Formal Specification and Verification of CRDTs

Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany
{p_zeller,bieniusa,poetzsch}@cs.uni-kl.de

Abstract. *Convergent Replicated Data Types* (CRDTs) can be used as basic building blocks for storing and managing replicated data in a distributed system. They provide high availability and performance, and they guarantee eventual consistency. In this paper, we develop a formal framework for the analysis and verification of CRDTs. We investigate and compare the three currently used specification techniques for CRDTs and formalize them based on an abstract model for managing replicated data in distributed systems. We show how CRDT implementations can be expressed in our framework and present a general strategy for verifying CRDTs. Finally, we report on our experiences in using the framework for the verification of important existing CRDT implementations. The framework and the proofs were developed within the interactive theorem prover Isabelle/HOL.

Keywords: CRDT, formal verification, eventual consistency

1 Introduction

Global computing systems and worldwide applications often require *data replication*, that is, the data is not only stored at one computing node, but at several nodes. There are a number of reasons for replication. To realize reliable services, high availability and fault-tolerance might be important. A centralized storage system might not provide enough throughput for allowing millions of users to access the data. Furthermore, systems often serve clients from different regions in the world; Geo-replication helps to keep the latency low. Last but not least, mobile clients might not be connected all the time, but should provide certain offline functionality using a local store that is synchronized with the global state when the connection is reestablished.

The CAP theorem[6] tells us that distributed systems cannot guarantee high availability, partition tolerance, and strong consistency at the same time. In this paper, we investigate techniques that aim at high availability and partition tolerance by providing a weaker form of consistency, called *eventual consistency*[14,15,11,5].

In an eventually consistent system, the different replicas do not have to provide the same view of the data at all times. Operations manipulating the data are first applied to a subset of the replicas. At some suitable later point in time, the updates are communicated to the other replicas. Only after all updates are delivered to all replicas the data has to be consistent again. This means in particular that the replicas have to be able to merge concurrent updates into a consistent view. It usually depends on the application how a merge operation should behave. The idea of replicated data types is to package a behavior given by operations and a way to handle concurrent updates, so that they can be reused in many situations.

Convergent replicated data types (CRDTs)[12] are a special class of replicated data types where concurrent updates are handled by merging the resulting states. The basic setting is that there is a number of replicas of the data type. Operations are executed on a single replica and the different replicas conceptually exchange their whole states with each other according to some protocol. The state of a replica is also called the *payload*. To guarantee convergence, there needs to be a partial order on the payloads and a merge operation computing the least upper bound, such that the payloads form a semilattice. All operations which change the payload have to increase the payload with respect to the partial order. This setup ensures that replicas which have seen the same set of updates also have the same state and thus return the same value. This property is called Strong Eventual Consistency (SEC)[13].

A simple example of a CRDT is a counter. A counter provides update operations to increment or decrement the value and a query function to read the value. It is usually implemented by keeping the number of increments and the number of decrements for each replica in a map from replica-IDs to positive integers. Each replica only increments its own increment- and decrement-counts and states are merged by taking the component-wise maximum. It is easy to prove that this computes a least upper bound. The value of the counter can be determined by summing up all increment counts in the map and subtracting the sum of the decrement counts. Separating increment- and decrement-counts ensures, that the updates are increasing operations. Having a count for every replica instead of a single counter ensures that no updates are lost due to concurrent writes.

Contributions. In general, CRDT implementations can be quite sophisticated and complex, in particular in the way they handle concurrent updates. Therefore it is important to provide high-level behavioral specifications for the users and techniques to verify CRDT implementations w.r.t. their specifications. Towards this challenge, the paper makes the following contributions:

- A framework in Isabelle/HOL [10] that supports the formal verification of CRDTs. The framework in particular provides a system model that is parametric w.r.t. the CRDT to be analyzed and support for verification.
- We analyzed, clarified, and formalized different specification and abstract implementation techniques for CRDTs (Sections 3, 4 and 6).
- We successfully verified a number of CRDTs described in the literature with our framework.

We present the system model in Section 2; describe our specification technique in Section 3, the implementation technique in Section 4 and the verification as-

System state: $version :: replicaId \rightarrow \mathbb{N}$ $payloadHistory :: (version \times payload)\ set$ $systemState :: (replicaId \rightarrow payload) \times (replicaId \rightarrow version) \times payloadHistory$
Operations and traces: $Operation := Update(replicaId, args) \quad \mid \quad Merge(replicaId, version, payload)$ $Trace := Trace;\ Operation \quad \mid \quad []$
Operational semantics: $s_{init} = (\lambda r.\ init_{crdt},\ \lambda r.\ v_0,\ \emptyset) \qquad \frac{}{s \xrightarrow{[]} s} \qquad \frac{s \xrightarrow{as} s' \qquad s' \xrightarrow{a} s''}{s \xrightarrow{as;a} s''}$ $(update) \qquad \frac{v' = vs(r)(r{+}{=}1) \qquad pl' = update_{crdt}(a, r, pls(r))}{(pls,vs,ph) \xrightarrow{Update(r,a)} (pls(r := pl'),\ vs(r := v'),\ ph \cup \{(v', pl')\})}$ $(merge) \qquad \frac{(v, pl) \in ph \qquad v' = vs(r) \sqcup v \qquad pl' = merge_{crdt}(pls(r), pl)}{(pls,vs,ph) \xrightarrow{Merge(r,v,pl)} (pls(r := pl'),\ vs(r := v'),\ ph \cup \{(v', pl')\})}$

Table 1. System Model

pects in Section 5. In Section 6 we discuss an alternative specification technique. Finally, we consider related work, summarize the results, and give an outlook on future work in Sections 7 and 8

2 System model

We developed a formal system model that supports an arbitrary but fixed number of replicas and is parametric in the CRDT to be analyzed, i.e., it can be instantiated with different CRDTs. A CRDT is parameterized by four type parameters and described by four components:

- **pl**, the type of the payload (i.e. the state at each replica)
- **ua**, the type for the update arguments (a sum type of all update operations)
- **qa**, the type for the query arguments (a sum type of all query operations)
- **r**, the sum type for the possible return values of the queries

- **init** :: pl , the initial payload for all replicas
- **update** :: $ua \Rightarrow replicaId \Rightarrow pl \Rightarrow pl$, the function expressing how an update operation modifies the payload at a given replica, where *replicaId* denotes the node on which the update is performed first
- **merge** :: $pl \Rightarrow pl \Rightarrow pl$, the function merging payloads

- **query** :: $qa \Rightarrow pl \Rightarrow r$, the function expressing the results of querying a payload

Given a CRDT ($init_{crdt}$, $update_{crdt}$, $merge_{crdt}$, $query_{crdt}$), the system model describes the labeled transition relation $s \xrightarrow{tr} s'$ expressing that executing trace tr in state s leads to state s' (cf. Table 1) where a trace is a sequence of operations and an operation is either the application of an update or a merge (queries need not be considered, as they do not modify the state). The state of the system consists of three components:

- For each replica r, its current payload.
- For each replica r, its current version vector[8]. The *version vector* or short *version* is a mapping from replica-IDs to natural numbers. If the version of r has m as entry for key k, the payload of r has merged the first m operations applied to replica k into its payload.
- The set of all version-payload pairs that have been seen during execution so far. This set is called the *payload history* and is used to make sure that a merge operation can only be applied to version-payload pairs that appeared earlier in the execution.

Initially, the payload of each replica is $init_{crdt}$, the version vector of each replica is the all-zero vector, and the payload history is the empty set. There are two kind of transition steps:

- An *update* operation $Update(r, a)$ applies an update function of the CRDT (determined by the arguments a) to the current payload of r and modifies the state accordingly; in particular, the rth component of the version vector of r is incremented by one.
- A *merge* operation $Merge(r, v, pl)$ is executed as follows: The new version v' is calculated by taking the least upper bound of the old version vector $vs(r)$ and the merged version vector v. Similarly, the new payload pl' is the least upper bound operation on payloads, which is specific to the respective CRDT, and implemented by the $merge_{crdt}$ function.

Discussion. The system model focuses on simple operational behavior. Other aspects, such as timestamps, were intentionally left out because they would make the system model more complicated than needed for most CRDTs. Only a few CRDTs like the Last-Writer-Wins-Register depend on timestamps. Also, often timestamps are only used to provide some total order on operations, and lexicographic ordering of the version vectors also suffices to provide such an order.

3 Specification

In this section we present and formalize a technique for specifying the behavior of CRDTs based on the system model presented in the previous section.

A specification should tell users the result value of any query operation, when a trace of operations performed in the system is given. A trace gives a total order on the operations performed in the system, but for operations performed independently on different replicas this order does not influence the result. Therefore, we would like to abstract from this total order. Furthermore, the traces include explicit merge operations, but from a user's perspective this merge operations are implicit. Thus, it should not be important, how updates were delivered to a replica. The result of an operation should only depend on those operations that are visible when the operation is about to be performed. Hence, the trace can be actually deconstructed into a partially ordered set of update operations, where the partial order is the visibility relation, which we denote by $\prec$ in the following.

The specification technique that we formalize here supports the sketched abstractions and explains the result of an operation only depending on the visible update history. It follows the ideas from Bouajjani et al.[4], and Burckhardt et al.[5], but is specifically tailored to CRDTs, allowing for some simplifications. In the case of CRDTs, the visibility relation is a partial order. All operations at one replica are ordered by time and a merge makes all operations, which are visible to the source of the merge, visible at the destination of the merge.

Formalization. In our formalization we represent the visibility relation using version vectors. The advantage of this is that they are easy to handle in the operational semantics and also when working with Isabelle/HOL. The properties of a partial order like transitivity and antisymmetry are already given by the structure and do not have to be specified additionally. The version vector at each replica can be directly derived from a given trace. It also uniquely identifies every operation, and we can encode the whole history of updates with the visibility relation as a set of update operations, represented by $(\mathit{version}, \mathit{replicaId}, \mathit{args})$ triples. We call this structure the **update history** and denote it by H in the following. The visibility relation $\prec$ on update operations is simply derived from the order on the version vectors.

A specification is formalized as a function *spec*, which takes the update history visible at a given replica and the arguments of a query and returns the result of the query. A specification is **valid** if for every reachable state and all queries a, the specification yields the same result as the application of the query to the current state:

$$\forall_{tr,pls,vs,a,r}.\ s_{init} \xrightarrow{tr} (pls, vs, \_) \ \Rightarrow\ \mathit{spec}(H(tr, vs(r)), a) = \mathit{query}_{crdt}(a, \mathit{pls}(r))$$

Here, the term $H(tr, vs(r))$ calculates the update history from the trace tr while only taking the operations before the version $vs(r)$ into account.

Examples. Table 2 shows specifications for several CRDTs from the literature[12]. The **Counter** is a data type providing an update-operation to increment or decrement the value of the Counter and a query-operation $\mathit{Get}()$ which returns the current value of the counter. The argument to the update is a single integer value. We specify the return value of a $\mathit{Get}()$ operation on a counter by

Counter:	$\mathit{spec}(H, \mathit{Get}()) = \sum_{e \in H.} \mathit{args}(e)$
Grow-Set:	$\mathit{spec}(H, \mathit{Contains}(x)) = \exists_{e \in H.} \mathit{args}(e) = \mathit{Add}(x)$
Two-Phase-Set:	$\mathit{spec}(H, \mathit{Contains}(x)) = \exists_{e \in H.} \mathit{args}(e) = \mathit{Add}(x) \wedge$ $\neg(\exists e \in H.\ \mathit{args}(e) = \mathit{Remove}(x))$
Two-Phase-Set (guarded remove):	$\mathit{spec}(H, \mathit{Contains}(x)) = \exists_{e \in H.} \mathit{args}(e) = \mathit{Add}(x) \wedge$ $\neg(\exists_{e \in H.} \mathit{args}(e) = \mathit{Remove}(x) \wedge (\exists_{f \in H.} \mathit{args}(f) = \mathit{Add}(x) \wedge f \prec e))$
Observed-Remove-Set:	$\mathit{spec}(H, \mathit{Contains}(x)) = \exists_{a \in H.} \mathit{args}(a) = \mathit{Add}(x) \wedge$ $\neg(\exists_{r \in H.} a \prec r \wedge \mathit{args}(r) = \mathit{Remove}(x))$
Multi-Value-Register:	$\mathit{spec}(H, \mathit{Get}()) = \{x \mid \exists_{e \in H.} \mathit{args}(e) = \mathit{Set}(x) \wedge \neg(\exists_{f \in H.} e \prec f)\}$

Table 2. Specifications of CRDTs

taking all update operations e from the update history H and then summing up their update arguments. The **Grow-Set** is a set which only allows adding elements. An element is in the set, when there exists an operation adding the element. The **Two-Phase-Set** also allows to remove elements from the set, with the limitation that an element cannot be added again once it was removed. An element is in the set, if there is an operation adding the element and no operation removing it. This specification allows removing an element before it was added to the set, which might not be desired. The **Two-Phase-Set with the guarded remove operation**, ignores remove operations, when the respective element is not yet in the set. For this data type, an element is in the set, when there exists an operation adding the element, and there is no operation which removes the element and which happened after an add operation of the same element. The **Observed-Remove-Set** is a set, where an element can be added and removed arbitrarily often. A remove operation only affects the add operations which have been observed, i.e. which happened before the remove operation. We specify that the query $\mathit{Contains}(x)$ returns true, if and only if there exists an update operation a adding x to the set and there exists no update operation r which happened after a and removes x from the set. The final example is the **Multi-Value-Register**. It has a $\mathit{Set}(x)$ operation to set the register to a single value. The $\mathit{Get}()$ query returns a set containing the values of the last concurrent *Set* operations. More precisely, it returns all values x so that there exists an operation $\mathit{Set}(x)$, for which no later update operation exists.

Properties and Discussion. It is not possible to describe non-converging data types with this technique. Since the specified return value of a query only depends on the visible update history and the arguments, two replicas which have seen the same set of updates will also return the same result.

One problem with this specification technique is that in general a specification can reference all operations from the past. The state of a replica is basically determined by the complete update history, which can be quite large and not very abstract. Therefore it is hard to reason about the effects of operations when programming with the data types or when verifying properties about systems using them, where one usually wants to reason about the effect of a single method in a modular way.

Also, the example of the Two-Phase-Set with a guarded remove operation shows that small changes to the behavior of one update operation can make the whole specification more complex. We would like such a change to only affect the specification of the remove operation. Thus, the question is whether we can specify CRDTs avoiding these problems. We present and discuss an alternative specification technique in Section 6. It is also possible to use abstract implementations as a form of specification, as detailed in the next section.

4 Implementations

To implement a CRDT in our framework one has to define the type of the payload and the four fields of the CRDT record ($init_{crdt}$, $update_{crdt}$, $merge_{crdt}$, $query_{crdt}$) as defined in the system model. Technically, the implementation can be any Isabelle function with a matching type.

To keep the examples short, we introduced a *uid*-function, which generates a new unique identifier. This can easily be implemented in our system model by adding a counter to the payload of the data type. A unique identifier can then be obtained by taking a pair (*replicaId, counter*) and incrementing the counter. It is also possible to use the version vector as a unique identifier for an update operation, which can make the verification easier, as the payload is then directly related to the update history.

Abstract Counter:	$s :: (id \times int)set = \{\}$ $\textbf{update}(x, r, s) = s \cup \{(uid(), x)\}$ $\textbf{merge}(s, s') = s \cup s'$ $\textbf{query}(Get(), s) = \sum_{(id,x)\in s} x$
Optimized Counter:	$s :: (replicaId \to int) \times (replicaId \to int) = (\lambda r.\ 0, \lambda r.\ 0)$ $\textbf{update}(x, r, (p, n)) = \textbf{if}\ x \geq 0\ \textbf{then}\ (p(r := p(r) + x), n)$ $\textbf{else}\ (p, n(r := n(r) - x))$ $\textbf{merge}((p, n), (p', n')) = (\lambda r.\ max(p(r), p'(r)), \lambda r.\ max(n(r), n'(r)))$ $\textbf{query}(Get(), (p, n)) = \sum_r p(r) - \sum_r n(r)$

Table 3. Abstract and optimized implementation of a Counter CRDT

Table 3 shows two implementations of the Counter CRDT. The first implementation is an abstract one, in which the payload is a set of all update arguments tagged with a unique identifier. The query can then be answered by summing up all the update arguments in the set. This implementation is very inefficient, but easy to understand. The second implementation is closer to Counter implementations found in real systems. Here, the payload consists of two mappings from replicaIds to integers. The first map (p) sums up all the positive update operations per replica and the second map (n) sums up all the negative ones. While this is still one of the easier CRDTs, it is not trivial to see, that the optimized implementation is valid with respect to its specification. We will come back to this example in Section 5 and show how the correctness can be proven using our framework.

Grow-Set:	$s ::' a\ set = \{\}$ $\mathbf{update}(Add(x), r, s) = s \cup \{x\}$ $\mathbf{merge}(s, s') = s \cup s'$ $\mathbf{query}(Contains(x), s) = x \in s$
Two-Phase-Set:	$s ::' a \Rightarrow \{init = 0, in = 1, out = 2\} = (\lambda x.\ init)$ $\mathbf{update}(Add(x), r, s) = (\mathbf{if}\ s(x) = init\ \mathbf{then}\ s(x := in)\ \mathbf{else}\ s)$ $\mathbf{update}(Rem(x), r, s) = s(x := out)$ $\mathbf{merge}(s, s') = (\lambda x.\ max(s(x), s'(x)))$ $\mathbf{query}(Contains(x), s) = (s(x) = in)$
Two-Phase-Set (guarded remove):	Same as above, but with different remove operation: $\mathbf{update}(Rem(x), r, s) = (\mathbf{if}\ s(x) = in\ \mathbf{then}\ s(x := out)\ \mathbf{else}\ s)$
Observed-Remove-Set:	$s.e :: (id \times' a) set = \{\}, s.t :: id\, set = \{\}$ $\mathbf{update}(Add(x), r, s) = s(e := s.e \cup \{(uid(), x)\})$ $\mathbf{update}(Rem(x), r, s) = s(t := s.t \cup \{id \mid \exists_x.(id, x) \in s.e\})$ $\mathbf{merge}(s, s') = (e = s.e \cup s'.e,\ t = s.t \cup s'.t)$ $\mathbf{query}(Contains(x), s) = \exists_{id}.(x, id) \in s.e \wedge id \notin s.t$
Multi-Value-Register:	$s.e :: (id \times' a) set = \{\}, s.t :: id\, set = \{\}$ $\mathbf{update}(Set(x), r, s) = s(e := \{(uid(), x)\},$ $t := s.t \cup \{id \mid \exists_x.(id, x) \in s.e\})$ $\mathbf{merge}(s, s') = (e = s.e \cup s'.e,\ t = s.t \cup s'.t)$ $\mathbf{query}(Get(), s) = \{x \mid \exists_{id}.(x, id) \in s.e \wedge id \notin s.t\}$

Table 4. State-based specifications of CRDTs

Table 4 shows abstract implementations of the other CRDTs introduced in Section 3. The Grow-Set can be implemented using a normal set where the merge is simply the union of two sets. The payload of the Two-Phase-Set can be described by assigning one out of three possible states to each element. In a new, empty set all elements are in the *init* state. Once an element is added, it goes to the *in* state, and when it is removed it goes to the *out* state. The merge simply takes the maximum state for each element with respect to the order $init < in < out$. The last two CRDTs in Table 4 have a very similar implementation. This is not very surprising, as the *Set* operation of the register is basically an operation, that first removes all elements from the set and then adds a single new element. In both cases the payload consists of a set of elements tagged with an unique identifier and a set of tombstones, that contains all unique identifiers of the removed elements. The unique identifier makes sure, that the remove operation only affects previous add-operations, as it is demanded by the specification.

Relation to Specifications. All CRDT implementations can be specified by the specification technique described in Section 3, when the merge operation computes a least upper bound with respect to a semilattice and the update operations are increasing with respect to the order on the semilattice. This is possible, as the state can be reconstructed from a given update history, when the implementation is known. Because the merge operation of a CRDT computes a least upper bound, it is straight-forward to extend it to a merge function, which merges a set of payloads. Then a function to calculate the state can be defined recursively in terms of previous versions: If there is an update at the top of the history, apply the update operation to the merge of all previous versions. If there is no update operation at the top, just take the merge of all previous versions. This terminates, when the set of all previous versions only consists of the initial state.

The converse is also true: each specification given in this form describes a CRDT. A specification can be turned into an inefficient implementation by storing the visible update history in the payload of the data type. The update history is just a growing set which can be merged using the union of sets, thus forming a semilattice.

5 Verification

In our work on verification of CRDTs we considered two properties. The first property is the convergence of a CRDT, meaning that two replicas, which have received the same set of updates, should return the same results for any given query. This property is common to all CRDTs and does not require any further specification. The second property is the behavior of a CRDT, i.e. we want to prove, that a specification as presented in Section 3 is valid for a given implementation. As we discussed earlier, this is a strictly stronger property, but it requires a specification for each data type.

Section 5.1 covers the verification of the convergence property, in Section 5.2 we present a technique for verifying the behavior, and in Section 5.3 we evaluate our experience in using Isabelle/HOL for the verification of CRDTs with the presented techniques.

5.1 Verification of Convergence

The convergence property can be verified by proving that the payload of the CRDT forms a semilattice, such that the merge-operation computes a least upper bound and the update-operations increase the payload with respect to the order on the semilattice.[12]

(refl)	$\mathit{Inv}(H, \mathit{pl}) \Rightarrow \mathit{pl} \leq_{crdt} \mathit{pl}$
(trans)	$\mathit{Inv}(H_1, \mathit{pl}_1) \wedge \mathit{Inv}(H_2, \mathit{pl}_2) \wedge \mathit{Inv}(H_3, \mathit{pl}_3) \wedge$ $\mathit{pl}_1 \leq_{crdt} \mathit{pl}_2 \wedge \mathit{pl}_2 \leq_{crdt} \mathit{pl}_3 \Rightarrow \mathit{pl}_1 \leq_{crdt} \mathit{pl}_3$
(antisym)	$\mathit{Inv}(H_1, \mathit{pl}_1) \wedge \mathit{Inv}(H_2, \mathit{pl}_2) \wedge \mathit{pl}_1 \leq_{crdt} \mathit{pl}_2 \leq_{crdt} \mathit{pl}1 \Rightarrow \mathit{pl}_1 = \mathit{pl}_2$
(commute)	$\mathit{Inv}(H_1, \mathit{pl}_1) \wedge \mathit{Inv}(H_2, \mathit{pl}_2) \Rightarrow \mathit{merge}_{crdt}(\mathit{pl}_1, \mathit{pl}_2) = \mathit{merge}_{crdt}(\mathit{pl}_2, \mathit{pl}_1)$
(upper bound)	$\mathit{Inv}(H_1, \mathit{pl}_1) \wedge \mathit{Inv}(H_2, \mathit{pl}_2) \Rightarrow \mathit{pl}_1 \leq_{crdt} \mathit{merge}_{crdt}(\mathit{pl}_1, \mathit{pl}_2)$
(least upper bound)	$\mathit{Inv}(H_1, \mathit{pl}_1) \wedge \mathit{Inv}(H_2, \mathit{pl}_2) \wedge \mathit{Inv}(H_3, \mathit{pl}_3) \wedge$ $\mathit{pl}_1 \leq_{crdt} \mathit{pl}_3 \wedge \mathit{pl}_2 \leq_{crdt} \mathit{pl}_3 \Rightarrow \mathit{merge}_{crdt}(\mathit{pl}_1, \mathit{pl}_2) \leq \mathit{pl}_3$
(monotonic updates)	$\mathit{Inv}(H, \mathit{pl}) \Rightarrow \mathit{pl} \leq_{crdt} \mathit{update}_{crdt}(args, r, \mathit{pl})$

Table 5. Verifying convergence of CRDTs

However, only very simple data types form a semilattice in the classical mathematical sense. Often the semilattice properties only hold for a subset of the payloads. For some states which are theoretically representable by the payload type, but are never reached in an actual execution, the semilattice properties sometimes do not hold. In theory it could even be the case that there are two reachable states for which the merge operation does not yield the correct result, but where the two states can never be reached in the same execution. However, for the examples we considered it was always sufficient to restrict the payload to exclude some of the unreachable states. Technically, this was done by giving an invariant *Inv* over the update history H and the payload *pl*. The same type of invariant will also be used for the verification of behavioral properties in the next section. In the examples we considered, it was not necessary to use the update history H in the invariant. An overview of the sufficient conditions for convergence, which we used, is given in Table 5. The order on the payloads is denoted by $\leq_{crdt}$.

In order to verify these conditions for the Counter CRDT, we have to define the order on the payloads. Here we can simply compare the mappings for each replicaId: $(p, n) \leq (p', n') \leftrightarrow \forall_r \; p(r) \leq p'(r) \wedge n(r) \leq n'(r)$. An invariant is not required for this example and the proof of the semilattice conditions can be done mainly automatically by Isabelle/HOL. In fact, for all the easier examples, it was possible to do the majority of the proofs with the automated methods provided by Isabelle/HOL (sledgehammer, auto, ...).

5.2 Verification of Behavior

For the verification of behavioral properties we have developed a small framework, which simplifies the verification and provides two general strategies for verifying a CRDT. The first strategy basically is an induction over the traces. The idea of the second strategy is to show that a CRDT behaves equivalently to

The invariant must hold initially:

$$Inv(\{\}, initial_{crdt})$$

Merges must preserve the invariant:

$$\forall_{H_1,H_2,pl_1,pl_2}\ valid(H_1) \land valid(H_2) \land Inv(H_1, pl_1) \land Inv(H_2, pl_2) \\ \land\ consistent(H_1, H_2) \Rightarrow Inv(H_1 \cup H_2, merge_{crdt}(pl_1, pl_2))$$

Updates must preserve the invariant:

$$\forall_{H,pl,r,v,args}\ valid(H) \land Inv(H, pl) \land v = sup_v(H) \\ \Rightarrow Inv(H \cup \{(v(r := v(r) + 1), r, args)\}, update_{crdt}(args, r, pl))$$

The invariant must imply the specification:

$$\forall_{H,pl,qa}\ valid(H) \land Inv(H, pl) \Rightarrow query_{crdt}(qa, pl) = spec(H, qa)$$

Table 6. Verifying behavior of CRDTs

another CRDT which has already been verified. In this paper we only present the first strategy.

When using this strategy, one has to provide an invariant between the payloads and the visible update history. It then has to be shown that the invariant implies the specification, that the invariant holds for the initial payload with the empty update history, and that the invariant is maintained by update- and merge-operations. Table 6 shows the four subgoals.

For both operations our framework provides basic properties about **valid update histories** (predicate *valid*), which hold for all CRDTs. Because we used version vectors for representing the visibility relation, it is not necessary to specify the partial order properties of the relation, but instead it is necessary to specify constraints for the version vectors. The most important property is that the updates on one replica form a total order where the local component of the version vector is always increased by one and the other components increase monotonically. Other properties describe the causality between version vectors in more detail and can be found in [16].

In the case of an update operation one has to show that the invariant is maintained when adding a new update to the top of the update history, meaning that all other updates are visible to the new update. In a real execution this is usually not the case, but the framework can still do this abstraction step, because updates which are not visible do not influence the new update.

In the case of a merge operation one can assume that the invariant holds for two *compatible* update histories with two corresponding payloads, and then has to show that the invariant also holds for the union of the two update histories with the merged payload. Two update histories are *compatible*, when for each replica, the sequence of updates on that replica in one update history is a prefix of the sequence of updates in the other update history.

To verify the counter example we used the following invariant: $Inv(H, (p, n)) \leftrightarrow \forall_r\ p(r) = \sum\{x | \exists_v\ (v, r, x) \in H \land x \geq 0\} \land n(r) = \sum\{-x | \exists_v\ (v, r, x) \in H \land x < 0\}$.

For proving, that a merge-operation preserves the invariant, we have to use the property of *compatible* histories. From this property we get, that for any replica r, we either have $\{x|\exists_v\ (v,r,x) \in H \wedge x \geq 0\} \subseteq \{x|\exists_v\ (v,r,x) \in H' \wedge x \geq 0\}$ or the other way around. This combined with the fact, that all elements are positive, ensures that calculating the maximum yields the correct result. The other parts of the verification, namely update-operations, the initial state and the connection between the invariant and the specification, are rather trivial on paper, whereas in Isabelle the latter requires some work in transforming the sums.

5.3 Evaluation

We used the interactive theorem prover Isabelle/HOL[10] for the verification of several important CRDTs. To this end, we manually translated the pseudocode implementations from the literature[12,2] into Isabelle functions, and then verified those implementations. The verified CRDTs are the Increment-Only-Counter, PN-Counter, Grow-Set, Two-Phase-Set, a simple and an optimized OR-Set implementation, and a Multi-Value-Register. The theory files are available on GitHub[1].

For the simple data types, the semilattice properties were mostly automatically proved by Isabelle/HOL. For the more complicated data types, like the optimized OR-Set or the similarly implemented MV-register, a suitable invariant had to be found and verified first, which required more manual work in the proofs.

Verifying the behavior of the data types was a more difficult task. Finding a suitable invariant has to be done manually, and the invariant has to be chosen such that it is easy to work with it in Isabelle/HOL. Proving that the invariant is maintained also requires many manual steps, as it usually requires some data transformations which can not be handled automatically by Isabelle/HOL.

We found two small problems, while verifying the CRDTs mentioned above:

- When trying to verify an implementation of the OR-set based on figure 2 in [3], we found a small problem in our translation of this implementation to Isabelle. In the original description the remove-operation computes the set R of entries to be removed with the formula $R = \{(e,n)|\exists n : (e,n) \in E\}$. When this expression is translated to Isabelle code in a direct way, one obtains an equation like $R = \{(e,n).\ \exists n.(e,n) \in E\}$. Then R will always contain all possible entries, because in Isabelle e and n are new variables, and e does not reference the parameter of the function as intended. This problem can be easily fixed, and was not a real issue in the original description, but rather a mistake made in the translation to Isabelle, which happened because of the different semantics of the pseudo-code used in the original description and Isabelle.

[1] `https://github.com/SyncFree/isabelle_crdt_verification`

- We discovered another small problem with the MV-Register presented in specification 10 from [12]. This MV-register is slightly different from the one described in the previous sections, as its assign operation allows to assign multiple values to the register in one step. The problem is in the assign function. When the assigned list of elements is empty, the payload will also be empty after the operation. This is a problem, because all information about the current version is lost. It thus violates the requirement that updates monotonically increase the payload and it can lead to inconsistent replicas. As an example consider the following sequence of operations executed on replica 1: $\{(\bot, [0, 0])\} \xrightarrow{Assign(\{a\})} \{(a, [1, 0])\} \xrightarrow{Assign(\{b\})} \{(b, [2, 0])\} \xrightarrow{Assign(\{\})} \{\} \xrightarrow{Assign(\{c\})} \{(c, [1, 0])\}$. Furthermore assume that replica 2 first merges the payload $\{(b, [2, 0])\}$ and then the payload $\{(c, [1, 0])\}$. Then all updates have been delivered to both replicas, but the payload of replica 1 is $\{(c, [1, 0])\}$ and the payload of replica 2 is $\{(b, [2, 0])\}$. This problem can be easily fixed by disallowing the assignment of an empty set or by storing the current version in an extra field of the payload.

6 Alternative Specifications

We have already seen two specification techniques: specifications based on the complete update history, and abstract implementations, which are a kind of state-based specifications. Another specification technique was sketched in [1]. In this section we discuss and formalize the technique.

The technique is a state-based one, and uses the notation of pre- and post-conditions to specify the effect of operations on the state. Using the technique of pre- and post-conditions, a sequential specification can be given as a set of Hoare-triples. The Hoare-triple $\{P\}op\{Q\}$ requires that Q should hold after operation op whenever P was true before executing the operation.

For example, the increment operation of a counter can be specified by the triple $\{val() = i\}\ inc(x)\ \{val() = i + x\}$ and similarly a set is specified using triples like $\{true\}\ add(e)\ \{contains(e)\}$. Such a sequential specification is applicable to replicated data types if there is no interaction with other replicas between the pre- and post-condition.

In such cases, the replicated counter and the Observed-Remove-Set behave exactly as their corresponding sequential data type. For the Two-Phase-Set this is not true, since an add-operation does not guarantee that the element is in the set afterwards. There are examples like the Multi-Value-Register, where no corresponding and meaningful sequential data type exists, but for replicated data types which try to mimic sequential data types, it is a desirable property to maintain the sequential semantics in time spans where there are no concurrent operations, i.e. where the visibility relation describes a sequence.

In [1] those sequential specifications are combined with concurrent specifications, that describe the effect of executing operations concurrently. The concurrent specification is written in a similar style as the sequential specification.

Instead of only a single operation it considers several operations of the following form executed in parallel: $\{P\}op_1 \parallel op_2 \parallel \cdots \parallel op_n\{Q\}$. The informal meaning is that if P holds in a state s, then executing all operations on s independently and then merging the resulting states should yield a state where Q holds.

Formally, we define a triple $\{P\}op_1 \parallel op_2 \parallel \cdots \parallel op_n\{Q\}$ to be valid, if the following condition is met:

$$\begin{aligned}
&\forall_{tr,pls,vs,ph,pls',vs',ph',r1,\ldots,r_n,op_1,op_n} : s_{init} \xrightarrow{tr} (pls, vs, ph) \\
&\wedge\ (pls, vs, ph) \xrightarrow{Update(r_1,op_1);\ldots;Update(r_n,op_n)} (pls', vs', ph') \\
&\wedge\ \forall_{i\in\{1,\ldots,n\}}\ pls(r_i) = pls(r_1) \\
&\wedge\ P(pls(r_1)) \Rightarrow Q(merge_{crdt}(pls'(r_1), \ldots, pls'(r_n)))
\end{aligned}$$

If we reach a state (pls, vs, ph) where the payload on the replicas r_1 to r_n are equal and satisfy the pre-condition P, then executing each operation op_i on replica r_i yields a state (pls', vs', ph') where the post-condition Q holds for the merged payload of replicas r_1 to r_n.

Obviously, one can only specify a subset of all possible executions using this specification techniques. The advantages of this technique is that it is more modular and thus better composable than the technique introduced in Section 3, and that it is easier to see the sequential semantics of the data type. Also, there is the principle of permutation equivalence[1], which can be applied to this technique very easily, and is a good design guideline for designing new CRDTs.

7 Related Work

Burckhardt et al.[5] worked on verifying the behavioral properties of CRDTs. Their techniques are very similar to ours, but they have not used a tool to check their proofs. Their formal system model is more general than ours, as it supports timestamps and visibility relations which are not partial orders.

Bouajjani et al.[4] present a specification technique which is based on the history of updates with the visibility relation. They obtain a more flexible system model by allowing the partial order to be completely different for different operations. This allows them to cover a wide selection of optimistic replication systems, in particular ones that use speculative execution. In contrast to our work, they use an algorithmic approach to reduce the verification problem to model-checking and reachability.

The only other work we are aware of which uses a theorem prover to verify CRDTs is by Christopher Meiklejohn. Using the interactive theorem prover Coq, the semilattice properties of the increase-only-counter and the PN-counter CRDTs[9] are verified. Unlike our work, the behavioral properties of CRDTs are not considered and the verification is not based on a formal system model.

8 Conclusion and Future work

In this paper, we have presented a formal framework for the analysis and verification of CRDTs. As the case studies have shown, it is feasible to verify CRDTs with Isabelle/HOL. The problem found in the MV-register during verification shows that it is easy to miss some corner case when designing a CRDT. The verified CRDTs were given in pseudo-code and then translated to Isabelle, which is a very high level language. Real implementations of the same CRDTs will probably be more complex, and thus the chance of introducing bugs might be even higher. But also the amount of work required for verifying a real implementation is higher.

It is an open question if more research into the automated verification and testing of CRDTs is required. This depends on how applications will use CRDTs in the future. For sequential data types, it is often sufficient to have lists, sets, and maps for managing data, as can be seen in commonly used data formats like XML or JSON. In the case of CRDTs, more data types are required, because different applications require different conflict resolution behavior. This could be very application specific. For example, an application could require a set where add-operations win over remove-operations, but when a remove-operation is performed by an administrator of the system, then that operation should win. If every application needs its own CRDTs, then automatic tools to auto-generate correct code might be a good idea.

In future work, we want to extend the specification techniques presented in this paper for reasoning about applications using CRDTs. Such large-scale distributed applications are usually long-running and should be stable, but are difficult to maintain. It is therefore of special interest to have a stable and correct code base.

Acknowledgement. This research is supported in part by European FP7 project 609 551 SyncFree

References

1. Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In Marcos K. Aguilera, editor, *DISC*, volume 7611 of *Lecture Notes in Computer Science*, pages 441–442. Springer, 2012.
2. Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012.
3. Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012.
4. Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying eventual consistency of optimistic replication systems. In Jagannathan and Sewell [7], pages 285–296.

5. Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In Jagannathan and Sewell [7], pages 271–284.
6. Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.
7. Suresh Jagannathan and Peter Sewell, editors. *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, 2014.
8. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
9. Christopher Meiklejohn. Distributed data structures with Coq. `http://christophermeiklejohn.com/coq/2013/06/11/distributed-data-structures.html`, June 2013.
10. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
11. Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.
12. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, January 2011.
13. Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *SSS*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011.
14. Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–183, 1995.
15. Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, 2008.
16. Peter Zeller. Specification and Verification of Convergent Replicated Data Types. Master's thesis, TU Kaiserslautern, Germany, 2013.