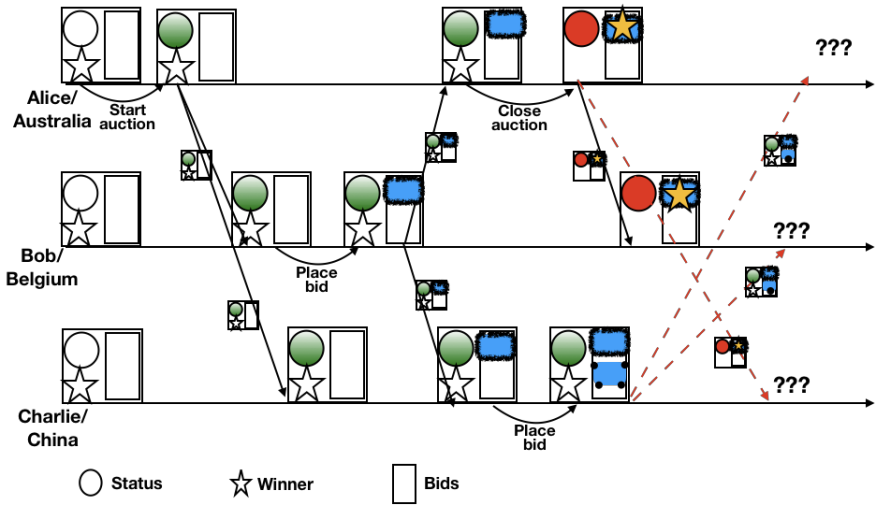

Figure 1: Evolution of state in an auction application

```
∀b, if b ∈ Bids then
    Bids_new.b.amount := Bids.b.amount
  else Bids_new.b.amount := Bids'.b.amount
```

In the absence of any extra synchronisation, it is possible to violate the invariant, as the following execution scenario illustrates. Alice from Australia starts an auction, by setting its status to `ACTIVE` (green in the figure) in the Australian replica. Henceforth, the auction can receive bids. The Australian replica sends its updated state to the other two replicas, which merge it into their own states. Now Bob in Belgium places a bid for $100 (blue). This update is sent to other replicas. Charlie, in China, sees the auction and Bob's bid. He updates the China state with a higher bid of $105 (dotted blue), which is sent to both other replicas. However, due to a network failure, the remote replicas do not receive this update (red dotted lines). Meanwhile Alice, unaware of Charlie's bid, closes the auction and declares Bob's bid as the winner. Later when the network heals, the updated states are sent and merged. The auction is new closed, and contains Bob's $100 bid and Charlie's $105 bid. Unfortunately, Bob's bid is the winner, violating the application invariant.

In the next section, we can discuss how to ensure invariants of applications build on top of the system model we described.

3 PROVING INVARIANTS

As explained earlier, each replica executes a sequence of state transitions, due either to a local update, or to a merge incorporating remote updates. Thus, concurrency can be observed only through merge.

Let us call *safe state* a replica state that satisfies the invariant. Assuming the current state is safe, any update (local or merge) must result in a safe state. To ensure this, every update is equipped with a precondition that disallows any unsafe execution.[1] Thus, a local update executes only when, at the origin replica, the current state is safe and its precondition currently holds. Similarly, merge executes only with two safe states that together satisfy a merge precondition.

Formally, an update u (an operation or a merge), mutates the local state σ, to a new state $\sigma_{new} = u(\sigma)$. To preserve the invariant, `Inv`, we require that

$$\sigma \in \mathrm{Pre}_u \implies u(\sigma) \in Inv$$

To illustrate local preconditions, consider an operation `close_auction(w: BidId)`, which sets auction status to `CLOSED` and the winner to `w`. The developer may have written a precondition such as `status = ACTIVE`, because closing an auction doesn't make sense otherwise. In order to ensure the invariant that the winner has the highest amount, one needs to strengthen it with the clause `is_highest(Bids, w)`, defined as $\forall \mathtt{b} \in \mathtt{Bids} : \mathtt{b.Amount} \leq \mathtt{w.Amount}$.

To illustrate merge precondition, consider a CRDT whose state is the pair of integers, $\sigma = (n, m) \in \mathbb{N} \times \mathbb{N}$. It has two operations, `incn` and `incm`, that respectively increment n or m by 1, and a merge function:

$$\mathtt{merge}(\sigma, \sigma') = (\max(n, n'), \max(m, m'))$$

We wish to maintain the invariant that their sum is no more than 10:

$$Inv \triangleq (n + m) \leq 10$$

[1] Technically, this is at least the weakest-precondition of the update for the invariant. It strengthens any *a priori* precondition that the developer may have set.

The precondition of incn is $\text{Pre}_{\texttt{incn}} \triangleq (n + m) \leq 9$; similarly for incm. Starting from a safe state (4, 5), two replicas may independently increment to states (5, 5) and (4, 6) respectively. Both are safe. However, merging them would violate the invariant. Therefore, merge(σ, σ') must have precondition

$$\text{Pre}_{\texttt{merge}} \triangleq \max(n, n') + \max(m, m') \leq 10$$

Since merge can happen at any time, it must be the case that its precondition is always true, i.e., it constitutes an additional invariant. Now our global invariant consists of two parts: first, the application invariant, and second, the precondition of merge. We can now state our proof rule informally as follows::[2]

State-Based Safety Rule. **Define the *precondition of merge* to be the weakest-precondition of merge, for the application invariant. The initial state must satisfy, and each local update or merge operation must preserve, the conjunction of: *(i)* the application invariant, and *(ii)* the precondition of merge.** □

In our Boogie notation, each operation can be verified as follows:

```
assume Inv ∧ Pre_merge ∧ Pre_op
call σ_new = op(σ)
assert Inv ∧ Pre_merge
```

The case of the merge function can be verified with the following condition:

```
assume Inv ∧ Inv′ ∧ Pre_merge
call σ_new = merge(σ, σ′)
assert Inv ∧ Pre_merge
```

Note that there are two copies of state, the unprimed local state of the replica applying the merge, and the primed state received from a remote replica. Inv′ denotes that σ' preserves the invariant Inv.

3.1 Applying the proof rule

Let us apply the proof methodology to the auction application. Its invariant is the following conjunction:

(1) A bid is placed only when status is ACTIVE.
(2) And: Once a bid is placed, its amount does not change.
(3) And: There is no winner until status is CLOSED.
(4) And: There is a single winner, the bid with the highest amount (breaking ties using the lowest identifier).

Computing the weakest-precondition of each update operation, for this invariant, is obvious. For instance, as discussed earlier, close_auction(w: BidId) gets precondition is_highest(Bids, w), because of Invariant Term 4 above.

[2] We omit the full formalisation and the proof of soundness for brevity.

Despite local updates to each replica preserving the invariant, Figure 1 showed that it is susceptible of being violated by merging. This is the case if Bob's $100 bid in Belgium wins, even though Charlie concurrently placed a $105 bid in China; this occurred because status became CLOSED in Belgium while still ACTIVE in China. The weakest-precondition of merge for Term 4 expresses that, if status in either states is CLOSED, the winner should be the bid with the highest amount in both the states. Therefore, merge(σ, σ') must have the following additional precondition:

```
status=CLOSED ⟹ is_highest(Bids, winner)
                ∧ is_highest(Bids′, winner)
∧ status′=CLOSED ⟹ is_highest(Bids, winner′)
                   ∧ is_highest(Bids′, winner′)
```

Furthermore, the code for merge uses Term 2, for which its weakest-precondition is as follows:

```
∀b ∈ Bids∩Bids′, Bids.b.amount = Bids′.b.amount
```

These two merge preconditions now strengthen the global invariant, in order to preserve safety in concurrent executions. Let us now consider how this strengthening impacts the local update operations. Since starting the auction doesn't modify any bids, this operation trivially preserves it. Placing a bid might violate it, if the auction is concurrently closed in some other replica; conversely, closing the auction could violate it, if a higher bid is concurrently placed in a remote replica. Thus, the auction application is safe when executed sequentially, but is unsafe when updates are concurrent. This indicates the specification has a bug, which we now proceed to fix.

3.2 Concurrency Control for Invariant Preservation

As we discussed earlier, the preconditions of operations and merge are strengthened in order to preserve the invariant. This provides a sequentially safe specification. An application must also preserve the precondition of merge in order to ensure concurrent safety. Violating this indicates the presence of a bug in the specification. In that case, the developer needs to strengthen the application by adding appropriate concurrency control mechanisms, ie., the operations that fail to preserve the precondition of merge might need to synchronise. The required concurrency control mechanisms are added as part of the state in our model. The modified application state is now composed of the CRDTs that represents the state and the concurrency control mechanism. Together, it behaves like a composition of state-based CRDTs. The whole state should now ensure the lattice conditions described in section 2.

Recall that in the auction example, placing bids and closing the auction were not preserving the precondition of merge.

This requires strengthening the specification by adding a concurrency control mechanism to restrict these operations. We can enforce them to be strictly sequential, thereby avoiding concurrency at all. But this will affect the availability of the application.

A concurrency control can be better designed with the workload characteristics in mind. For this particular application, we know that placing bids are very frequent operations than closing an auction. Hence we try to formulate a concurrency control like a readers-writer lock. In order to realise this we distribute tokens to each replica. As long as a replica has the token, it can allow placing bids. Closing the auction requires recalling the tokens from all replicas. This ensures that there are no concurrent bids placed and thus a winner can be declared, respecting the application safety.

The entire specification of the auction application can be seen in Figure 2. The shaded lines in blue indicate the effect of adding concurrency control to the state.

An alternative approach to our treatment of concurrency control could be to consider the *invariant as a resource* in the style of Concurrent Separation Logic [11]. In this case, access to the application state, described through a separation logic invariant, is guarded by a concurrency control mechanism (typically some form of a lock). However, this approach is tied to separation logic reasoning, where assertions act as resources, and allows one to distinguish local from global resources. We consider that this was not essential for the kind of proofs that we conduct, but it might be more promising when verifying client programs of our data types.

4 AUTOMATING THE VERIFICATION

In this section we present a tool to automate the verification of invariants as discussed in the previous sections. Our tool, called *Soteria* is based on the Boogie [2] verification framework. The input to Soteria is a specification of the application written in Boogie, an intermediate verification language.

A specification in Soteria will consist of the following parts:

- **State:** a declaration of the state. It can be a single CRDT or a composition of CRDTs.
- **Comparison function:** The programmer provides a comparison function (annotated with keyword `@gteq`) that determines the partial order on states.
- **Operations:** The programmer provides the implementation of the operations and their respective preconditions, $\mathrm{Pre}_{\mathrm{op}}$. Operations are encoded either imperatively as Boogie procedures or declaratively as postconditions.
- **Merge function:** The special `merge` operation is distinguished from the other operations (with annotation `@merge`). The programmer must provide a precondition to `merge` that is strong enough to prove the invariant.
- **Application Invariant:** The programmer provides the invariant (with keyword `@invariant`) to be verified by the tool as a Boogie assertion over the state.

In addition, Boogie often requires additional information such as: • User-defined data types, • Constants to declare special objects such as the origin replica me, or to bound the quantifiers, • Axioms for inductive functions over aggregate data structures, for instance, to compute the maximum of a set of values, • Loop invariants.

The specification of the auction application can be seen in Figure 2.

Verification

The verification of a specification is performed in multiple stages; in order:

(1) **Syntactic checks**: validates the specification for syntactical errors and checks whether the pre/post conditions are sound.
(2) **Compliance check**: checks whether the specification provides all the elements explained earlier.
(3) **Convergence check**: checks whether the specification respects the properties of a state-based CRDT, ie., each operation inflates the state and merge is the least upper bound.
(4) **Safety check**: verifies the safety of the application invariant, as discussed in section 3. This stage is divided further into two sub-stages:
   - *Sequential safety:* whether each individual operation (or merge) upholds the invariant. If not, the designer needs to strengthen the precondition of the corresponding operation (or merge)
   - *Concurrent safety:* whether every operation (and merge) upholds the precondition of merge. Note that, while this check relates to the concurrent behaviour of state-based CRDTs, the check itself is completely sequential, ie., it does not require reasoning about operations performed by other processes. This check ensures that the invariant remains safe during concurrent operation. If this check fails, the application needs stronger concurrency control.

Each check in Soteria [3] generates counterexamples when the verification fails. These counterexamples might guide the developer in debugging the specification according to the verification steps.

[3]The tool along with some sample specifications can be accessed at https://github.com/sreeja/soteria_tool.

```
Initial state:
  status = INVALID ∧ winner = ⊥
  ∧ ∄ b, Bids.b.placed
  ∧ ∀ r, Tokens.r = true

Comparison function:
  status₁ ≥ status₂ ∧ (winner₁≠⊥ ∨ winner₂=⊥)
  ∧∀b, (Bids₁.b.placed ∨ ¬Bids₂.b.placed)
  ∧ (∀r, ¬Tokens₁.r ∨ Tokens₂.r)

Invariant:
  ∀b, Bids.b.placed ⟹ status≥ACTIVE
      ∧ Bids.b.amount>0
  status≤ACTIVE ⟹ winner=⊥
  status=CLOSED ⟹ Bids.winner.placed
    ∧ is_highest(Bids, winner)
  status=CLOSED ⟹ ∀r,¬Tokens.r

{Pre_merge:
  status=CLOSED ⟹ is_highest(Bids, winner)
            ∧is_highest(Bids', winner)
  ∧ status'=CLOSED ⟹ is_highest(Bids, winner')
            ∧is_highest(Bids', winner')
  ∧ ∀b, Bids.b.amount = Bids'.b.amount
  ∧ ∀r, Tokens.r.me ⟹ Tokens'.r.me
  ∧ ∀r,b, (¬Tokens.r ∧ ¬Bids.b.placed)
              ⟹ ¬Bids'.b.placed
  ∧ ∀ r,b, (r≠me ∧ ¬Tokens.r ∧ ¬Bids.b.placed)
              ⟹ ¬Bids'.b.placed
  ∧ ∀ r,¬Tokens.r ⟹ winner'=winner ∨ winner'=⊥
  ∧ ∀ r, Tokens.r ⟹ winner=⊥ ∧ winner'=⊥}
merge((status, winner, Bids, Tokens),
        (status',winner',Bids',Tokens')):
  <merge of status, winner, Bids as in section 2>
  ∀r, Tokens_new.r := Tokens.r∧Tokens'.r

{Pre_start_auction: status = INVALID ∧ winner = ⊥
  ∧ ∀r, Tokens.r}
start_auction():
  status_new := ACTIVE
  winner_new := ⊥
{Pre_place_bid: ¬Bids.b_id.placed
  ∧ Bids.b_id.amount = value
  ∧ status = ACTIVE ∧ winner = ⊥
  ∧ Tokens.me}
place_bid(b_id, value):
  Bids_new.b_id.placed := true
  Bids_new.b_id.amount := value
{Pre_close_auction: status = ACTIVE ∧ winner = ⊥
  ∧ Bids.w.placed ∧ is_highest(Bids, w)
  ∧ ∀r, ¬Tokens.r}
close_auction(w):
  status_new := CLOSED
  winner_new := w
```

Figure 2: Specification of auction application

5 RELATED WORK

Several works have concentrated on the formalisation and specification of eventually consistent systems [3, 4, 14]. A number of works concentrate on the specification and correct implementation of CRDTs [5, 7]. Our work also verifies the CRDT (lattice) conditions, but additionally verifies an arbitrary application invariant over a replica state.

Gotsman et al. [6] provides a proof methodology for proving invariants of CRDTs that propagate operations. The associated tools [8, 10] performs the check using an SMT solver as the backend and Nair and Shapiro [9] discusses some concurrency control suggestions by using the counterexamples generated by the failed proofs. Gotsman et al. [6] assume that the underlying network ensures causal consistency, and their methodology requires reasoning about concurrent behaviours. This requires checks for each pair of operations in the application (reflected as stability verification conditions). Gotsman et al. [6] uses an abstract notion of *tokens* as concurrency control mechanisms. The operations acquire tokens in order to preserve the application invariant.

In contrast, Soteria focuses on state-based CRDTs. We check convergence by verifying the lattice conditions of section 2 and that because of the rules shown in section 3, we can reduce the problem of verifying the invariant to sequential proof obligations. This is reflected by the fact that all of our proofs are standard pre/post conditions checks using the Boogie framework. Boogie framework. In contrast with Gotsman et al. [6], Soteria includes concrete specification of concurrency control as part of the application state.

To the best of our knowledge, ours is the first attempt in automated verification of invariants of state-based CRDTs.

6 CONCLUSION

We have presented a proof methodology to verify invariants of state-based CRDT implementations guaranteeing: (1) that the implementation satisfies the lattice conditions of state-based CRDTs [1], and (2) that the implementation satisfies programmer provided invariant reducing the problem to checking that each operation of the data type satisfies a precondition of the `merge` function of the state.

We implemented Soteria, a tool sitting on top of the Boogie verification framework, to specify the implementation, its invariant and validate it.

In future work, we plan to automate concurrency control synthesis. The synthesised concurrency control can be analysed and adjusted dynamically to minimise the cost of synchronisation. Another direction for future work can be to decouple the update propagation mechanism of CRDT from the proof rule resulting in a generic proof rule to verify distributed systems.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments which helped in improving this paper. This research is supported in part by the RainbowFS project (*Agence Nationale de la Recherche*, France, number ANR-16-CE25-0013-01) and by European H2020 project 732 505 LightKone (2017–2020).

REFERENCES

[1] C. Baquero, P. S. Almeida, A. Cunha, and C. Ferreira. Composition in state-based replicated data types. *Bulletin of the EATCS*, 123, 2017. URL http://eatcs.org/beatcs/index.php/beatcs/article/view/507.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-36749-7, 978-3-540-36749-9. doi: 10.1007/11804192_17. URL http://dx.doi.org/10.1007/11804192_17.

[3] S. Burckhardt. *Principles of Eventual Consistency*, volume 1 of *Foundations and Trends in Programming Languages*. Now Publishers, Oct. 2014. doi: 10.1561/2500000011. URL http://research.microsoft.com/pubs/230852/final-printversion-10-5-14.pdf.

[4] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. pages 271–284, San Diego, CA, USA, Jan. 2014. doi: 10.1145/2535838.2535848. URL http://doi.acm.org/10.1145/2535838.2535848.

[5] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. A framework for establishing strong eventual consistency for conflict-free replicated datatypes. *Archive of Formal Proofs*, 2017, 2017. URL https://www.isa-afp.org/entries/CRDT.shtml.

[6] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'Cause I'm Strong Enough: Reasoning about consistency choices in distributed systems. pages 371–384, St. Petersburg, FL, USA, 2016. doi: 10.1145/2837614.2837625. URL http://dx.doi.org/10.1145/2837614.2837625.

[7] R. Jagadeesan and J. Riely. Eventual consistency for CRDTs. In A. Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 968–995. Springer, 2018. ISBN 978-3-319-89883-4. doi: 10.1007/978-3-319-89884-1_34. URL https://doi.org/10.1007/978-3-319-89884-1_34.

[8] G. Marcelino, V. Balegas, and C. Ferreira. Bringing hybrid consistency closer to programmers. PaPoC '17, pages 6:1–6:4, Belgrade, Serbia, 2017. ACM. doi: 10.1145/3064889.3064896. URL http://doi.acm.org/10.1145/3064889.3064896.

[9] S. Nair and M. Shapiro. Improving the "Correct Eventual Consistency" tool. Rapport de recherche RR-9191, Paris, France, July 2018. URL https://hal.inria.fr/hal-01832888.

[10] M. Najafzadeh, A. Gotsman, H. Yang, C. Ferreira, and M. Shapiro. The CISE tool: Proving weakly-consistent applications correct. EuroSys 2016 workshops, London, UK, Apr. 2016. doi: 10.1145/2911151.2911160. URL http://dx.doi.org/10.1145/2911151.2911160.

[11] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. doi: 10.1016/j.tcs.2006.12.035. URL https://doi.org/10.1016/j.tcs.2006.12.035.

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, Jan. 2011.

[13] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. volume 6976, pages 386–400, Grenoble, France, Oct. 2011. doi: 10.1007/978-3-642-24550-3_29. URL https://doi.org/10.1007/978-3-642-24550-3_29.

[14] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. PLDI '15, pages 413–424, Portland, OR, USA, 2015. doi: 10.1145/2737924.2737981. URL http://doi.acm.org/10.1145/2737924.2737981.