Master's Thesis

Specification and Verification of Convergent Replicated Data Types

Peter Zeller
p_zeller@cs.uni-kl.de

November 2013

Department of Computer Science,
University of Kaiserslautern,
D 67653 Kaiserslautern,
Germany

Supervisors:

Prof. Dr. Arnd Poetzsch-Heffter
Dr. Annette Bieniusa

Ich versichere hiermit, dass ich die vorliegende Masterarbeit mit dem Thema “Specification and Verification of Convergent Replicated Data Types” selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, habe ich durch die Angabe der Quelle, auch der benutzten Sekundärliteratur, als Entlehnung kenntlich gemacht.

(Ort, Datum) (Unterschrift)

Abstract *Conflict Free Replicated Data Types* (CRDTs) can be used as basic building blocks for storing and managing data in a distributed system. They provide high availability and performance, and they guarantee that conflicts are resolved in a well defined way. In this master's thesis, techniques for verifying these data types with the interactive theorem prover *Isabelle/HOL* are presented. The verification covers convergence properties, as well as behavioral specifications. For this task, a basic framework for the verification of CRDTs has been developed and several known CRDTs from the literature have been verified using the framework. The result are machine checked proofs for the correctness of the CRDTs, which rely on only a small set of unchecked assumptions, which are captured in a simple system model.

Zusammenfassung *Conflict Free Replicated Data Types* (CRDTs) können als grundlegende Bausteine für das Speichern und Verwalten von Daten in verteilten Systemen verwendet werden. Sie bieten eine hohe Verfügbarkeit und Performanz und sie garantieren, dass Konflikte auf eine wohldefinierte Art und Weise aufgelöst werden. In dieser Arbeit werden Techniken zum verifizieren dieser Datentypen mit dem interaktivem Theorem-Beweiser *Isabelle/HOL* vorgestellt. Die Verifikation umfasst sowohl Konvergenz-Eigenschaften, als auch Spezifikationen, die das Verhalten betreffen. Dafür wurde ein grundlegendes System für das verifizieren von CRDTs entwickelt und mehrere bekannte CRDTs aus der Literatur wurden damit verifiziert. Das Ergebnis sind vom Computer geprüfte Beweise für die Korrektheit dieser CRDTs, die nur von einer kleinen Menge von nicht geprüften Annahmen ausgehen, welche in einem einfachen System-Modell zusammengefasst sind.

Contents

1. Introduction

For many distributed systems, there is a big challenge in managing data. Using a single node to store data is often not a feasible option. There are different reasons for this. Some systems require a higher throughput for writing and reading data, than what can be provided by a single node. Other systems require a high availability, which can only be provided by several nodes without a single point of failure. Then there are systems which serve clients from different regions in the world and want to avoid the delay of messages sent around half the globe. So these systems need nodes located at different places in the world. Finally there are systems where nodes are not always connected. For example mobile phones and tablets are not always connected to the Internet and thus need a local data store on the device, which is only synchronized with a more central data store when a connection is available.

Classical relational, transactional databases like Microsoft SQL Server, Oracle, or IBM DB2 try to keep the data stores on different nodes (also called replicas) consistent all the time. But requiring strong consistency has a negative effect on the availability and delay of the data store operations. Some newer databases like Riak or Cassandra provide weaker consistency guarantees and therefore can achieve better availability and performance. These kind of databases are called eventual consistent as opposed to strongly consistent. Such databases can often process an operation locally and then synchronize the changes between the data stores asynchronous to the operation.

For many use cases this implies that there can be conflicting updates in distinct data stores. In such a case the conflict can only be detected after the operation was completed, and it has to be resolved either by the database itself or by the application. Letting the application handle conflicts increases the complexity of the application, the probability of loosing data due to programming errors, and it requires that the database stores all necessary information about conflicts until they are resolved by the application. Therefore developers usually want the database to handle conflicts by itself according to some reasonable rules.

Unfortunately there is no universal rule to resolve conflicts, as it always depends on the application. As an example consider an integer variable which is changed from 42 to 43 on one replica and changed from 42 to 46 on an other replica. Now there are the two conflicting values 43 and 46. If the application's intend was counting, then it would make sense to resolve the conflict by setting the variable to 47 as this includes the one new element counted at the first replica and the four new elements counted at the second replica. However, if the application's intend was to set the variable to some absolute value, it might make more sense to resolve the conflict by keeping both values or by taking an aggregate function like the maximum of the values.

Behaviors like the counter are required in more than one application, so it makes sense to put this behavior in a reusable data type. Convergent Replicated Data Types (CRDTs) are such data types. They are similar to data types for sequential programs such as lists, sets, or maps, but they are replicated across multiple nodes. Operations can be performed on only a single replica and updates are communicated to other replicas

at some later time. A common property of all CRDTs is, that they converge, so if two replicas have seen the same set of updates they are guaranteed to be in the same state. This guarantee is also preserved, when replicas are not synchronized for a longer period of time and therefore CRDTs are a good approach to address the afore mentioned aspects, namely throughput, availability, delay, and connectivity.

When CRDTs are used as data storage of an application, it is essential that the CRDT implementations behave as intended by the application programmer. Therefore it is necessary to have a precise specification of how the CRDT behaves and it must be verified that the implementation is a correct implementation of the specification.

Testing is one option to check if an implementation behaves as intended, but it is hard to test all relevant cases. This is especially hard in a concurrent environment, where even more different executions are possible for a given number of operations. If a bug is not detected in testing, it is likely that it only occurs in certain corner cases. This can lead to bugs in an application, which are hard to find and hard to debug. Since concurrent applications are usually highly nondeterministic, it can be hard to reproduce a bug which only happens in certain cases and it could take some time to detect the problem. Until the problem is detected, a lot of important data could be lost. Because CRDTs can be used in a lot of applications, there would probably be more than one application affected by a bug in a CRDT application. Therefore it is important to be sure, that implementations are correct. As testing cannot guarantee correctness, it is necessary to use static verification. If the static verification is checked by a tool, one can be sure that an implementation behaves as specified.

Verified, formal specifications of CRDTs are also necessary for verifying applications, which use CRDTs. It is not realistic to verify a whole application directly. Instead libraries, and CRDTs can be seen as a library, have to provide a well defined interface with a specification, on which higher level proofs can be based.

This master's thesis presents a framework for specifying and verifying CRDTs with the interactive theorem prover Isabelle/HOL[10]. Different approaches for specifying the behavior of CRDTs are discussed and the specification and verification techniques are applied to several CRDT implementations found in the literature.

The remainder of this thesis is structured as follows. Section 2 gives a short introduction to CRDTs and presents state-based and operation-based types. The later chapter only consider state-based types. Section 3 presents the underlying system model for this work. Section 4 defines general consistency and convergence properties of CRDTs and the theory required to verify convergence of a implementation. Section 5 discusses different approaches for specifying CRDTs and presents the formalization of the used specification technique in Isabelle. Section 6 presents the techniques used for verifying that an implementation satisfies its specification. Then section 7 applies the previously introduced techniques to case studies, which are CRDT implementations from the literature.

2. Conflict-Free Replicated Data Types

The general idea of Conflict-Free Replicated Data Types (CRDTs) is, that they form a reusable component for storing data in a distributed system. A CRDT object is usually replicated onto several nodes in the system and has to include functionality for keeping the different replicas synchronized. The interface of a CRDT consists of several methods for updating and querying the state of the data type. These methods can always be executed locally on a single replica. The synchronization with other replicas can happen at a later point in time. [12]

The synchronization also is done in a single message, so updates are visible after a single message is exchanged. The synchronization does not depend on consensus protocols like Paxos, or on locking mechanisms, and therefore synchronization is relatively fast.

There are two different kind of CRDTs: operation based types and state based types. The two types differ in the mechanism used for synchronization. Both types have in common, that they provide a number of update- and query-operations and that a single instance of a CRDT consists of several replicas. Every replica has a local state which is called the **payload** of the object. There is no global state or central authority in a CRDT instance.

2.1. Operation Based CRDTs

Operation based CRDTs are also called Commutative Replicated Data Types (CmRDTs).

Operation based CRDTs synchronize on each operation. When an update-method is executed, the local replica first calculates a so called *downstream effect*. The downstream effect is basically a function $f : Pl \to Pl$ where Pl is the type of the CRDT's payload. This function is then used to update the payload of every replica in the system. The local payload is updated immediately and the payload at other replicas is updated asynchronously.

All generated downstream effects are expected to commute. So, if f and g are downstream effects, then it should hold that $f(g(pl)) = g(f(pl))$ for all possible payloads pl.

Therefore the underlying network protocol is not required to deliver the downstream effects in a certain order. However is must be ensured, that each downstream effect is executed exactly once, as downstream effects are not necessary idempotent (i.e. $f(f(pl)) = f(pl)$ does not hold in general).

2.1.1. Example: Counter

Figure 1 shows the implementation of an operation based counter[12]. It allows to increment and decrement an integer with the two provided update operations. Because the downstream effects of the update operations commute with each other, this is a CmRDT.

```
payload integer i
    initial 0
query value() : integer j
    let j = i
update increment()
    downstream()
        i := i + 1
update decrement()
    downstream()
        i := i − 1
```

Figure 1: Operation based counter, specification 5 from[12]

2.1.2. Example: U-Set

Figure 2 shows the implementation of an operation based U-Set[12]. This set allows adding an element only once with the add-operation. When it has been added it can be removed with the remove-operation. The precondition of the downstream effect of the remove-operation requires that the corresponding add-operation has already been delivered to the replica. Otherwise the downstream effect of the remove-operation and the downstream effect of the add-operation would not commute. If the system guarantees, that messages are delivered in an order consistent with the happens-before relation, then all downstream effects commute and therefore this example is a CmRDT.

2.2. State Based CRDTs

State based CRDTs are also called Convergent Replicated Data Types (CvRDTs). In the remainder of this thesis, "CRDT" refers to state based CRDTs. The developed theory and the case studies only consider state based data types.

State based CRDTs are synchronized by exchanging the payload between different replicas and merging them together. A state based CRDT implementation has to provide a merge function for doing the merge. The merge function takes two payloads and returns a new payload.

To ensure the convergence of different replicas the merge function has to compute a least upper bound with respect to some partial order and updates have to monotonically increase the value of payloads with respect to that partial order. These properties are discussed in more detail in section 4.

With this mechanism it is not necessary that network messages are delivered exactly once, as it was the case for the operation based approach. Messages can be delivered in any order and multiple times.

payload set S
    initial $\emptyset$
query lookup(element e) : boolean b
    let $b = (e \in S)$
update add(element e)
    atSource(e)
        pre e is unique
    downstream()
        $S := S \cup \{e\}$
update remove(element e)
    atSource(e)
        pre lookup(e)
    downstream(e)
        pre add(e) has been delivered
        $S := S \setminus \{e\}$

Figure 2: Operation based U-Set, specification 13 from[12]

2.2.1. Example: Counter

Figure 3 shows the implementation of a state based counter[12]. The payload is a vector of integers, where each component of the vector stores the number of increments on one replica. The value of the counter is then just the sum of all the individual counts. Two payloads are merged by taking the component-wise maximum, and as the compare function also does a component-wise comparison, it is easy to see that the merge function computes a least upper bound. Also the increment function increases the payload in the order. Therefore this counter implementation is a CRDT.

If the implementation had only used one integer instead of a vector of integers, it would still be able satisfy the required properties for a state based CRDT, but it would not count the number of increments correctly. For example, if two replicas would perform an increment in parallel starting from the initial state, then both would have a payload of 1. After merging the two payloads it would still be 1, since this is the maximum of the two. In contrast the correct implementation will result in payloads $[1, 0]$ and $[0, 1]$ and the merged value will be $[1, 1]$, which represents the correct value of 2 increment operations.

2.2.2. Example: Two-Phase-Set

Figure 4 shows an implementation of a Two-Phase-Set[12]. In this set elements can be added and removed, but after an element has been removed, the same element cannot be added again.

The payload consists of two sets. The set A contains all added elements and the set

payload integer[n] P
    initial $[0, 0, ..., 0]$
update increment()
    let $g = myID()$
    $P[g] := P[g] + 1$
query value() : integer v
    let $v = \sum_i P[i]$
compare (X,Y) : boolean b
    let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i])$
merge (X,Y) : **payload** Z
    let $\forall i \in [0, n-1] : Z.P[i] = max(X.P[i], Y.P[i])$

Figure 3: State based counter, specification 6 from[12]

R contains all removed elements. Since update operations only add elements to the payload, the monotonicity requirements are satisfied, and since set union computes a least upper bound, this implementation is a CRDT.

2.2.3. Example: Observed-Remove-Set

The Observed-Remove-Set (OR-Set) is a set which allows adding and removing elements any number of times. An remove operation only affects the add operations which happened before the remove operation, or phrased differently, a remove only affects the already observed add operations. So if an operation Add(x) is executed in parallel to an operation Remove(x), the add operation will win, as it is not visible to the remove operation. This is why the Observed-Remove-Set is also called Add-Wins-Set.

The implementation of this set is shown in figure 5[11]. Similar to the Two-Phase-Set the payload consists of a set of elements and a set of tombstones. But the sets do not just contain the single elements. Instead each add operation generates a unique identifier and stores elements together with that identifier. This allows the remove-operation to only affect the add-operations which already are observed. Add-operations which happen in parallel to the remove-operation or after it will have a unique identifier which is not in the tombstones set.

payload set A, set R
    initial $\emptyset, \emptyset$
query lookup(element e) : boolean b
    let $b = (e \in A \wedge e \notin R)$
update add(element e)
    A := $A \cup \{e\}$
update remove(element e)
    pre lookup(e)
    R := $R \cup \{e\}$
compare (S,T) : boolean b
    let $b = (S.A \subseteq T.A \wedge S.R \subseteq T.R)$
merge (S,T) : **payload** U
    let $U.A = S.A \cup T.A$
    let $U.R = S.R \cup T.R$

Figure 4: State-based 2P-set, specification 12 from[12]

payload set E, set T
    initial $\emptyset, \emptyset$
query contains(element e) : boolean b
    let $b = (\exists n : (e, n) \in E)$
update add(element e)
    let n = unique()
    $E := E \cup \{(e, n)\}$
update remove(element e)
    let R = $\{(e, n) | \exists n : (e, n) \in E\}$
    E := $E \setminus R$
    T := $T \cup R$
compare (A,B) : boolean b
    let $b = ((A.E \cup A.T) \subseteq (B.E \cup B.T)) \wedge (A.T \subseteq B.T)$
merge (B)
    $E := (E \setminus B.T) \cup (B.E \setminus T)$
    $T := T \cup B.T$

Figure 5: State-based OR-set, based on figure 2 from [1, 11] but without the operation based elements

3. System Model for State Based Data Types

To be able to formally reason about data types, a formal model of the system is needed. The formal model has to be suitable for the Isabelle proof assistant tool. It has to reflect the reality in a way, that the results proven with the model can be transferred to the real world. At the same time, the model must be simple and abstract from lower level details, so that the proofs can focus on the essential aspects without getting too technical.

When designing a formal model, there always is a trade-off between simplicity and generality. In this chapter a system model for reasoning about state based CRDTs is presented along with the design decisions taken.

3.1. Overview

The model considers only one instance of a CRDT. For the purpose of this master's thesis this is sufficient as it is not a goal to study the interplay between several objects, which would have been important when considering transactions or programs interacting with several objects in an interweaved way. Instead this thesis focuses on the behavior of one object which interacts with its environment. The environment is not modeled explicitly, but instead the system model uses traces which capture the interactions between the environment and the different replicas of the data type instance. The trace also captures the interactions between the replicas, which are the merge operations. The merge operations happen independently from the update operations. The traces and execution semantics are explained in more detail later.

The number of replicas is arbitrary but fixed in the system model, which means adding and removing replicas at runtime is not part of the model. In practice this is a necessary feature, and it requires some additional work to implement this correctly. However, the basic mechanics of the CRDTs are not influenced by this and dynamically adding and removing replicas can in parts be represented by a model with a fixed number of replicas. Removing a replica can be represented by no longer sending any messages to a replica. Adding a replica can be represented by having a replica, which does not receive any messages until it is "added" to the system.

Updates are executed atomically. Immediately after every update, the new state of the replica is implicitly sent to all other replicas. The other replicas can receive this state at any time in the future and merge it with their own state. It is also possible that they never receive the state or receive the same state multiple times. The model does not restrict the order in which such merge messages are delivered. The only assumptions on message delivery are, that message contents are not corrupted, and the obvious assumption that messages cannot be received before they are sent. The assumption that messages are not corrupted is necessary in order prove any property about the behavior of the system. If messages could be corrupted in arbitrary ways, the system would be unpredictable.

An alternative approach to the implicit sending of messages, would be to have an explicit send-action. This is what has been done in *Replicated Data Types: Specification, Verification, Optimality*[3], where send operations are explicit and are also allowed to

```
definition replicaCount :: nat where "replicaCount = (SOME i. i>0)"

typedef replicaId = "{x::nat. x < replicaCount}"
by (rule_tac x="0" in exI, auto)
```

Figure 6: Modeling of replicas in Isabelle

change the state. However this feature of the model is only relevant for operation based CRDTs, which are not considered in this thesis. The implicit send operation after every update makes the model simpler, as it ensures that the set of current payloads on the replicas is a subset of the set of payloads in the messages. Therefore proofs do not have to distinguish the two places where payloads can occur and can instead only work on the messages. Furthermore the sent messages present a history of all payloads including current and past payloads. This makes it easier to formulate properties which relate states from different points in time (see section 4, figure 13).

In the following the realization of the above description in Isabelle/HOL is described.

3.2. Replicas and Version Vectors

The basis for replicas in Isabelle is shown in figure 6. As already stated above, there is a fixed number of replicas. In Isabelle the number of replicas is defined as some integer greater zero. The use of the SOME function makes sure that proofs cannot assume anything about the number of replicas, except that there is at least one replica and that the number is finite. The type definition replicaId introduces a new type with one element for each replica. The type is represented by the numbers $\{0, 1, 2, \ldots, replicaCount - 1\}$.

Another fundamental concept for the system model are version vectors. Version vectors (also known as vector clocks)[6] are a technique to maintain a happens-before relation between events in a distributed system. Time stamps are not suitable for this task, because they define a total order on events, while the happens-before relation is a partial order. Events can happen in parallel.

Version vectors are implemented as a mapping from replica-Ids to natural numbers, where the number is the number of updates executed on the respective replica. The mapping can also be written as a vector where the nth component is the entry for the nth replica. For example the vector $[3, 0, 2]$ denotes that there has been observed 3 updates from replica 0, 0 updates from replica 1 and 2 updates from replica 2.

A version vector is less or equal to an other version vector, if all entries of the version vector are less or equal. When two version vectors are incomparable (i.e $v_1 \not\leqslant v_2$ and $v_1 \not\geqslant v_2$), this is also written as $v_1 \parallel v_2$. The $\leqslant$ order on version vector is exactly the happens-before relation. The least upper bound of two version vectors can be calculated by taking the component-wise maximum of the two vectors. The least upper bound is written as $sup\ x\ y$ or as $x \sqcup y$. Accessing the entry for a replica r of a version vector v

```
record ('pl, 'ua, 'qa, 'r) stateBasedType =
        t_compare :: "'pl ⇒ 'pl ⇒ bool"
        t_merge :: "'pl ⇒ 'pl ⇒ 'pl"
        t_initial :: "'pl"
        t_update ::"'ua ⇒ replicaId ⇒ 'pl ⇒ 'pl"
        t_query ::"'qa ⇒ 'pl ⇒ 'r"
```

Figure 7: General structure of state based types

is written as v_r or as v»r.

One limitation of version vectors is, that they can only work in a system where updates are observed transitively, so if an update x has been delivered to a replica and y is an update which happened before x, then y must also have been delivered. With state-based CRDTs this is the case, as merges always receive the whole payloads, and the payloads represent all observed updates.

Since the system model only supports state based data types, version vectors are a good way for representing the happens-before relation. In contrast to having an explicit relation, version vectors have the advantage that properties like transitivity are already in the structure of version vectors, and do not have to be stated explicitly.

3.3. Data type record

The general structure of data types is defined by the record in figure 7. A concrete CRDT implementation is then just an instance of the record. A concrete example is given in figure 8 where an implementation of a two phase set is shown.

The record is parametrized by four type parameters:

'pl	The type of the payload
'ua	Type for argument to the update function
'qa	Type for argument to the query function
'r	Return type of query function

The fields of the record are the same as used in [12], except for the fact that there is only one update method and only one query method. But this limitation can easily be worked around by using a sum type for the argument.

The compare function t_compare :: "'pl ⇒ 'pl ⇒ bool" defines a partial order on the payload. It returns true if the the first argument is less than or equal to the second argument and false otherwise. To make this relation more readable the notation x ≤ [crdt] y is used instead of writing t_compare crdt x y. The compare function is only required for proving the convergence property of the CRDT. It does not influence the behavior of the CRDT, so it could have been omitted at this place. It is included here mainly for the reason to be consistent with the CRDT notation in [12], where the

```
type_synonym 'a payload = "'a set × 'a set"

datatype 'a updateArgs = Add 'a | Remove 'a

fun update :: "'a updateArgs ⇒ replicaId ⇒ 'a payload ⇒ 'a payload" where
  "update (Add x) r (E,T) = (insert x E, T)"
| "update (Remove x) r (E,T) = (E, insert x T)"

fun lookup :: "'a ⇒ 'a payload ⇒ bool" where
"lookup x (E,T) = (x∈ E ∧ x∉T)"

definition twoPhaseSet :: "('a payload, 'a updateArgs, 'a, bool) stateBasedType" where
"twoPhaseSet = (|
      t_compare = (λx y. fst x ⊆ fst y ∧ snd x ⊆ snd y),
      t_merge = (λx y. (fst x ∪ fst y, snd x ∪ snd y)),
      t_initial = ({},{}),
      t_update = update,
      t_query = lookup
  |)"
```

Figure 8: A two phase set as represented in Isabelle

compare function is also noted as part of the data type.

The merge function t_merge :: "'pl ⇒ 'pl ⇒ 'pl" defines how two payloads are merged together. An alternative notation for t_merge crdt x y is x ⊔[crdt] y.

The initial payload is determined by the field t_initial :: "'pl". Note that the initial payload does not depend on any parameters, so all replicas will start with the same payload. In [12] this is handled the same way, but theoretically it would be possible to have different initial payloads on different replicas as long as the query functions hide this difference and returns the same result for all replicas. However allowing this would also have consequences on the semilattice conditions required for convergence and probably make the proofs more difficult.

The function t_update ::"'ua ⇒ replicaId ⇒ 'pl ⇒ 'pl" is responsible for update operations. The first parameter is generic and depends on the implemented data type. For example the two phase set in figure 8 uses a data type with cases for adding and removing elements for this parameter. As shown by the example, one update operation is enough to model an arbitrary number of methods a data type might provide. The second parameter of the update function is the replicaId of the replica where the operation is performed. This parameter is essential for CRDTs like the counter, while other CRDTs can ignore this parameter. The last parameter and the return type are of the payload's type, which represents the fact, that an update operation will update the payload. The used parameters capture the essential aspects of CRDT updates, but it deliberately omits some possibilities.

For example some CRDTs like the last-writer-wins-register require a time stamp in the update operations. Such examples cannot be expressed in this model, but usually timestamps only have the purpose of extending the partial happens-before order on events to a unique total order. This can also be done by other means, for example by using the lexicographical ordering of vector clocks. Adding time stamps to the model would be difficult, because it is not clear what assumptions about time stamps are valid. One question would be, if time stamps increase monotonically. Most of the time this is the case but there are some corner cases like leap seconds[5]. An other question would be if the ordering induced by time stamps is consistent with the happens-before relation. As clocks on different nodes are probably not synchronized perfectly in every system, this is probably not true.

An other possible addition to the update function would have been the possibility to generate globally unique identifiers. For example the Observed-Remove-Set uses such identifiers to distinguish different updates. In practice one could use big enough random identifiers like UUIDs so that the probability of a collision is small enough, but for correctness proofs this would be infeasible. Besides that concern it is not possible to create random numbers out of thin air, so some kind of support would have to be added to the model. Luckily, globally unique identifiers can also be generated without adding a special mechanism to the model as seen in the case study of the observed remove set (see section 7.5).

Finally there is the query function t_query ::"'qa ⇒ 'pl ⇒ 'r" . Like in the update function, the first parameter is generic and depends on the CRDT. The same holds true

for the return type. Besides the first parameter the result of a query only depends on the local payload of the queried replica.

Queries and updates could theoretically be combined into one function of type 'ua ⇒ replicaId ⇒ 'pl ⇒ ('pl × 'r). This would make the model more powerful, as it would allow operations which atomically modify the state and return a value. For example the Add operation on sets could return whether the element already was in the set. Instead the model requires to first evaluate a query and then execute an update. This is not a big restriction for the model, as queries can be evaluated on a specific state. So there is no problem of race conditions, as there would be when doing the same in a real system. An advantage of a separate query function is, that some statements are easier to express. For example specifications often use queries for describing pre- and post-conditions and side effects on the payload would be distractive there.

3.4. Traces and Actions

The interactions of the system with its environment and the interactions between different nodes are captured in traces. In this model a trace is a finite list of actions. A trace describes the behavior of the whole system, which consists of many replicas, but a single action only affects a single replica.

The formal definition of actions and traces is given in figure 9. There are two kind of actions which can happen in the system. Both kind of actions take a replicaId as their first parameter, which determines the replica executing the action.

```
datatype ('ua, 'pl) action =
    Update replicaId 'ua
  | MergePayload replicaId versionVector 'pl

datatype ('ua, 'pl) trace =
     EmptyTrace
   | Trace "('ua, 'pl) trace" "('ua, 'pl) action"
```

Figure 9: Traces and Actions

The first kind of action is an update. Updates have a second parameter for the update arguments ('ua). This is the generic parameter for updates as also used in the data type record.

The other kind of action is a merge. A merge takes a version vector and a payload ('pl). The idea is, that the replica receives the payload from an older version given by the version vector. The version vector alone would be sufficient to determine the payload, if it could be assumed that the CRDT converges. However this cannot be assumed a priori and has to be proven first. So to keep the model deterministic, the payload was included in the merge action. One disadvantage of including the payload in the traces is, that

it exposes implementation details in the traces, which makes it a bit more difficult to compare different implementations of the same abstract CRDT. An alternative approach would have been to use unique identifiers instead of the version vectors, so that there could be a unique mapping from identifiers to payloads.

As an example, consider the execution shown in figure 10. This execution can be described by the following trace, assuming the underlying implementation is a Two-Phase-Set as introduced in section 2.2.2, where the payload consists of an elements- and a tombstones-set:
$[Update(r_1, add(x)),$
$Update(r_2, add(y)),$
$MergePayload(r_3, [0,1,0], (\{y\}, \{\})),$
$MergePayload(r_2, [1,0,0], (\{x\}, \{\})),$
$Update(r_3, add(x)),$
$Update(r_2, remove(x)),$
$Update(r_3, remove(y)),$
$MergePayload(r_2, [0,1,2], (\{x,y\}, \{y\}))].$

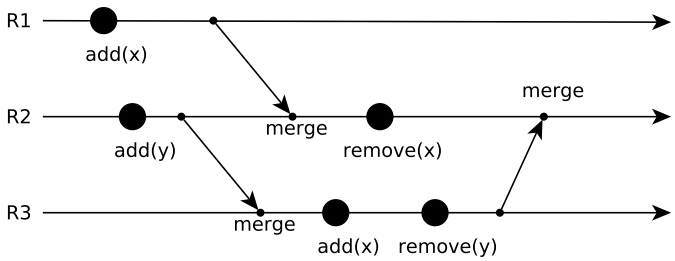

Figure 10: Example Execution

Basing the semantics of the model on traces allows the semantics to be deterministic. For one given trace there is only one possible execution. This makes it easier to reason about the system. But from the perspective of a client, who cannot see the internal merge operations happening in the system, the semantics are still nondeterministic.

Furthermore traces have the advantage, that it is easy to do a proof by induction over traces, whereas an induction over a graph, for example the happens-before graph, is more difficult.

3.5. System State

The state of the system (see figure 11) contains a history of all payloads with their version vector. This history represents messages sent between replicas. A replica can receive any payload which is in the history.

The two other fields of the system state are replicaPayload and replicaVV which store the current state and version vector for each replica.

```
type_synonym 'pl history = "(versionVector × 'pl) set"

record 'pl systemState =
  replicaPayload :: "replicaId ⇒ 'pl"
  replicaVV :: "replicaId ⇒ versionVector"
  history :: "'pl history"
```

Figure 11: System State

Note that $(replicaVV\ r,\ replicaPayload\ r) \in history$ holds for every replica. Compared to having an explicit message channel and explicit sending this makes the analysis less technical in some cases. Nevertheless it is still clear that the results could easily be transferred to a model with explicit message sending.

3.6. Operational Semantics

The operational semantics are defined by two relations step and steps and the initial state (see figure 12).

Initially, each replica has the initial payload and the all-zero version vector (vvZero) and the history also contains only this entry.

The relation step defines the effect of a single action. An update action updates the payload of the replica based on the update function of the data type and it increases the version vector by one for the local replica (function incVV).

A merge action updates the payload of the replica by using the merge operation of the data type and the received payload. The new version vector is defined by taking the supremum of both version vectors. A merge operation can only be executed, when the merged state is in the history ((vv,pl)∈ history s).

Both transitions (merges and updates) use the helper function updateState to update the state. This function not only updates the current state of the updated replica, but also adds the new version vector and payload to the history, so that it can be received by later merge operations. Putting the element in the history can also be seen as sending messages to all replicas.

The relation steps is just lifting the step relation to traces, which are finite sequences of steps.

```
 definition initialState :: "('pl,'ua,'qa,'r) stateBasedType ⇒ ('pl) systemState" where
"initialState crdt = ⦇
  replicaPayload= (λr. t_initial crdt),
  replicaVV=(λr. vvZero),
  history={(vvZero, t_initial crdt)}
⦈"

fun updateState :: "replicaId ⇒ (versionVector ⇒ versionVector) ⇒ ('pl ⇒ 'pl)
                          ⇒ ('pl) systemState ⇒ ('pl) systemState" where
"updateState rep updateVV updatePl state = (
  let newPl = updatePl (replicaPayload state rep);
      newVV = updateVV (replicaVV state rep) in
⦇
  replicaPayload = (replicaPayload state)(rep:= newPl),
  replicaVV = (replicaVV state)(rep:= newVV),
  history = (history state) ∪ {(newVV, newPl)}
⦈)"


inductive step :: "('pl, 'ua, 'qa, 'r) stateBasedType ⇒ ('ua, 'pl) action ⇒ ('pl)
  systemState ⇒ ('pl) systemState ⇒ bool" where
 update:
  "step crdt (Update r args) s
           (updateState r (incVV r) (t_update crdt args r) s)"
| merge:
   "(vv,pl)∈ history s ⟹
    step crdt (MergePayload r vv pl) s
      (updateState r (sup vv) (t_merge crdt pl) s)"

inductive steps :: "('pl, 'ua, 'qa, 'r) stateBasedType ⇒ ('ua, 'pl) trace ⇒ ('pl)
  systemState ⇒ ('pl) systemState ⇒ bool" where
  start: "steps crdt EmptyTrace s s"
| step: "⟦steps crdt as s1 s2; step crdt a s2 s3 ⟧ ⟹ steps crdt (Trace as a) s1 s3"
```

Figure 12: Operational Semantics

4. Consistency

One of the most important properties of CRDTs is the provided consistency guarantee. As updates can be executed on a single replica, CRDTs cannot provide strong consistency guarantees, where it would be guaranteed that the system behaves like it was a centralized system with a single replica. Instead CRDTs only guarantee a form of eventual consistency.

4.1. Eventual Consistency

There are different definitions of eventual consistency, but intuitively they all capture the idea, that not all replicas will necessarily provide the same answer to a query and only after some time the replicas will converge and eventually return the same answer to queries.

In [12] the definition of eventual convergence consists of a safety property and a liveness property. The liveness property states that every update operation is eventually delivered to all replicas. The safety property states, that if two replicas have seen the same set of updates, then their abstract state has to be the same, where the abstract state is defined as the state which can be observed by queries.

In [2] roughly the same liveness property is defined more formally as follows:

$$\forall a \in A.\ \neg(\exists \text{ infinitely many } b \in A.\ sameobj(a, b) \wedge \neg(a \xrightarrow{vis} b))$$

Here A is the set of actions and $\xrightarrow{vis}$ denotes the visibility relation between actions. So the definition states that for every action, there cannot be an infinite number of actions, which have not seen the former action.

The problem with these kind of liveness definition is that it can only be applied to infinite executions and the system model introduced in section 3 only models finite executions. With such a model it is only possible to define the liveness property as the possibility to reach a desired state: In every reachable state of the system it should be possible to execute a number of steps, so that any update operation is visible on any replica. However this property trivially holds, as merge operations with previous states are always enabled in the model, so it is always possible to perform a merge operation in order to get visibility.

As the system model cannot make any nontrivial liveness guarantees, the safety property is the only remaining property with respect to consistency and convergence. The formalization of the property can be found in figure 13. The definition actually is a bit stronger, than the informal definition presented above, as it not only relates replicas at a fixed point in time, but can relate any elements from the history. As the current state is a part of the history is is a strictly stronger definition.

The definition states, that if two elements from the history of a reachable state s have the same version ($v = v'$), then they must also have the same payload ($pl = pl'$). Note that the version v captures exactly, which updates are visible at a that point, so an equal version implies that the same same set of updates is visible for both elements.

```
definition convergent_crdt :: "('pl,'ua','qa,'r) stateBasedType ⇒ bool" where
"convergent_crdt crdt = (∀tr s v pl v' pl'.
  steps crdt tr (initialState crdt) s
∧ (v , pl ) ∈ history s
∧ (v', pl') ∈ history s
∧ v=v' ⟶ pl=pl')"
```

Figure 13: Convergence definition 1

```
definition convergent_crdt2 :: "('pl,'ua','qa,'r) stateBasedType ⇒ bool" where
"convergent_crdt2 crdt = (∀tr s v pl v' pl'.
  steps crdt tr (initialState crdt) s
∧ (v , pl ) ∈ history s
∧ (v', pl') ∈ history s
∧ v≤ v' ⟶ pl ≤ [crdt] pl')"
```

Figure 14: Convergence definition 2

When considering the partial order defined on the CRDT, it is possible to define the even stronger convergence property shown in figure 14. This definition states that one element in the history has seen a subset of the updates from the other element ($v \leq v'$), then the payload of the first element is also smaller than the payload of the second element with respect to the partial order defined for the CRDT. As a partial order is antisymmetric, this definition is stronger than the first definition.

4.2. A sufficient condition for Convergence

There is a well known sufficient condition for CRDTs to converge: If the payload of a CRDT forms a semilattice where the merge operation computes a least upper bound with respect to a partial order and update operations monotonically increase a payload in the order, then the CRDT converges[12].

My first approach was to use the semilattice typeclass provided by the Isabelle Main library as the basis for these conditions. But the case studies showed, that CRDTs are often not designed with one fixed semilattice in mind. The semilattice properties just hold for payloads which are actually reachable by the implementation. For example the original OR-Set, which will be presented in section 7.5.2, has two sets as the payload, but it can never reach a payload, where the same element is in both sets.

So the semilattice properties hold only with respect to a certain invariant. My first approach was to define a new data type for the payload, which only contains the elements

satisfying the invariant. However this approach led to a lot of proof steps, which were just about conversions between the newly defined data types and standard data types. Therefore I decided to use standard data types instead and define a separate invariant, although this had the drawback, that it was no longer possible to use the semilattice typeclass. So a lot of semilattice related lemmas from the Main library had to be ported to the new approach.

The invariant used has the following type:

```
type_synonym ('pl, 'ua) crdtInvariant = "'ua updateHistory ⇒ 'pl ⇒ bool"
```

It does not only depend on the payload, but also on the update history (UH). The update history contains all updates happened in the system and the happens-before relation between updates. It will be introduced in more detail in section 5.3. The purpose of adding this parameter to the invariant is to have more flexibility in defining the invariant. When the invariant depends only on the payload, it can only describe local properties. When the update history is included, it can also reference the global set of updates which happened so far. It turned out, that for the CRDTs in the case studies this feature was not required, but it was left in the framework, in case it is required in future work.

Definition 4.2.1 (Invariant Preserving)
A CRDT is invariant preserving if the following conditions hold:

1. The invariant has to hold for the initial update history and initial payload.
2. A merge must preserve the invariant.
3. An update must preserve the invariant.
4. An update must not invalidate the invariant of older payloads (this could be the case because the update history changes).

A formal definition of these properties is shown in figure 15. It would be possible to make the invariant more precise by describing the update history in more detail, similar to what is done in section 6.1. But as this was not required for verifying the case study, this was not done.

Definition 4.2.2 (Monotonic Updates)
A CRDT has monotonic updates, when the payload after an update is at least as big as before in the partial order defined by the CRDT (figure 16).

Definition 4.2.3 (Semilattice Properties)
A CRDT has the required semilattice properties, when the compare operations defines a partial order and the merge operation computes a least upper bound with respect to the partial order.

To be a partial order, the order must be reflexive, transitive and antisymmetric.

To show that merge computes a least upper bound, it suffices to show that:

```
definition "invariantInitial crdt Inv ≡
  (Inv initialUpdateHistory (t_initial crdt))"
definition "invariantMerge crdt Inv ≡
  (∀UH x y. Inv UH x ∧ Inv UH y ⟶ Inv UH (t_merge crdt x y))"
definition "invariantUpdate crdt Inv ≡ (∀args UH x rx v. Inv UH x ⟶
  Inv (UH(rx:=UH rx@[(v,args)])) (t_update crdt args rx x))"
definition "invariantUpdateOther crdt Inv ≡ (∀args UH ry x v. Inv UH x ⟶
  Inv (UH(ry:=UH ry@[(v,args)])) x)"
```

Figure 15: Properties for preserving invariant

```
definition monotonicUpdates ::"('pl, 'ua, 'qa, 'r) stateBasedType
                                ⇒ ('pl,'ua) crdtInvariant ⇒ bool" where
"monotonicUpdates crdt Inv =
      (∀UH args pl r. Inv UH pl ⟶ pl ≤ [crdt] t_update crdt args r pl)"
```

Figure 16: Monotonic updates property

1. The merge operation is commutative.
2. The merge operation computes an upper bound, so merging x and y yields a value which is greater than x and greater than y.
3. The merge operation computes a minimal value. If there is a value z which is an upper bound, then it must at least be as big as the result of merging x and y.

A formalization of the semilattice properties with invariants taken into account, can be found in figure 17.

Assuming all the above properties hold, it is now possible to prove the desired convergence properties:

Lemma 4.2.4
Let s be the state after executing some steps in the system. Let H_1 and H_2 be subsets of the history H of s so that the supremum of all version vectors from H_1 is less than or equal to the supremum of all version vectors from H_2. Then merging all payloads from H_1 yields a payload which is less than or equal to the payload yielded by merging all payloads from H_2. See figure 18 for the statement of the lemma in Isabelle.

For a shorter notation we use $\bigsqcup_v S$ for the supremum of the versions in S and similarly $\bigsqcup_{pl} S$ for the supremum of the payloads in S. Then the main statement of the lemma can be expressed as follows:

$\forall_{H_1,H_2} H_1 \subseteq H \wedge H_2 \subseteq H \wedge \bigsqcup_v H_1 \leq \bigsqcup_v H_2 \rightarrow \bigsqcup_{pl} H_1 \leq \bigsqcup_{pl} H_2$

```
definition "compareReflexive crdt Inv ≡ (∀UH x. Inv UH x ⟶
  x ≤ [crdt] x)"
definition "compareTransitive crdt Inv ≡ (∀UH x y z. Inv UH x ∧ Inv UH y ∧ Inv UH z ∧
  x ≤ [crdt] y ∧ y ≤ [crdt] z ⟶ x ≤ [crdt] z)"
definition "compareAntisym crdt Inv≡ (∀UH x y. Inv UH x ∧ Inv UH y ∧
  x≤ [crdt]y ∧ y≤ [crdt]x ⟶ x=y)"
definition "mergeCommute crdt Inv = (∀UH x y. Inv UH x ∧ Inv UH y ⟶
  x ⊔ [crdt] y = y ⊔ [crdt] x)"
definition "mergeUpperBound crdt Inv ≡ (∀UH x y. Inv UH x ∧ Inv UH y ⟶
  x ≤ [crdt] x ⊔ [crdt] y)"
definition "mergeLeastUpperBound crdt Inv ≡
  (∀UH x y z. Inv UH x ∧ Inv UH y ∧ Inv UH z ∧
      x≤ [crdt]z ∧ y≤ [crdt]z ⟶ x ⊔ [crdt] y ≤ [crdt] z)"
```

Figure 17: Semilattice properties

```
definition historyInvariant :: "('pl, 'ua, 'qa, 'r, 'g) stateBasedType ⇒ 'pl history ⇒ bool"
  where
"historyInvariant crdt H = (
   ∀H1 H2. H1 ⊆ H ∧ H2 ⊆ H
   ∧ supSet (fst ‘ H1) ≤ supSet (fst ‘ H2)
   ⟶ mergeAll crdt (snd ‘ H1) ≤ [crdt] mergeAll crdt (snd ‘ H2))"


lemma stepsToHistory: "⟦
steps crdt tr (initialState crdt) s;
crdtProperties crdt
⟧ ⟹ historyInvariant crdt (history s)"
```

Figure 18: Convergence Lemma

Proof. By induction over the steps.

Initial state In the initial state the history contains only one element: $([0, \ldots, 0], t_{initial}(crdt))$. So the claim follows by the reflexivity of the partial order.

Update After an update operation, one new element is in the history.

Let e_{new} be the new element in the history and e_{old} the element before the update.

Let H_1 and H_2 be two subsets of the new history with $\bigsqcup_v H_1 \leq \bigsqcup_v H_2$. There are four cases to be considered:

Case 1: The new element is neither in H_1 nor H_2. In this case the claim follows directly from the induction hypothesis.

Case 2: The new element is only in H_1.

Then we must have $\bigsqcup_v H_1 > \bigsqcup_v H_2$, because the versions in H_1 include e_{new}. The version vector of the new element is higher than all other version vectors in the history. This is a contradiction to the assumption that $\bigsqcup_v H_1 \leq \bigsqcup_v H_2$.

Case 3: The new element is only in H_2.

Consider the two sets $H_1' = H_1$ and $H_2' = H_2 \backslash \{e_{new}\} \cup \{e_{old}\}$.

Then $\bigsqcup_v H_1' \leq \bigsqcup_v H_2'$, because the version vector is only decreased in the local component, where the old version vector was already maximal.

So by the induction hypothesis, we get that $\bigsqcup_{pl} H_1' \leq \bigsqcup_{pl} H_2'$. Because updates are monotonically increasing we also have $\bigsqcup_{pl} H_2' \leq \bigsqcup_{pl} H_2$. The claim follows by transitivity.

Case 4: The new element is in H_1 and H_2.

Similar to case 3, consider the two sets

$$H_1' = H_1 \backslash \{e_{new}\} \cup \{e_{old}\}$$
$$H_2' = H_2 \backslash \{e_{new}\} \cup \{e_{old}\}$$

Again, we have $\bigsqcup_v H_1' \leq \bigsqcup_v H_2'$ and thus $\bigsqcup_{pl} H_1' \leq \bigsqcup_{pl} H_2'$.

Because the merge operation is associative, the element e_{old} can be extracted from the set:

$$\bigsqcup_{pl} H_1' = e_{old} \sqcup \bigsqcup_{pl} (H_1 \backslash \{e_{new}\}) \leq e_{old} \sqcup \bigsqcup_{pl} (H_2 \backslash \{e_{new}\}) = \bigsqcup_{pl} H_2'$$

By the monotonicity of merges we can replace e_{old} by the bigger e_{new} and preserve the order:

$$e_{new} \sqcup \bigsqcup_{pl} (H_1 \backslash \{e_{new}\}) \leq e_{new} \sqcup \bigsqcup_{pl} (H_2 \backslash \{e_{new}\})$$

By using associativity of merges we can move the element e_{new} inside the overall supremum:

$$\bigsqcup_{pl} H_1 \leq \bigsqcup_{pl} H_2$$

This finishes the case for updates.

Merge After a merge operation there is one new element e_{new} in the history, which is the supremum or merge of two elements e_a and e_b from the old history.

If e_{new} is in H_1, we can take $H_1' = H_1 \backslash \{e_{new}\} \cup \{e_a, e_b\}$. Then $\bigsqcup_v H_1' = \bigsqcup_v H_1$ and $\bigsqcup_{pl} H_1' = \bigsqcup_{pl} H_1$ by associativity. If e_{new} is not in H_1, let $H_1' = H_1$.

The same can be done for H_2.

Because we have found equivalent sets to H_1 and H_2 without the new element, the claim holds by the induction hypothesis.

□

Theorem 4.2.5 (Convergence of CRDTs)
If a CRDT is invariant preserving, has monotonic updates and satisfies the semilattice properties, then it converges (as defined in figure 13).

Proof. Let e_1 and e_2 be elements from the history with $version(e_1) \leq version(e_2)$.

Lemma 4.2.4 can be instantiated with $H_1 = \{e_1\}$ and $H_2 = \{e_2\}$. This yields $payload(e_1) \leq payload(e_2)$.

Thus, if $version(e_1) = version(e_2)$, we get $payload(e_1) = payload(e_2)$ by the antisymmetry of the ordering. □

5. Specification of CRDTs

While the specification of sequential data types is relatively straight forward, it is debatable what the best way to specify parallel, distributed data types is. Operations on sequential data types are usually described by pre- and post-conditions[4]. When a sequence of operations is given, the pre- and post-conditions can be chained together and pre- and post-conditions for the whole sequence can be obtained. This approach cannot be directly transferred to parallel, distributed data types:

1. In general there is no sequence of operations but instead a directed acyclic graph (DAG) of operations, where some operations can happen in parallel.
2. Pre- and post-conditions could be defined with respect to local states or the global state.
3. Distributed data types have mechanisms for synchronization. Specifications can choose to explicitly specify those mechanisms or to keep it implicit.

The remainder of this chapter presents different approaches to specify CRDTs using the examples of the counter and the observed-removed set, which were introduced in section 2.2.

Notations. The following notations and conventions are used to describe the different specification techniques: The variable s always refers to a state of the whole system. When s is indexed this refers to the state on a single replica. For example s_{r_1} refers to the state of replica r_1. An squiggled arrow with operations written on top denotes, that the system changes from the left state to the right state by executing the operations on top of the arrow. For example $s \overset{add(x)}{\rightsquigarrow} s'$ means that the system goes from state s to state s' by executing a add-operation. A capital S denotes the set of all states reachable from the initial state and S_l is the set of all reachable local states. Operations are ordered by a happens-before relation which is denoted by $\prec$.

5.1. Separated sequential and concurrent specifications

In [11] specifications consist of two parts: a sequential specification and a concurrent specification. Both parts use a notation with pre- and post-conditions to specify the outcome of operations. The sequential part of the specification is equivalent to specifications of sequential data types. The parallel part then specifies how the data type behaves when several operations are executed in parallel.

A sequential specification is 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. This definition can be written as follows:

$$\forall s, s' \in S_l \bullet (P(s) \land s \overset{op}{\rightsquigarrow} s') \rightarrow Q(s')$$

The parallel specification is also written in a similar style, but instead of just one operation it considers several operations executed in parallel. The general form is $\{P\}op_1 \parallel op_2 \parallel \cdots \parallel op_n\{Q\}$. The rough idea 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.

When formalizing the idea of parallel specifications, it has to be refined on which states the pre- and post-condition should hold. For the pre-condition there are some possibilities:

1. The pre-condition refers to a local state of one replica, which also is the state on which all operations are performed.
2. The pre-condition refers to each local states of every affected replicas.
3. The pre-condition refers to the merged state of all affected replicas.

For the post-condition the only reasonable state is the merged state of all the local states after executing the parallel operations.

In the following definition of the semantics for $\{P\}op_1 \parallel op_2 \parallel \cdots \parallel op_n\{Q\}$, the pre-condition has to hold for a local state s. Then starting from this state s, all operations $op_1, op_2, \ldots, op_n$ are executed in parallel, resulting in states $s_1, \ldots, s_n$. Then, when all these states are merged into a state s', the post-condition Q has to hold for s':

$$\forall s, s_1, \ldots, s_n, s' \in S_l \bullet (P(s) \land (\forall_i\ s \overset{op_i}{\rightsquigarrow} s_i) \land (s \overset{merge(s_1);\ldots;merge(s_n)}{\rightsquigarrow} s')) \rightarrow Q(s')$$

5.1.1. Principle of permutation equivalence

The principle of permutation equivalence restricts the choices which can be taken by concurrent operations. It is not necessary for a CRDT to adhere to this principle, but it is considered good practice. The principle states that if all possible sequential executions of a set of operations yields a common post-condition, then that post-condition should also hold when the operations are executed in parallel.[11]

For two operations the principle can be expressed as:

$$\{P\}op_1; op_2\{Q\} \land \{P\}op_2; op_1\{Q\} \longrightarrow \{P\}op_1 \parallel op_2\{Q\}$$

5.1.2. Example: Counter

Sequential specification:

$$\{val() = i\} \quad add(x) \quad \{val() = i + x\}$$

Concurrent specification:

$$\{val() = i\} \quad add(x_1) \parallel \cdots \parallel add(x_n) \quad \{val() = i + x_1 + \cdots + x_n\}$$

The concurrent specification of the counter is already determined by the principle of permutation equivalence. Because all operations commute, the post-condition in the concurrent specification cannot be chosen differently. Therefore the sequential specification alone would be sufficient to specify the Counter.

5.1.3. Example: Observed-Remove Set

Sequential specification:

$$\begin{array}{rcl}
\{true\} & add(e) & \{e \in S\} \\
\{e \in S\} & add(f) & \{e \in S\} \\
\{e \notin S \wedge e \neq f\} & add(f) & \{e \notin S\} \\
\{true\} & remove(e) & \{e \notin S\} \\
\{e \in S \wedge e \neq f\} & remove(f) & \{e \in S\} \\
\{e \notin S\} & remove(f) & \{e \notin S\}
\end{array}$$

The principle of permutation equivalence already fixes the semantics for parallel operations which are independent from each other. For example we have:

$$\begin{array}{rcl}
\{true\} & add(e) \parallel add(f) & \{e \in S \wedge f \in S\} \\
\{e \neq f\} & add(e) \parallel remove(f) & \{e \in S \wedge f \notin S\} \\
& \cdots &
\end{array}$$

The only interesting case is when two operations are conflicting, which only is the case if an element is added and removed in parallel. In this case the OR-set will let the add operation win, so the element will be in the set.

$$\{true\} \quad add(e) \parallel remove(e) \quad \{e \in S\}$$

This can also be generalized to more than two parallel operations:

$$\begin{array}{lll}
\{S = S_{pre}\} & op_1 \parallel \cdots \parallel op_n & \{e \in S \ \leftrightarrow ((\exists_i \bullet op_i = add(e)) \\
& & \vee (e \in S_{pre} \wedge \forall_i \bullet op_i \neq remove(e))\}
\end{array}$$

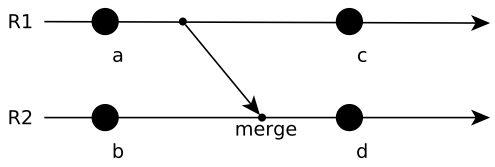

Figure 19: Example Execution

5.1.4. Discussion

As can be seen in the examples, this specification technique is relatively easy to understand. The sequential semantics is similar to specifications of sequential data types. When the principle of permutation equivalence is used, the concurrent specification only has to be given for some of the cases. For those cases it is then very easy to see how conflicts are resolved. For example in the OR-set it is very easy to see that an add operation wins, when it conflicts with a remove operation. However, writing down the concurrent specification for the general case of more than two parallel operations is not so easy.

A major drawback of this specification approach is, that the specification structure does not reflect the actual structure of executions. In the specifications there are only sequences of operations and sets of parallel operations. But in general the happens-before relation on operations is a directed acyclic graph. Such a graph cannot be described as just a combination of parallel operations and sequences.

For example the execution in figure 19 cannot be expressed using such a combination. When just looking at the operations a, b and d it can still be expressed as $a; (b \parallel d)$. But it is not possible to add operation c to this expression because it would have to be parallel to a and b and still be before d.

It is possible to specify a data type which just records the operations:

$$\begin{array}{lll} \{val() = x\} & op(y) & \{val() = x; y\} \\ \{val() = x\} & op(y_1) \parallel \cdots \parallel op(y_n) & \{val() = x; (y1 \parallel \cdots \parallel y_n)\} \end{array}$$

However such a data type cannot exist because of executions as in figure 19. As shown by this example, it is in general not possible to determine the outcome of an execution by just using the specification given in the form of sequential and concurrent specifications.

5.2. Explicit Specification of Merge

Another approach is to consider a merge as an explicit operation and replace the concurrent specification in the previous approach with a specification of the merge operation. When trying to specify the merge operation for the examples it becomes clear, that pre-conditions on the local states alone are insufficient for precisely determining the

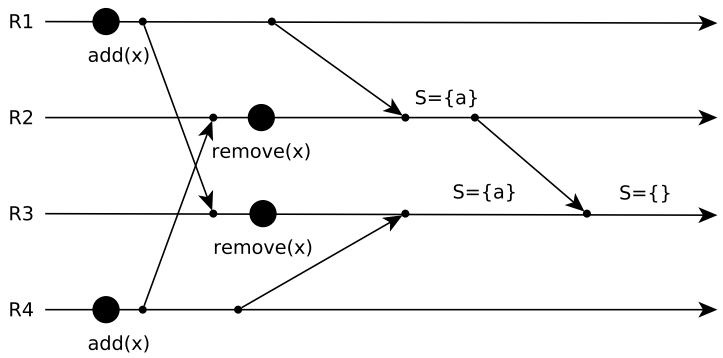

Figure 20: Before the last merge operation a is in both sets. After the merge a is gone.

post-condition. For example in the case of an increase-only counter the postcondition can only specify a range of possible values:

$$\{c_1.val() = x \wedge c_2.val() = y\} \quad c = merge(c_1, c_2) \quad \{max(x, y) \leq c.val() \leq x + y\}$$

In the case of a counter, which can also decrease the value, no meaningful post-condition can be given. For the OR-set the situation is similar. For example it does not even hold that if an element is in both sets before the merge, it is still in the set after the merge (see execution in figure 20).

The examples show, that more information is needed in the pre-condition. One possible extension would be to base the pre-condition not only on the state observable via the normal queries of the data type, but also by queries which expose more details. For example the increment-only counter could be specified if there was a query which returned the number of increments on each replica. This would have the disadvantage of making the specification less abstract and closer to an implementation.

An other possible extension would be to add a third local state $s_{1\sqcap 2}$ to the pre-condition. The state $s_{1\sqcap 2}$ is the state a replica would have, if it had seen exactly all the operations which the states s_1 and s_2 have both seen. This idea is visualized in figure 21, which can be seen as an other visualization of the execution in figure 19. The event a has been observed in both states s_1 and s_2. The other events have only be observed in one of the states. The state $s_{1\sqcap 2}$ would then be a state which has only observed operation a.

With this addition, it is possible to specify the merge of two counter states like this:

$$\left\{ \begin{array}{c} c_1.val() = x \\ c_2.val() = y \\ c_{1\sqcap 2}.val() = z \end{array} \right\} \quad c = merge(c_1, c_2) \quad \{c.val() = x + y - z\}$$

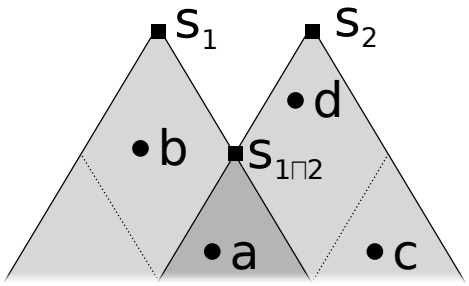

Figure 21: Meet State

For specifying the OR-set, it is necessary to extend the interface of the set. It must be possible to distinguish two different add-operations. Assuming every add-operation assigns a unique id i to the added element, and it can be tested if such an element is in the set with "$x_i \in S$", the merge operation can be specified as follows:

$$\begin{aligned}
\{x_i \in S_1 \wedge x_i \in S_2 \wedge x_i \in S_{1\sqcap 2}\} \quad & S = merge(S_1, S_2) \quad \{x_i \in S\} \\
\{(x_i \notin S_1 \vee x_i \notin S_2) \wedge x_i \in S_{1\sqcap 2}\} \quad & S = merge(S_1, S_2) \quad \{x_i \notin S\} \\
\{(x_i \in S_1 \vee x_i \in S_2) \wedge x_i \notin S_{1\sqcap 2}\} \quad & S = merge(S_1, S_2) \quad \{x_i \in S\} \\
\{x_i \notin S_1 \wedge x_i \notin S_2 \wedge x_i \notin S_{1\sqcap 2}\} \quad & S = merge(S_1, S_2) \quad \{x_i \notin S\}
\end{aligned}$$

5.2.1. Discussion

In both examples the specification is based on the changes made with respect to the state $S_{1\sqcap 2}$. The state $S_{1\sqcap 2}$ can be calculated from a given execution but it is not necessarily a state which as actually reached. For example the last merge in the execution shown in figure 22, merges payloads with version $[2, 1]$ and $[1, 2]$. So the version vector representing the common pre-state would be $[1, 1]$. There is no point, where the execution reaches this version vector. However, it would still be possible to compute the conditions, which would hold at this version, by recursively applying the specification.

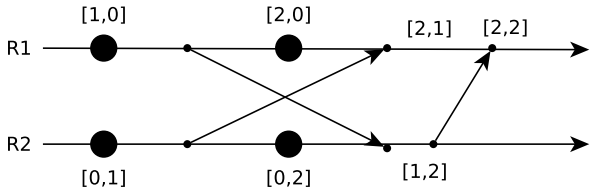

Figure 22: Example execution

But this is still one point which makes this specification technique less intuitive than the previous one. Another point is, that the focus is on the merge-operation instead of

the actual data type operations.

The advantage of this approach is, that it allows to specify the behavior completely. There are no executions where a specification of this kind cannot be applied.

5.3. Specification based on Update History

The update history of an execution is a tuple $(E, \prec)$, where E is the set of events in the execution and $\prec$ is the happens-before relation between operations. An event has a replica on which it occurred and an update-operation. Merges are not an explicit part of the update history, but they influence the happens-before relation.

The happens-before relation is a partial order. All events on the same replica are ordered by time. A merge introduces a happens-before ordering between different replicas. In the system model the happens-before relation is described by version vectors.

Note that there can be several executions which have the same update history. For example the two executions shown in figure 23 both have the same update history but different merge events. The first merge has no effect on the happens-before relation between updates. When the merge operation is associative, commutative and idempotent the different merge events will still yield the same result, for the same reason version vectors are the same.

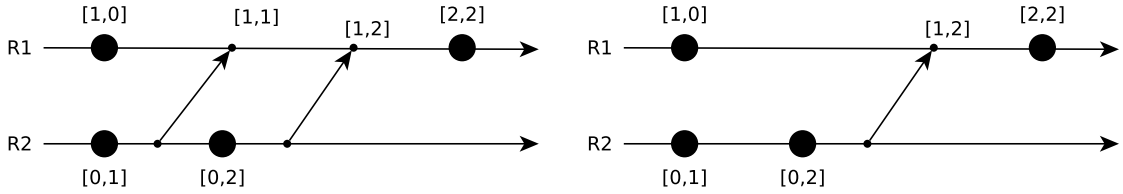

Figure 23: Same update history but different merge actions

Now the specification can be expressed based on the visible update history. For example the counter can be specified as just the sum of all add-events.

$$c.val() = \sum_{e \in E,\, op(e) = add(x)} x$$

Here E refers to all events visible to the counter c and $op(e)$ is the operation executed in event e.

The OR-set can be specified by stating that an element is in the set when there is an add operation for that element and no remove operation comes after the add operation:

$$x \in S \leftrightarrow (\exists_{e_a \in E} \bullet op(e_a) = add(x) \wedge (\nexists_{e_r \in E} \bullet e_a \prec e_r \wedge op(e_r) = remove(x)))$$

5.3.1. Discussion

This specification technique has a very straight-forward semantics. One can take an execution, compute the events and happens-before relation and then apply the specification. It would even be possible to generate a data type implementation from the specification. The data type would store all the events and maintain a happens-before relation and the query functions would be equivalent to the specification.

A disadvantage of this technique is, that it is based on the entire execution, whereas the other techniques are better suited for reasoning about the effect of a single operation or a part of an execution when knowing the pre-state of the system. This might be relevant when using a CRDT specification to verify the behavior of an application using CRDTs. For example an application could execute an update only after checking some conditions, so while the application code is not aware of the entire histories of the used CRDT objects, it can know something about the state of the objects before executing the updates.

But while specifications based on pre-conditions might be better suited for verifying applications, a specification which is based on the complete update history is a complete specification and so it should be possible to derive the validity of other specifications from the validity of the former specification.

For this reason I decided to use this kind of specification as the basis for describing and verifying the behavior of CRDTs.

5.3.2. Realization in Isabelle

This section describes how the specification technique was formalized in Isabelle/HOL. First of all a type for the update history is required. As seen in the previous section, the update history has to capture all update operations and the happens-before relation between those operations. In a mathematical model this would usually be modeled as a tuple $(E, \prec)$, where E is the set of events (update-operations) and $\prec$ is the happens-before relation between events. And an event is then just a tuple $(v, r, args)$, where v is the version vector describing the time of the event, r is the replica-ID of the replica on which the operation was executed, and $args$ are the arguments to the update function. The version v is a unique identifier for an event. If the event was just a tuple $(r, args)$, then it would not be possible to distinguish two update operations with the same arguments on the same replica. The version vector also captures the happens-before relation between events, so it is only necessary to explicitly have a set E instead of the tuple $(E, \prec)$.

When the events are stored in a set, hardly any properties of the update history are reflected in the data structure and have to be expressed as additional constraints. For example the uniqueness of version vectors would be clear if the data structure for events was a map from versions to $(r, args)$ tuples. When using a set this property has to be captured elsewhere. So my first approach was to use a map for storing the events. However some other properties like the compatibility of update histories was not so nice with this structure, as it does not reflect the concept of replicas very well. In many

```
type_synonym 'ua updateHistory = "replicaId ⇒ (versionVector × 'ua) list"
```

Figure 24: Update History Type

properties it was necessary to separate the events by replica. Therefore I have chosen the structure shown in figure 24 for the update history. It is a mapping from replicas to a list of operations, which are represented by $(v, args)$ pairs. The list is sorted so that the earliest operations are at the beginning of the list. Lists have the advantage, that it is easier to determine if one sequence of operations is a prefix of an other. There also is a nice relation between the position of an operation in the list and the local component of its version vector, which also is a property required later: For the nth operation in the list it holds, that $v_r = n$.

As stated earlier, the update history can be computed from the trace of an execution. The functions for doing this are shown in figure 25. First there is a function replicaVersion, which calculates the version vector at each replica for a given trace. The rules are equivalent to the rules used in the operational semantics of the system model. The function updateHistoryH then uses this function to compute the update history. The function just appends each operation in the trace to the list for the respective replica, using the version as calculated by the replicaVersion function. There is also a function visibleUpdateHistory, which returns only the part of the update history visible at a certain version. This is what the specifications are based on.

A specification is a function which determines the result for given query arguments ('qa) and a given update History containing the visible updates. The type of the function is shown in figure 26. It would have been possible to use a relation or a partial function instead of the total function. This would have allowed specifications to describe the behavior only partially, whereas a function always has to describe the full behavior. This could be useful to describe a data type in a more abstract way. For example it would be possible to have a specification which holds for different kind of sets. Most sets satisfy the property, that an element is in the set if it has been added to the set and there is no remove operation which happens after the add operation or in parallel to it.

However, it is usually easy to go from a total specification to a partial specification and therefore only the total specifications were considered in this thesis.

```
fun replicaVersion :: "('ua, 'pl) trace ⇒ replicaId ⇒ versionVector" where
  "replicaVersion EmptyTrace r = vvZero"
| "replicaVersion (Trace tr (Update r args)) ra = (
      if r=ra then incVV r (replicaVersion tr r)
      else replicaVersion tr ra)"
| "replicaVersion (Trace tr (MergePayload r vv pl)) ra = (
      if r=ra then sup vv (replicaVersion tr r)
      else replicaVersion tr ra)"


definition initialUpdateHistory :: "'ua updateHistory" where
"initialUpdateHistory = (λr. [])"

fun updateHistoryH :: "('ua, 'pl) trace ⇒ 'ua updateHistory" where
"updateHistoryH tr = (case tr of
    EmptyTrace ⇒ initialUpdateHistory
  | (Trace tra (Update r args)) ⇒ (updateHistoryH tra)(
                          r := updateHistoryH tra r @ [(replicaVersion tr r, args)])
  | (Trace tra (MergePayload r vv pl)) ⇒ updateHistoryH tra
)"

definition "updateHistory tr = (updateHistoryH tr)"

definition updateHistoryFilterInvisible :: "'ua updateHistory ⇒ versionVector ⇒ 'ua
  updateHistory" where
"updateHistoryFilterInvisible H v = (λr. filter (λ(vv,args). vv≤ v) (H r))"

definition visibleUpdateHistory :: "('ua, 'pl) trace ⇒ versionVector ⇒ 'ua updateHistory"
  where
"visibleUpdateHistory tr v = updateHistoryFilterInvisible (updateHistory tr) v"
```

Figure 25: Update History Calculation

```
type_synonym ('ua,'qa,'r) crdtSpecification = "'ua updateHistory ⇒ 'qa ⇒ 'r"
```

Figure 26: Specification

6. Verifying CRDT behavior

In order to verify that a given CRDT implementation is a valid implementation of a specification, where the specification is given in the form introduced in 5.3, the property defined in figure 27 has to be shown.

When a CRDT object reaches state s from the initial state, then all possible queries on all replicas should return the result given by the specification, where the specification function is based on the visible update history of the replica and the arguments to the query.

Proving this property could be done by an induction over the traces. The induction start would handle the case of an empty trace and in the induction step one would have to handle the cases for updates and merge operations. However when using this induction scheme, there are some steps which occur for every data type:

In the case of update operations it is always a single replica which is affected by the update. For the other replicas it has to be explicitly shown, that the visible update history does not change and that the payload of the replica is not changed. Also it is tedious to work with the filter function, which filters out invisible events from the update history in every step.

In the case of a merge a current payload and a payload from the past are merged together. This means that the induction hypothesis must also hold for the old payloads and in every step it has to be shown that the step does not break the invariant. This is trivial to show, as a step cannot change older payloads or the visible update history for that past time, but nevertheless it has to be proven when using a tool like Isabelle. Another point is that when merging update histories, there are some properties of the update history which are used in the verification proofs of more than one data type. Therefore it makes sense to extract properties like this in a theory which can be used for every CRDT.

The theorem showCrdtSpecificationValid shown in figure 28 shows an induction scheme which is more specialized to the situation. The theorem uses an invariant Inv, which relates payloads with the part of the update history relevant for that payload, the visible update history. After finding such an invariant, there are then four basic properties which have to be proven to show that the CRDT implementation satisfies the specification:

```
definition crdtSpecificationValid :: "('pl, 'ua, 'qa, 'r) stateBasedType ⇒ ('ua,'qa,'r)
  crdtSpecification ⇒ bool" where
"crdtSpecificationValid crdt spec = (∀tr s r qa.
  steps crdt tr (initialState crdt) s ⟶
    t_query crdt qa (replicaPayload s r) =
    spec (visibleUpdateHistory tr (replicaVV s r)) qa)"
```

Figure 27: Update History Calculation

1. invariantImpliesSpec:

   The invariant must imply the specification. When the invariant holds for a valid update history H and a payload pl, then performing a query on that payload should return the result as given by the specification, where the specification does only depend on the update history H and not on the payload pl.

   A detailed definition of when an update history is considered valid is given in section 6.1.

2. updateHistoryInvariant_initial:

   The invariant has to hold for the empty update history and the initial payload.

3. updateHistoryInvariant_update:

   After an update operation, the invariant has to hold for the new update history and the new payload. The new payload is a successor of the old payoad pl, which has a version v.

   The update history is changed by appending the new update event to the local replica r. The version of the event is calculated by increasing the version v by one for the local replica (incVV r v). The new payload is calculated by applying the update function of the CRDT to the old payload pl.

   The important difference to the induction over traces is, that only the relevant part of the update history is considered. The version v of the old payload is the supremum of all versions in the history (versionIsTop). The version v does not necessarily occur in the update history, because the payload might not be generated by an update operation, but by one or more merge operations. So it is only possible to say, that it is the supremum of all the other versions before. From this it also follows, that the version of the new update is strictly greater in the component of the local replica, than all other updates.

4. updateHistoryInvariant_merge:

   After a merge operation, the invariant has to hold for the merged update history and the merged payloads. The merged payload is simply calculated by the merge function of the CRDT. The merged update history is calculated by a function historyUnion, which just takes the longer list of updates for each replica. One assumption for the merge is, that the two histories are consistent. This means that on every replica, one of the local update histories is a prefix of the other. Because of this, the historyUnion function preserves the contents of both lists and the history only grows.

   Like in the case for updates, the assumption of valid update histories is also included in the merge case.

```
definition updateHistoryInvariant_initial where
"updateHistoryInvariant_initial crdt Inv = (Inv (λr. []) (t_initial crdt))"

definition updateHistoryInvariant_merge where
"updateHistoryInvariant_merge crdt Inv = (∀H1 pl1 H2 pl2.
    validUpdateHistory H1
  ∧ validUpdateHistory H2
  ∧ Inv H1 pl1
  ∧ Inv H2 pl2
  ∧ consistentHistories H1 H2
    ⟶ Inv (historyUnion H1 H2) (t_merge crdt pl1 pl2))"

definition updateHistoryInvariant_update where
"updateHistoryInvariant_update crdt Inv = (∀H pl r v args.
    validUpdateHistory H
  ∧ Inv H pl
  ∧ versionIsTop v H
    ⟶ Inv (H(r := (H r)@[(incVV r v, args)])) (t_update crdt args r pl))"

definition updateHistoryInvariant_all where
"updateHistoryInvariant_all crdt Inv = (
    updateHistoryInvariant_initial crdt Inv
  ∧ updateHistoryInvariant_merge crdt Inv
  ∧ updateHistoryInvariant_update crdt Inv)"

definition invariantImpliesSpec where
"invariantImpliesSpec crdt Inv spec = (∀H pl qa.
    validUpdateHistory H ∧ Inv H pl
      ⟶ t_query crdt qa pl = spec H qa)"

theorem showCrdtSpecificationValid: "⟦
invariantImpliesSpec crdt Inv spec;
updateHistoryInvariant_all crdt Inv
⟧ ⟹ crdtSpecificationValid crdt spec"
```

Figure 28: Induction over histories

6.1. Valid update histories

For the proofs it is important to have a good characterization of valid update histories, so that it is easy to work with them. The defined properties for a valid update history are listed in figure 29. These properties hold for every update history which can be generated by a valid trace, where a trace tr is considered valid if there is a state s, such that s can be reached from the initial state by executing the trace tr. The converse does probably not hold, so there might be cases where the defined properties are too weak for verifying a CRDT. However, these properties were sufficient for verifying the CRDTs considered in the case studies.

The first property validUpdateHistory_versionVectorR describes the relation between the position of an update operation in the update operation list of a replica and the component of the version vector of that update for the same replica: The first update operation on replica r has v»r=1, the second update has v»r=2, and so on. If (v,args) is at position i (positions are zero based) in the list of update operations on replica r (H r ! i = (v,args)), then the component r of the version v will be one higher than i (v»r=i+1).

The second property validUpdateHistory_localMax describes the fact, that there cannot be any update in the update history with a version v, where v»r is higher than the number of updates on replica r.

The third property validUpdateHistory_monotonicSame describes the fact, that version vectors of updates on the same replica increase monotonically. So if v1 and v2 are versions of updates on the same replica and v1»r<v2»r, then also v1<v2.

The fourth property validUpdateHistory_monotonicOther is similar to the third property, but describes the relation to versions of updates on other replicas. If v1 is the version of an update on replica r1 and v2 the version of an update on some other replica and v1»r1≤ v2»r1, then v1<v2.

6.2. Consistent update histories

Two update histories are consistent, when they can be unified to one update history. This is the case when for each replica the list of updates from one update history is a prefix of the list of updates from the other. If this is the case, then the historyUnion can just take the longer list for each replica and thus the result grows monotonically. If at one replica the prefix relation would not hold, this would mean that there is an element in the list, where the two histories have seen different updates at the same position. But this cannot happen if the two update history are derived from the same trace.

The Isabelle definitions of the historyUnion and consistentHistories are shown in figure 30.

6.3. Verification by showing equivalence

Another approach to verifying the correctness of an implementation is to show that it behaves equivalent to an other CRDT. This approach can be used when an easier version

```
definition validUpdateHistory_versionVectorR :: "'ua updateHistory ⇒ bool" where
"validUpdateHistory_versionVectorR H = (∀v r args i.
      i<length(H r) ∧ H r ! i = (v,args) ⟶ v»r = Suc i)"

definition validUpdateHistory_localMax :: "'ua updateHistory ⇒ bool" where
"validUpdateHistory_localMax H = (∀r v.
      v∈ allVersions H ⟶ (length (H r) ≥ v»r))"

definition validUpdateHistory_monotonicSame :: "'ua updateHistory ⇒ bool" where
"validUpdateHistory_monotonicSame H = (∀r v1 v2 y1 y2.
      (v1,y1) ∈ set (H r) ∧ (v2,y2) ∈ set (H r) ∧ v1»r<v2»r⟶ v1<v2)"

definition validUpdateHistory_monotonicOther :: "'ua updateHistory ⇒ bool" where
"validUpdateHistory_monotonicOther H = (∀r1 r2 v1 v2 y1 y2.
      r1≠r2 ∧ (v1,y1) ∈ set (H r1) ∧ (v2,y2) ∈ set (H r2) ∧ v1»r1≤ v2»r1
        ⟶ v1<v2)"


definition validUpdateHistory :: "'ua updateHistory ⇒ bool" where
"validUpdateHistory H = (
    validUpdateHistory_versionVectorR H
  ∧ validUpdateHistory_localMax H
  ∧ validUpdateHistory_monotonicSame H
  ∧ validUpdateHistory_monotonicOther H
)"
```

Figure 29: Properties of a valid update history

```
definition historyUnion :: "'ua updateHistory ⇒ 'ua updateHistory ⇒ 'ua updateHistory"
  where
"historyUnion H1 H2 = (λr.
            if length (H1 r) ≤ length (H2 r) then H2 r else H1 r)"

definition consistentHistories :: "'ua updateHistory ⇒ 'ua updateHistory ⇒ bool" where
"consistentHistories H1 H2 = (∀r.
        isPrefix (H1 r) (H2 r)
      ∨ isPrefix (H2 r) (H1 r))"
```

Figure 30: Consistent update histories

of a CRDT is already verified and an optimized version of the CRDT behaves in a similar way. Behaving in a similar way means, that it is possible to define a coupling between the payloads of the two CRDTs. The properties of this coupling are shown in figure 31. The coupling must hold in the initial state and it must be preserved by update and merge operations, and for two coupled payloads, the same query must return the same result.

The theorem simulation in figure 31 states that when an crdtA fulfills a specification spec and there is a coupling between crdtA and crdtB, then crdtB also fulfills the same specification. This theorem can be proven by an induction over the traces for crdtA. Because the traces include the payloads in the merge actions, traces for crdtA are not applicable for crdtB, but there always exists an equivalent trace where the payloads in the trace are replaced with coupled payloads.

The idea of simulation could probably be expanded to more than one CRDT. For example a PN-counter could be simulated by two increase-only-counters. This could make it easier to verify CRDTs which are compositions of other CRDTs. Because of time constraints this approach was not examined further.

```
definition couplingInitial where
"couplingInitial Inv crdtA crdtB = Inv (t_initial crdtA) (t_initial crdtB)"

definition couplingUpdate where
"couplingUpdate Inv crdtA crdtB = (∀pIA pIB args r.
    Inv pIA pIB ⟶
        Inv (t_update crdtA args r pIA) (t_update crdtB args r pIB))"

definition couplingMerge where
"couplingMerge Inv crdtA crdtB = (∀pIA1 pIB1 pIA2 pIB2.
    Inv pIA1 pIB1 ∧ Inv pIA2 pIB2 ⟶
        Inv (t_merge crdtA pIA1 pIA2) (t_merge crdtB pIB1 pIB2))"

definition couplingQuery where
"couplingQuery Inv crdtA crdtB = (∀pIA pIB args.
    Inv pIA pIB ⟶ t_query crdtA args pIA = t_query crdtB args pIB)"

definition coupling where
"coupling Inv crdtA crdtB = (
    couplingInitial Inv crdtA crdtB
  ∧ couplingUpdate Inv crdtA crdtB
  ∧ couplingMerge Inv crdtA crdtB
)"

theorem simulation: "⟦
crdtSpecificationValid crdtA spec;
couplingQuery Inv crdtB crdtA;
coupling Inv crdtB crdtA
⟧ ⟹ crdtSpecificationValid crdtB spec"
```

Figure 31: Showing validity by simulation

7. Case Studies

In the case studies the verification approaches from the previous chapters are applied to several examples from the literature. For each CRDT the semilattice properties are verified first, and then the behavioral specifications are verified. Those two verification tasks are handled independently, to see what the difference in effort is for the two. It would have been sufficient to show that the implementations satisfy the behavioral specifications, as this also implies that the CRDT converges. Since the specifications are total functions of the update history, and version vectors are equal if and only if visible update histories are equal, this implies that two replicas with the same version vector will return the same results for queries.

7.1. Increment-Only Counter

The implementation of the Increment-Only Counter is given in figure 32. This implementation is a translation of specification 6 from [12] to Isabelle.

The payload of the counter is just a version vector which stores the number of increments on each replica. The increment operation increases the version vector at the local replica. The compare and merge functions are just taken from the version vector which already form a semilattice. Therefore showing the convergence of this CRDT is trivial.

The counter provides one query function which should return the total number of increments seen by the replica. Since all updates are increments, this is captured in the following specification:

```
definition counterSpec :: "(unit,unit,nat) crdtSpecification" where
"counterSpec H q = card(allUpdates H)"
```

- allUpdates H is the set of all updates
- card is the cardinality of the set

This specification can be proven by using the following invariant:

```
definition counterInvariant where
"counterInvariant H pl = (∀r. pl»r = length (H r))"
```

- H r is a list containing all updates on replica r
- pl»r is the value of component r in the version vector pl

It is straight forward to show that the invariant is preserved, but proving that the invariant implies the specification cannot be done automatically by Isabelle.

```
definition increment :: "unit ⇒ replicaId ⇒ versionVector ⇒ versionVector" where
"increment _ r = incVV r"

definition getValue :: "unit ⇒ versionVector ⇒ nat" where
"getValue _ v = (∑r∈ UNIV. v»r)"

definition incrementOnlyCounter :: "(versionVector, unit, unit, nat) stateBasedType"
  where
"incrementOnlyCounter = (|
      t_compare = op≤ ,
      t_merge = sup,
      t_initial = vvZero,
      t_update = increment,
      t_query = getValue
  |) "
```

Figure 32: Increment-Only Counter Implementation

The main proof obligation for this is:

```
(∑r∈ UNIV. length (H r)) = card (allUpdates H)
```

The left part is the implementation of the getValue function with the invariant plugged in. The right part is the specification. Unfolding the definition of allUpdates yields:

```
(∑r∈ UNIV. length (H r)) = card (⋃r. set (H r))
```

Isabelle cannot prove this automatically, because the length of a list and the cardinality of a set made from that list is only the same if the list contains no duplicate elements. Because every event is unique, there are no duplicate elements and the equality can be shown with some manual work.

7.2. PN-Counter

The PN-Counter is a counter which also allows decrementing its value. The implementation in figure 33 is based on specification 7 in [12], but instead of having separate increment and decrement functions there is only one update function which takes an integer value.

The payload is basically a pair of version vectors, but to avoid conversions between natural numbers and integers, the version vector type was not used. Instead the payload uses a function from replicaIds to integers.

The first component of the payload counts the positive updates and the second component counts the negative updates. Splitting the updates like this is essential for having monotonic updates and thus convergence.

Proving that this CRDT forms a semilattice is relatively simple. No invariant is required for the payload. Replacing the natural numbers with integers in the payload had no influence on the semilattice properties. But choosing a normal function type instead of the version vector type led to some additional steps in the proof.

Figure 34 shows the specification of the PN-counter and the invariant to prove it correct. The value of the counter is just the sum of the update arguments of all events. The invariant captures the splitting of events in a group of positive and a group of negative updates.

The proof is similar to the increment-only case. The main problem is in showing that the invariant implies the specification. This requires rewriting of some sums over sets to sums over lists, which requires some manual steps in Isabelle.

```
type_synonym payload = "(replicaId ⇒ int) × (replicaId ⇒ int)"

fun update :: "int ⇒ replicaId ⇒ payload ⇒ payload" where
 "update x r (p,n) = (if x ≥ 0 then (p(r:=p r + x), n) else (p, n(r:=n r − x)))"

fun getValue :: "unit ⇒ payload ⇒ int" where
"getValue _ (p,n) = ((∑r∈ UNIV. p r) − (∑r∈ UNIV. n r))"

definition pnCounter where
"pnCounter = ⦇
      t_compare = (λx y. ∀r. fst x r ≤ fst y r ∧ snd x r ≤ snd y r),
      t_merge = (λx y. (λr. max (fst x r) (fst y r), λr. max (snd x r) (snd y r))),
      t_initial = (λr. 0, λr. 0),
      t_update = update,
      t_query = getValue
  ⦈"
```

Figure 33: PN-Counter Implementation

```
definition pnCounterSpec :: "(int,unit,int) crdtSpecification" where
"pnCounterSpec H q = (∑e∈ allUpdates H. updArgs(e))"

definition pnCounterInvariant where
"pnCounterInvariant H pl = (∀r.
  (fst pl) r =
    listsum(map updArgs (filter (λx. updArgs x ≥ 0) (H r)))
∧ (snd pl) r =
    listsum(map (λx. −updArgs(x)) (filter (λx. updArgs x < 0) (H r))))"
```

Figure 34: PN-Counter specification and invariant

```
definition add :: "'a ⇒ replicaId ⇒ 'a set ⇒ 'a set" where
"add x r = insert x"

definition lookup :: "'a ⇒ 'a set ⇒ bool" where
"contains x pl = (x∈ pl)"

definition gSet :: "('a set, 'a, 'a, bool) stateBasedType" where
"gSet = (|
      t_compare = op⊆,
      t_merge = op∪,
      t_initial = {},
      t_update = add,
      t_query = lookup
  |)"
```

Figure 35: Grow-Set implementation

```
definition gSetSpec :: "('a,'a,bool) crdtSpecification" where
"gSetSpec H x = (∃e∈ allUpdates H. updArgs(e) = x)"

definition gSetInvariant where
"gSetInvariant H pl = (∀x. x∈ pl ↔ (∃e∈ allUpdates H. updArgs(e) = x))"
```

Figure 36: Grow-Set specification and invariant

7.3. Grow-Set

The Grow-Set (or G-Set) is a set data structure which can only grow with time. There is an add operation, but no remove operation. Figure 35 shows an implementation of the Grow-Set which is based on specification 11 from [12].

The semilattice properties of the G-Set are very easy to prove, since the payload is just a set and sets form a semilattice with the used operations for merging and comparing.

The specification of the G-Set and the invariant used to prove it are given in figure 36. The proof can be done using automatic methods.

```
type_synonym 'a payload = "'a set × 'a set"

datatype 'a updateArgs = Add 'a | Remove 'a

fun update :: "'a updateArgs ⇒ replicaId ⇒ 'a payload ⇒ 'a payload" where
  "update (Add x) r (E,T) = (insert x E, T)"
| "update (Remove x) r (E,T) = (E, insert x T)"

fun lookup :: "'a ⇒ 'a payload ⇒ bool" where
"lookup x (E,T) = (x∈ E ∧ x∉T)"

definition twoPhaseSet :: "('a payload, 'a updateArgs, 'a, bool) stateBasedType" where
"twoPhaseSet = (|
      t_compare = (λx y. fst x ⊆ fst y ∧ snd x ⊆ snd y),
      t_merge = (λx y. (fst x ∪ fst y, snd x ∪ snd y)),
      t_initial = ({},{}),
      t_update = update,
      t_query = lookup
  |)"
```

Figure 37: Two-Phase-Set Implementation

7.4. Two-Phase-Set

The Two-Phase set is a set where elements can be added and removed, but where an element cannot be added again after it was removed. An implementation of a Two-Phase-Set is shown in figure 37. This implementation is based on specification 12 from [12]. The payload consists of two sets, where the first set E contains all elements added to the Two-Phase-Set and the second set T is a set of tombstones for the removed elements. This CRDT is basically a combination of two G-Sets and so the semilattice properties are similarly easy to show.

The specification shown in figure 38 states that an element x is in the set when there exists an operation Add x and no operation Remove x. This specification can be shown valid by using the invariant in figure 38 mainly by automatic methods.

Compared to the original description in [12], the implementation in figure 37 simplifies the case for removing an element. In the original the remove method has a precondition stating that the element to be removed must be in the set. One way this precondition can be added to the set is to only add an element to the tombstones when the element was in the set before. The update function for the remove case then becomes (E, **if** x∈ E **then** insert x T **else** T). Also the specification changes, as remove operations only have an effect when there was an add operation for the same element which happened

```
definition twoPhaseSetSpec :: "('a updateArgs,'a,bool) crdtSpecification" where
"twoPhaseSetSpec H x = (∃e∈ allUpdates H. updArgs(e) = Add x ∧ ¬(∃e∈ allUpdates H.
  updArgs(e) = Remove x))"

definition Inv where
"Inv H pl = (∀x. (x∈ fst pl ↔ (∃e∈ allUpdates H. updArgs(e) = Add x))
  ∧ (x∈ snd pl ↔ (∃e∈ allUpdates H. updArgs(e) = Remove x)))"
```

Figure 38: Two-Phase-Set specification and invariant

```
definition twoPhaseSetSpec :: "('a updateArgs,'a,bool) crdtSpecification" where
"twoPhaseSetSpec H x = ((∃e∈ allUpdates H. updArgs(e) = Add x)
  ∧ ¬(∃e∈ allUpdates H. updArgs(e) = Remove x
        ∧ (∃f∈ allUpdates H. updArgs(f) = Add(x) ∧ f ≺ e)))"
```

Figure 39: Two-Phase-Set specification with guarded remove operations

before the remove. Figure 39 shows the changed specification. Proving this specification requires some more work, since it involves the happens-before relation ($\prec$).

For example, when showing that a merge preserves the invariant, a case comes up, where there is a remove operation in the first update history and an add operation in the second update history, which happens after the remove operation. Here it has to be shown, that the remove operation must also be in the second update history. Intuitively, this holds true, because a valid update history cannot skip any operations, which happened before other operations in the update history. The lemma shown in figure 40 captures this intuition. There are some variants of this lemma to make it easier to apply in different situations.

When showing, that an update operation preserves the invariant, the need for an other helper lemma arises. In the case of an add operation it has to be shown that the add operation cannot happen before any of the remove operations in the update history. This can be derived from the fact that the version of the new operation is incVV r v and v is the supremum of all the updates in the old update history.

When the helper lemmas for the above cases are available, Isabelle is able to find the proofs using the sledgehammer tool. As the helper lemmas are not specific to the Two-Phase-Set, it can be said, that this variation of the Two-Phase-Set can also be verified using automated methods.

```
lemma consistentHistoriesHappensBefore3: "⟦
consistentHistories H1 H2;
validUpdateHistory H1;
validUpdateHistory H2;
e ≺ f;
e ∈ allUpdates H1;
f ∈ allUpdates H2
⟧ ⟹ e ∈ allUpdates H2"
```

Figure 40: Helper Lemma

```
datatype 'a updateArgs = Add 'a | Remove 'a
datatype 'a queryArgs = Contains 'a | GetElements
datatype 'a result = ContainsResult bool | GetElementsResult "'a set"

definition setSpecContains where
"setSpecContains H x = (∃a∈ allUpdates H. updArgs a = Add x
                ∧ ¬(∃r∈ allUpdates H. a ≺ r ∧ updArgs r = Remove x))"

fun setSpec :: "('a updateArgs) updateHistory ⇒ 'a queryArgs ⇒ 'a result" where
  "setSpec H (Contains x) = ContainsResult(setSpecContains H x)"
| "setSpec H GetElements = GetElementsResult {x. setSpecContains H x}"
```

Figure 41: Observed-Remove-Set specification

7.5. Observed-Remove-Set

Figure 41 shows the specification of the OR-Set. Compared to the previous set implementations, this set has two different query operations. One query Contains, which checks whether an element is in the set and an other query GetElements, which returns all elements in the set. The essential part of the specification is the definition setSpecContains which states that an element is in the set if there exists an operation Add(x), so that there is no operation Remove(x) which happened after the add-operation.

7.5.1. Simplified Implementation

This first implementation, which is shown in figure 42, is a simplified variant of the implementation shown in figure 2 from [11]. The payload has a set of elements and a set of tombstones, similar to the Two-Phase-Set. The difference is that elements are stored together with an unique identifier, so that different add-operations can be distinguished.

In the case of a remove-operation tombstones are only added for the identifiers, which are already present. So add-operations which happen later or in parallel to the remove-operation are not affected.

The biggest difference to the original implementation is, that the original uses a built-in function unique, which returns a new unique identifier. Such a function is not supported by the system model used in this thesis, but it can be emulated, for example by using version vectors. Version vectors are unique for each update, so they have the same uniqueness property as a built-in unique function would have, when it is called only once per update. As getting the current version vector is not possible in the system model, the version vector has to be part of the payload and has to be maintained by the implementation. This means that every update operation has to increase the version vector in the local component and the merge operation has to take the supremum of the two merged version vectors.

The semilattice properties for this set can be proven automatically, without providing an invariant, because the payload is just an independent combination of standard semilattices, namely sets and version vectors.

Showing that this implementation satisfies the OR-Set specification requires some more effort. The invariant used for the proof is shown in figure 43. The first part of the invariant describes that an element x with unique version v is in the elements set, if and only if there exists an operation Add x with version v. The second part describes, when an element is in the tombstone set. This is the case when there exists an operation Add x as before and an operation Remove x, so that the remove-operation happened after the add-operation. The last part of the invariant states that the version vector in the payload corresponds with the version of the update history[1].

Using a version vector as the unique identifier had the advantage, that there is a very easy mapping between add-operations in the update history and the entries of the payload. If a different unique identifier had been used, the invariant would have to be extended by a description of this mapping. This would have made the proofs more complicated.

During the verification of this implementation a small error was discovered. In the original description the remove-operation computes the set R of entries to be removed with the formula $R = \{(e, n) | \exists n : (e, n) \in E\}$. But when this expression is translated to Isabelle code in a straight forward way, one gets an equation like
$R = \{(e, n).\ (e, n) \in E \wedge (\exists n.(e, n) \in E)\}$. 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, as done in the implementation shown in figure 42. This problem was not a real problem 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] The version of the update history is defined by just taking the number of updates on each replica: updateHistoryVersion H = VersionVector (λr. length (H r))

```
type_synonym 'a payload = "('a × versionVector) set × ('a × versionVector) set ×
  versionVector"
abbreviation elements :: "'a payload ⇒ ('a × versionVector) set" where
"elements x ≡ fst x"
abbreviation tombstones :: "'a payload ⇒ ('a × versionVector) set" where
"tombstones x ≡ fst (snd x)"
abbreviation versionVec :: "'a payload ⇒ versionVector" where
"versionVec x ≡ snd (snd x)"

definition add :: "replicaId ⇒ 'a payload ⇒ 'a ⇒ 'a payload" where
"add myId pl e = (let vv = incVV myId (versionVec pl) in
      (elements pl ∪ {(e,vv)}, tombstones pl, vv))"

definition remove :: "replicaId ⇒ 'a payload ⇒ 'a ⇒ 'a payload" where
"remove myId pl e = (let R = {(e',n). (e',n)∈ elements pl ∧ e'=e} in
    (elements pl, tombstones pl ∪ R, incVV myId (versionVec pl)))"

definition contains :: "'a payload ⇒ 'a ⇒ bool" where
"contains pl e = (∃n. (e,n)∈ elements pl ∧ (e,n)∉tombstones pl)"

definition getElements :: "'a payload ⇒ 'a set" where
"getElements pl = {e. contains pl e}"

fun update :: "'a updateArgs ⇒ replicaId ⇒ 'a payload ⇒ 'a payload" where
  "update (Add e) r pl = add r pl e"
| "update (Remove e) r pl = remove r pl e"

fun getValue :: "'a queryArgs ⇒ 'a payload ⇒ 'a result" where
  "getValue (Contains e) pl = ContainsResult (contains pl e)"
| "getValue (GetElements) pl = GetElementsResult (getElements pl)"

definition ORsetSimple where
"ORsetSimple = (|
      t_compare = (λA B. elements A ⊆ elements B
                       ∧ tombstones A ⊆ tombstones B
                       ∧ versionVec A ≤ versionVec B),
      t_merge = (λA B. (elements A ∪ elements B,
                        tombstones A ∪ tombstones B,
                        sup (versionVec A) (versionVec B))),
      t_initial = ({}, {}, vvZero),
      t_update = update,
      t_query = getValue
  |)"
```


Figure 42: Observed-Remove-Set, simplified implementation

```
definition ORsetInv :: "('a updateArgs) updateHistory ⇒ 'a payload ⇒ bool" where
"ORsetInv H pl = (∀v x.
    ((x,v)∈ elements pl ↔
        (∃a∈ allUpdates H. updVersion(a)=v ∧ updArgs(a) = Add x))
  ∧ ((x,v)∈ tombstones pl ↔
        (∃a∈ allUpdates H. ∃r∈ allUpdates H. a ≺ r
            ∧ updVersion(a) = v
            ∧ updArgs(a) = Add x
            ∧ updArgs(r) = Remove x))
  ∧ versionVec pl = updateHistoryVersion H)"
```

Figure 43: Observed-Remove-Set, invariant for simplified implementation

7.5.2. Original Implementation

The original implementation from [11] has one small optimization, which was not included in the simplified implementation: It removes entries from the elements set when they are added to the tombstone set. The implementation shown in figure 44 is equivalent to the original implementation shown in figure 2 in [11], except for the generation of unique identifiers, which is the same as in the simplified implementation. Because of the small optimization, the implementation differs in the query-, remove-, merge-, and compare-functions, when compared to the simplified implementation.

This small optimization makes a huge difference when verifying the semilattice properties of the CRDT. Now the set of elements and the set of tombstones are no longer independent and the semilattice properties of the two individual sets can not simply be transferred to the pair of sets. For example when an element with the same version vector is both in the elements set and the tombstones set, then merging the payload with itself would remove the entry from the element set. This would contradict the requirement, that merge is idempotent. Therefore an invariant has to at least include the requirement that the elements and tombstones are disjoint. However this invariant alone is not sufficient for showing that the invariant is maintained. With just this invariant it would be possible, that an add-operation adds an entry which is already in the tombstones set. So the invariant has to include enough information to imply the uniqueness of version vectors. This can be achieved by stating that all version vectors appearing in the elements- and tombstones-sets are smaller or equal to the current version vector. As newly created version vectors are always greater than all previous version vectors, this implies the uniqueness. Figure 45 shows the final invariant, which is sufficient to show the convergence of the CRDT.

For showing that the implementation satisfies the OR-Set specification, the technique of showing equivalence to the simplified implementation was used. This technique is well suited in this case, because the two implementations behave very similar. Entries in the

sets have the same version vectors, so no complicated coupling is required. The coupling invariant is shown in figure 46. Besides the relations between the sets, the coupling also includes the same information about version vectors, as the invariant used for proofing the convergence of the CRDT. Using this coupling invariant the equivalence of the two implementations can be shown in only a few lines of proof code.

```
definition add :: "replicaId ⇒ 'a payload ⇒ 'a ⇒ 'a payload" where
"add myId pl e = (let vv = incVV myId (versionVec pl) in
      (elements pl ∪ {(e,vv)}, tombstones pl, vv))"

definition remove :: "replicaId ⇒ 'a payload ⇒ 'a ⇒ 'a payload" where
"remove myId pl e = (let R = {(e',n). (e',n)∈ elements pl ∧ e'=e} in
    (elements pl − R, tombstones pl ∪ R, incVV myId (versionVec pl)))"

definition merge :: "'a payload ⇒ 'a payload ⇒ 'a payload" where
"merge A B = ((elements A − tombstones B) ∪ (elements B − tombstones A)
                  , tombstones A ∪ tombstones B
                  , sup (versionVec A) (versionVec B))"

definition contains :: "'a payload ⇒ 'a ⇒ bool" where
"contains pl e = (∃n. (e,n)∈ elements pl)"

definition getElements :: "'a payload ⇒ 'a set" where
"getElements pl = {e. contains pl e}"

fun update :: "'a updateArgs ⇒ replicaId ⇒ 'a payload ⇒ 'a payload" where
  "update (Add e) r pl = add r pl e"
| "update (Remove e) r pl = remove r pl e"

fun getValue :: "'a queryArgs ⇒ 'a payload ⇒ 'a result" where
  "getValue (Contains e) pl = ContainsResult (contains pl e)"
| "getValue (GetElements) pl = GetElementsResult (getElements pl)"

definition compare :: "'a payload ⇒ 'a payload ⇒ bool" where
"compare A B = (versionVec A ≤ versionVec B
            ∧ (elements A ∪ tombstones A) ⊆ (elements B ∪ tombstones B)
            ∧ (tombstones A ⊆ tombstones B))"

            definition ORset where
"ORset = (|
      t_compare = compare,
      t_merge = merge,
      t_initial = ({}, {}, vvZero),
      t_update = update,
      t_query = getValue
  |)"
```

Figure 44: Observed-Remove-Set, original implementation

```
definition elementsAndTombstonesDisjoint where
"elementsAndTombstonesDisjoint x = (elements x ∩tombstones x = {})"

definition elemetsSmallerVersionVec where
"elemetsSmallerVersionVec x = ((∀(a,av)∈ elements x. av ≤ versionVec x) ∧
  (∀(a,av)∈ tombstones x. av ≤ versionVec x))"

definition invariant :: "'a payload ⇒ bool" where
"invariant x = (elementsAndTombstonesDisjoint x ∧ elemetsSmallerVersionVec x)"
```

Figure 45: Observed-Remove-Set, invariant for original implementation

```
fun couplingInv :: "'a stateBasedORsetSimple.payload ⇒ 'a payload ⇒ bool" where
"couplingInv (Ea, Ta, Va) (Eb, Tb, Vb) = (
  Ea = Eb − Tb ∧ Ta = Tb ∧ Va = Vb
  ∧ (∀(x,v)∈ Eb. v≤ Vb)
  ∧ (∀(x,v)∈ Tb. v≤ Vb)
)"
```

Figure 46: Observed-Remove-Set, coupling between simplified and original implementation

7.5.3. Optimized Implementation

The original implementation of the OR-set still stores a lot of redundant information. Every add-operation is represented with an entry in the elements- or tombstones-set. The optimized implementation from [1] shows that this is not necessary. The translation of this implementation to Isabelle is shown in figure 47 and figure 48. The Isabelle translation only covers the state based part of the original optimized OR-Set.

The payload of the CRDT consists of two parts. The first part is a version vector, which counts the number of add-operations on each replica. The second part is a set of triples (e, c, r) representing add-operations, where e is the added element, r is the replica on which the element was added, and c is a kind of timestamp. More precisely c is the number of add-operations on the same replica, which happened up to and including the add-operation the triple represents. So for the first add-operation c will be one, for the second add-operation it will be two, and so on.

The implementation does not need a set of tombstones. It uses the version vector to decide if an element has been removed from the set. When merging two payloads and both contain the same triple, then it is certain, that the element has not been removed (set M in the merge function). When neither payload contains a triple for an element it is certain, that the element is not in the set. When one payload contains a triple, which the other does not contain, then the version vector is used to check if the add-operation represented by that triple has been observed by the other replica. If it has been observed, then the element is removed, otherwise it stays in the set (sets M' and M'' in the merge function).

Another optimization besides the missing tombstones set is the removal of old triples from the payloads, when there is a triple representing a newer add-operation.

Convergence. The aforementioned optimizations make it difficult to show the semilattice properties for this CRDT. The compare function in the original implementation was not suited well for showing the semilattice properties and so a different function was used (see figure 48). The problem with the original compare function was, that it assumed some uniqueness properties, for example that there cannot be two triples (x, c, r) and (y, c, r) with same timestamp and replica but different elements. This is an invariant which cannot be described locally. It could be described by describing the relation of the triples to the update history, but that would make the proofs unnecessary complicated. The new function does not need such an invariant and it corresponds better to the merge function, which makes the proofs easier.

The invariant used for showing the semilattice properties is shown in figure 49. The definitions invariant_c_unique and invariant_e_unique describe the fact, that on each replica there is at most one timestamp per element and at most one element per timestamp. The definitions invariant_c_lower_bound and invariant_c_upper_bound state the timestamps c in the triples are greater than zero and less than the respective component in the version vector. Showing that this invariant is preserved by updates is very easy, showing that merges preserve the invariant is a bit more difficult, but can also be done with mainly automatic methods. With this invariant it is possible to show the semilat-

```
type_synonym 'a payload = "(versionVector × ('a × nat × replicaId) set)"

abbreviation versionVec :: "'a payload ⇒ versionVector" where
"versionVec x ≡ fst x"

abbreviation elements :: "'a payload ⇒ ('a × nat × replicaId) set" where
"elements x ≡ (snd x)"

fun contains :: "'a payload ⇒ 'a ⇒ bool" where
"contains pl e = (∃c i. (e,c,i)∈ elements pl)"

definition getElements :: "'a payload ⇒ 'a set" where
"getElements pl = {e. ∃c i. (e,c,i)∈ elements pl}"

definition add :: "replicaId ⇒ 'a payload ⇒ 'a ⇒ 'a payload" where
"add r pl e = (
    let c = (versionVec pl)»r + 1;
        Old = {(e',c',r'). e=e'∧ r=r' ∧ (e,c',r)∈ elements pl ∧ c' < c}
    in (incVV r (versionVec pl), elements pl ∪ {(e,c,r)} − Old))"

definition remove :: "replicaId ⇒ 'a payload ⇒ 'a ⇒ 'a payload" where
"remove myId pl e = (let R = {(e',c,i). e=e' ∧ (e',c,i)∈ elements pl} in
    (versionVec pl, elements pl − R))"

fun update :: "'a updateArgs ⇒ replicaId ⇒ 'a payload ⇒ 'a payload" where
  "update (Add e) r pl = add r pl e"
| "update (Remove e) r pl = remove r pl e"

fun getValue :: "'a queryArgs ⇒ 'a payload ⇒ 'a result" where
  "getValue (Contains e) pl = ContainsResult (contains pl e)"
| "getValue (GetElements) pl = GetElementsResult (getElements pl)"
```

Figure 47: Observed-Remove-Set, optimized implementation based on figure 3 from [1] (Part 1/2)

```
definition compare2:: "'a payload ⇒ 'a payload ⇒ bool" where
"compare2 A B = (
    versionVec A ≤ versionVec B
  ∧ (∀c i. ((c > (versionVec B)»i ∨ (∃e. (e, c, i) ∈ elements B))
          ⟶ (c > (versionVec A) »i ∨ (∃e. (e, c, i) ∈ elements A))))
  ∧ (∀e1 e2 c i. (e1,c,i)∈ elements A ∧ (e2,c,i)∈ elements B ⟶ e1=e2))"

definition merge:: "'a payload ⇒ 'a payload ⇒ 'a payload" where
"merge A B = (
  let M = elements A ∩elements B;
      M' = {(e,c,i). (e,c,i)∈ elements A − elements B ∧ c > (versionVec B)»i};
      M'' = {(e,c,i). (e,c,i)∈ elements B − elements A ∧ c > (versionVec A)»i};
      U = M ∪ M' ∪ M'';
      Old = {(e,c,i). (e,c,i)∈ U ∧ (∃c'. (e,c',i)∈ U ∧ c < c')}
  in (sup (versionVec A) (versionVec B), U − Old))"

definition ORsetOpt where
"ORsetOpt = (|
      t_compare = compare2,
      t_merge = merge,
      t_initial = (vvZero, {}),
      t_update = update,
      t_query = getValue
  |)"
```

Figure 48: Observed-Remove-Set, optimized implementation based on figure 3 from [1] (Part 2/2)

```
definition invariant_c_unique :: "'a payload ⇒ bool" where
"invariant_c_unique A = (∀e i c1 c2.
      (e,c1,i)∈ elements A ∧ (e,c2,i)∈ elements A ⟶ c1=c2)"

definition invariant_e_unique :: "'a payload ⇒ bool" where
"invariant_e_unique A = (∀e1 e2 i c.
      (e1,c,i)∈ elements A ∧ (e2,c,i)∈ elements A ⟶ e1=e2)"

definition invariant_c_upper_bound :: "'a payload ⇒ bool" where
"invariant_c_upper_bound A = (∀e c i.
      (e,c,i)∈ elements A ⟶ c≤ (versionVec A)»i)"

definition invariant_c_lower_bound :: "'a payload ⇒ bool" where
"invariant_c_lower_bound A = (∀e c i.
      (e,c,i)∈ elements A ⟶ 0<c)"

definition invariant :: "'a payload ⇒ bool" where
"invariant A = (invariant_c_unique A ∧ invariant_e_unique A
         ∧ invariant_c_upper_bound A ∧ invariant_c_lower_bound A)"
```

Figure 49: Observed-Remove-Set, invariant for showing semilattice properties

tice properties. There the most effort is in showing that merge computes a least upper bound. Other properties like the transitivity of the compare function are easier to show.

Behavior. For showing that the optimized implementation implements the OR-set specification it is not easily possible to define a coupling between the optimized and the simple implementation, because there is no easy relation between the version vectors used in the simple implementation and the timestamps of the triples in the optimized implementation. The simple implementation counts add- and remove-operations in the version vector, while the optimized implementation counts only add-operations. It would probably be easier to first implement a version of the OR-set which is structured similar to the simple version, except that it does not count the remove-operations in the version vector. A similar approach has been taken in [11] to show the equivalence of the simple and optimized implementation. The approach taken here does not use this technique, but instead shows that the specification is valid without comparing it to other implementations.

The invariant used for the proof is shown in figure 50. The abbreviation addOperations filters a list of operations and only retains the add-operations. The predicate existsAddOperationWithoutRemove is true if there exists an add-operation on a certain replica, for a certain element, and there is no remove operation for that element which comes after

the add-operation. The invariant consists of three parts A, B, and C. Part A describes the contents of the version vector. Each component of the version vector is equal to the number of add-operations on the respective replica. Part B describes when a triple is in the set of elements. When there is an add operation for x on replica r without a later remove operation for x, then there exists a c so that the triple (x, c, r) is in the set. Otherwise no such triple exists in the set. Part C describes what it means for a triple (x, c, r) to be in the set of elements. This part contains several properties. First, it gives the same bounds to the timestamp c, as were used in the verification of the semilattice properties. Then, it states that the c-th element in the list of add-operations adds the element x to the set, and as last point part C states, that after that add-operation, there comes no other add-operation, which adds the same element x on the same replica r.

Part B of the invariant is enough to show that the invariant implies the specification and it can easily be seen, that this part implies the specification. The other parts are important for proving that the invariant is maintained. In this part of the proof the main difficulties are in the case for the merge operation and part B of the invariant, as there are many cases to be considered. There are a few more complex cases, which mostly are a result of the rather complicated relation between timestamps c and the position of the add-operations in the filtered list.

```
abbreviation "addOperations xs ≡ [e←xs. ∃x. updArgs(e) = Add x]"

definition "existsAddOperationWithoutRemove x r H = (∃e.
  e∈ set (H r)
  ∧ updArgs(e) = Add x
  ∧ ¬(∃f∈ allUpdates H. updArgs(f) = Remove x ∧ e ≺ f))"

definition
"invA H pl = (∀r. versionVec pl»r = length (addOperations (H r)))"

definition
"invB H pl = (∀x r. (if existsAddOperationWithoutRemove x r H
  then (∃c. (x,c,r) ∈ elements pl)
  else (∀c. (x,c,r) ∉elements pl)))"

definition
"invC H pl = (∀x c r. (x,c,r)∈ elements pl ⟶
          0<c
        ∧ c≤ length(addOperations (H r))
        ∧ updArgs(addOperations (H r) ! (c − 1)) = Add x
        ∧ (∀i<length(addOperations(H r)). i ≥ c
                ⟶ updArgs(addOperations(H r)!i) ≠Add x))"

definition Inv :: "('a updateArgs) updateHistory ⇒ 'a payload ⇒ bool" where
"Inv H pl = (invA H pl ∧ invB H pl ∧ invC H pl)"
```

Figure 50: Observed-Remove-Set, invariant for verifying the behavior

```
datatype 'a updateArgs = Assign "'a list"
type_synonym 'a returnType = "('a × versionVector) set"

definition mvRegisterSpec :: "('a updateArgs, unit, 'a returnType) crdtSpecification"
  where
"mvRegisterSpec H _ = {(x,v).
        (∀vv∈ allVersions H. ¬v<vv)
    ∧ (∃l. x∈ set l ∧ (v,Assign l)∈ allUpdates H)}"
```

Figure 51: Multi-Value-Register specification

7.6. Multi-Value-Register

A register is an object for storing and retrieving values. It provides an operation Assign for assigning values. The Multi-Value-Register (MV-Register) allows an arbitrary, finite number of values in the assign-operations. When there are two parallel assign-operations, a query will return the values from both assignments. A query does not just return the bare values. It also returns a version vector for each value, which is the version at which the value has been assigned. This version information can be used by clients to resolve the conflicts in the application.

Figure 51 shows a formal specification for a MV-Register. It describes more precisely which values are returned by a query: All the values which have a version that is not dominated by a higher version.

7.6.1. Simple Implementation

A simple implementation of the MV-Register just maintains a version vector and a set of pairs containing the assigned elements with the respective version vector. Such an implementation is shown in figure 52.

The merge operation just takes the supremum of the version vectors and the union of the sets. Therefore the semilattice properties of this CRDT are trivial to show. That the CRDT satisfies the specification is also easy to show by using the invariant from figure 53.

7.6.2. Optimized Implementation

The implementation shown in figure 10 from [12] requires less storage space than the simple implementation. It does not keep elements in the set, which are no longer relevant, because there exists an other element with a greater version. It also does not explicitly maintain a version vector in the payload. Instead the current version is implicitly defined by the supremum of all versions in the set. Figure 54 shows a translation of this implementation to Isabelle.

```
fun update :: "'a updateArgs ⇒ replicaId ⇒ 'a payload ⇒ 'a payload" where
"update (Assign l) r (V,S) = (incVV r V, S ∪ {(l, incVV r V)})"

fun getValue :: "unit ⇒ 'a payload ⇒ ('a × versionVector )set" where
"getValue _ (V,S) = {(x,v). ∃l. x∈ set l ∧ (l,v)∈ S ∧ (∀vv∈ snd`S. ¬v<vv)}"

definition mvRegister where
"mvRegister = ⦇
      t_compare = (λx y. fst x ≤ fst y ∧ snd x ⊆ snd y),
      t_merge = (λx y. (sup (fst x) (fst y), snd x ∪ snd y)),
      t_initial = (vvZero, {}),
      t_update = update,
      t_query = getValue
  ⦈"
```

Figure 52: Multi-Value-Register, simple implementation

```
definition mvRegisterInvariant :: "('a updateArgs) updateHistory ⇒ 'a payload ⇒ bool"
  where
"mvRegisterInvariant H pl = (
    (∀v l. (l,v)∈ snd pl ↔ (∃a∈ allUpdates H. updVersion(a)=v ∧ updArgs(a) = Assign l))
  ∧ fst pl = updateHistoryVersion H)"
```

Figure 53: Multi-Value-Register, invariant for simple implementation

```
type_synonym 'a payload = "('a option × versionVector) set"

definition incVersions :: "'a payload ⇒ replicaId ⇒ versionVector" where
"incVersions pl myId = (
    let V = snd ‘ pl;
        vv = supSet V
    in (incVV myId vv)
)"

definition assign :: "'a payload ⇒ replicaId ⇒ 'a list ⇒ 'a payload" where
"assign pl myId R = (let V = incVersions pl myId in (Some ‘ set R) × {V})"

fun update :: "'a updateArgs ⇒ replicaId ⇒ 'a payload ⇒ 'a payload" where
"update (Assign R) r pl = (assign pl r R)"

fun getValue :: "unit ⇒ 'a payload ⇒ 'a returnType" where
"getValue _ pl = {(x,v). (Some x, v) ∈ pl}"

definition compare :: "'a payload ⇒ 'a payload ⇒ bool" where
"compare A B = (∀x∈ A. ∀y∈ B. snd x ≤ snd y)"

definition merge :: "'a payload ⇒ 'a payload ⇒ 'a payload" where
"merge A B = (
    let A' = {(x,V). (x,V)∈ A ∧ (∀(y, W)∈ B. V ‖W ∨ V ≥ W)};
        B' = {(y, W). (y,W)∈ B ∧ (∀(x, V)∈ A. W ‖V ∨ W ≥ V)}
    in A' ∪ B'
)"

definition mvRegister where
"mvRegister = ⦇
      t_compare = compare,
      t_merge = merge,
      t_initial = {(None, vvZero)},
      t_update = update,
      t_query = getValue
  ⦈"
```

Figure 54: Multi-Value-Register, optimized and incorrect implementation. Translation of specification 10 from [12]

When trying to verify the semilattice properties of this CRDT two small problems were found. The first problem is the compare function, which does not define a partial order, as it is not reflective. For example the compare function returns false, when comparing the payload $\{(a, [1, 0]), (b, [0, 1])\}$ with itself.

The second 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. This 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: $\{(None, [0, 0])\} \xrightarrow{Assign(a)} \{(a, [1, 0])\} \xrightarrow{Assign(b)} \{(b, [2, 0])\} \xrightarrow{Assign()} \{\} \xrightarrow{Assign(c)} \{(c, [1, 0])\}$. 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 avoided by ignoring all assignments of an empty list of elements. The problem with the compare function requires a rewrite of the compare function. The new implementation of the assign and compare function are shown in figure 55.

For this fixed implementation it is possible to verify the semilattice properties using the invariant shown in figure 56. This figure also includes the definition of an alternative merge function, which makes the proofs easier. The function removeSmallerElements removes all elements from the given set, for which an element with a greater version vector exists.

The first part of the invariant states that the payload set is nonempty and finite. The finiteness requirement is necessary to ensure that there always is at least one element which is not dominated by an other element. In the case of an infinite set, it would be possible to have an infinite sequence of increasing version vectors with no such element. If no such element exists, a merge operation would result in an empty set and the incVersions function would not be well defined.

The second part of the invariant states that the version is zero if and only if there is no element. The third part states that there is no element in the set which is dominated by an other element, i.e. for which an other element has a greater version vector.

Unfortunately, the fix of the assign operation by making it ignore assignments of empty lists is not compatible with the specification of the MV-Register (see figure 51). It is possible to adjust the specification to also ignore the same operations, but then the simple mapping between the versions in the payload and the version of the replica no longer exists. And because query operations expose the version vectors, the specification also has to reflect this. Figure 57 shows the specification for the fixed implementation. Compared to the original specification, this specification is really complicated because of the necessary filtering. Because of limited time and the complexity of the specification, it was not verified, whether this specification is in fact a valid specification of the fixed optimized implementation.

```
definition assign :: "'a payload ⇒ replicaId ⇒ 'a list ⇒ 'a payload" where
"assign pl myId R = (if R=[] then pl
      else let V = incVersions pl myId in (Some ‘ set R) × {V})"

definition compareSingle :: "('a option × versionVector) ⇒ 'a payload ⇒ bool" where
"compareSingle x B = (case x of (a,v) ⇒ (∃(b,w)∈ B. v < w) ∨ (∃(b,w)∈ B. v=w ∧
  a=b))"

definition compare :: "'a payload ⇒ 'a payload ⇒ bool" where
"compare A B = (∀x∈ A. compareSingle x B)"
```

Figure 55: Multi-Value-Register, fixed assign and compare function for optimized implementation

```
definition noElementDominated where
"noElementDominated S = (∀(a,v)∈ S. ∀(b,w)∈ S. ¬v<w)"

definition vvZeroIffNone where
"vvZeroIffNone S = (∀(a,v)∈ S. v=vvZero ↔ a=None)"

definition invariant where
"invariant S = (
      S≠{}
    ∧ finite S
    ∧ vvZeroIffNone S
    ∧ noElementDominated S)"

definition mergeAlt where
"mergeAlt A B = removeSmallerElements (A ∪ B)"
```

Figure 56: Multi-Value-Register, invariant and alternative merge function for optimized implementation

```
definition mvRegisterSpec :: "('a updateArgs, unit, ('a × versionVector) set)
  crdtSpecification" where
"mvRegisterSpec H _ =
   ({(x,c). (∃r l v. x∈ set l ∧ (v, Assign l)∈ set(H r)
    ∧ (∀rr. c»rr = length (filter (λe. updArgs e ≠Assign [] ∧ updVersion e ≤ v ) (H rr)))
    ∧ ¬(∃l' vv. l' ≠[] ∧ vv>v ∧ (vv, Assign l')∈ allUpdates H))})"
```

Figure 57: Multi-Value-Register, specification of the fixed optimized implementation

8. Related work

There exists work by Christopher Meiklejohn, which uses the interactive theorem prover Coq for verifying semilattice properties of CRDTs[7, 8, 9]. The behavioral properties of CRDTs are not considered. Verified CRDTs are the increase-only-counter and the PN-counter. In contrast to this thesis the verification is not based on a formal system model. Instead the semilattice properties are verified standing alone, but this is not a problem, because it is known from the literature[12], that the semilattice properties with monotonic updates are a sufficient condition for convergence. However the verified semilattice properties are slightly different than the properties used in this thesis. Instead of showing that merge computes a least upper bound, it is only shown that it computes an upper bound and that the operation is commutative, idempotent and associative. The last three properties are equivalent to the computation of a least upper bound, when the compare function satisfies the equation $compare\ x\ y \leftrightarrow (merge\ x\ y = y)$, but this equation is not verified. It would be interesting to check if the verified conditions are sufficient for convergence, as well.

Another work including verification of CRDTs is *Replicated Data Types: Specification, Verification, Optimality*[3]. This paper follows a similar approach of basing the theory on a system model and it follows a very similar approach in specifying and verifying the behavior of CRDTs. In the paper no use of a tool for the verification is mentioned, so it can be assumed that the proofs are not machine checked.

The basic system model is more flexible than the system model used in this thesis. The model uses explicit send- and receive- actions, where the send actions also can have side-effects on the payload. This feature allows to also emulate operation-based CRDTs in the model, by caching the effect of operations locally and then send them in a batch of operations. The system also has a more flexible notation of visibility, than the visibility which was defined by version vectors in this thesis and the system supports timestamps, which allows having data types like the Last-Writer-Wins-Register. While the system is more flexible, only state-based CRDTs are verified, and except for timestamps the flexibility of the system is not used in the state-based implementations. For the verification the flexible parameters of the system are instantiated so that the result is a system, which is very close to the system used in this thesis. In the paper the behavior of the Increment-Only-Counter, a Last-Writer-Wins-Register and a optimized OR-Set are verified. The OR-Set has a slightly different structure than the OR-Set verified in this thesis, but is basically equivalent.

9. Conclusion and future work

The case studies have shown, that 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-language 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. The framework presented in this thesis is not directly applicable to real implementations. Several approaches are possible to close the gap between the verified Isabelle code and a real implementation:

1. Isabelle can generate code for the languages SML, OCaml, Haskell and Scala from the Isabelle code. An advantage of using this approach would be that there is a real end-to-end verification from the abstract system model to the executable code. So there would only be the system model, the specification of the specific CRDT, and the Isabelle tool, which would have to be trusted, as the rest is verified to be correct. A disadvantage is, that the generated code is not very readable and not very efficient. Also, the generated code does not use the default data types of the language, so probably some wrapper code would be necessary to conform with the necessary interfaces.

2. It would be possible to describe CRDTs in a domain specific language and generate Isabelle code and executable code from this description. This approach would be more flexible in the generation of the executable code, so that the code could be more readable, efficient and be compliant to the necessary interfaces. The advantages come at the cost of more possible points of failures. With this approach one additionally has to trust the code generator for the domain specific language.

3. The executable code could be written manually and then it could be verified that the executable code is equivalent to the more abstract Isabelle code. This can be done separately for each function. As there is no concurrency inside a function (each update and merge is executed on a single replica) this equivalence can be shown using techniques and tools for sequential programs. This approach is very flexible in what can be done in the implementation of the executable code, but it has more possible points of failure than the first two approaches.

There are some other challenges in real system which could be addressed in future work. One problem is the space complexity of payloads. Because of the requirement for monotonically increasing updates the required space for storing the payload is also increasing with time. Optimized implementations like the optimized OR-Set can reduce the space complexity, but there are limits for the possible optimizations [3]. Furthermore, in real systems replicas can be dynamically added to and removed from the system, and data which references removed nodes should also be garbage-collected. Therefore long-running systems will probably need a form of garbage-collection. With the system

model used in this thesis, correct garbage collection is not possible, because arbitrary old messages can arrive at any time. In a real system, one can probably guarantee a certain bound on how old messages can be, and this would allow to implement garbage-collection. It would be interesting to verify CRDTs with garbage-collection in such a setting.

An other possible future work is researching different specification techniques for CRDTs. In this master thesis, some techniques were discussed, but they were not evaluated against applications. It would be interesting to look at applications, which use CRDTs, and evaluate what kind of specifications are required for the CRDTs, so that it is possible to verify that the application works correctly.

It is also an open question, which kind of CRDTs are required by applications. 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 might be a good idea.

Some automatic tools are imaginable in this area. A simple automatic tool could just do automatic testing of the semilattice properties. This would not require any additional steps by the developer like writing a specification. When the developer can write down a suitable invariant for the payload, then automatic verification of the semilattice properties could be possible. The case studies in this thesis showed, that the semilattice properties for the simpler CRDTs could be shown with mainly automated methods, so an automated verification tool for simple CRDTs is realistic.

Automatic testing would also be possible for checking behavioral specifications. Automatic verification of the behavioral properties could also be possible, but the case studies showed that verifying the behavioral properties is more difficult, than verifying the semilattice properties.

Automatic testing tools could also benefit from a defined invariant. A testing tool could then try to generate random payloads satisfying the invariant and random update arguments to test a single operation in isolation. Without an invariant, an automated testing tool has to generate complete traces, which form a much greater search space. Finding payloads satisfying an invariant can maybe be done using an automatic theorem prover like Z3, similar to what is done in the automatic testing tool Pex[13].

Finally, to give a brief overview of the effort required for the different CRDTs verified in the case studies, figure 58 shows the lines of code used in each case. The lines are counted including comments and whitespace and should be interpreted only as a very rough estimation of effort. It would probably be possible to invest additional effort or give the task to a more experienced Isabelle user and get a much smaller theory file. Figure 59 shows the lines of code for the general theories. A mapping between the theory files and the sections in this thesis is given in the Appendix.

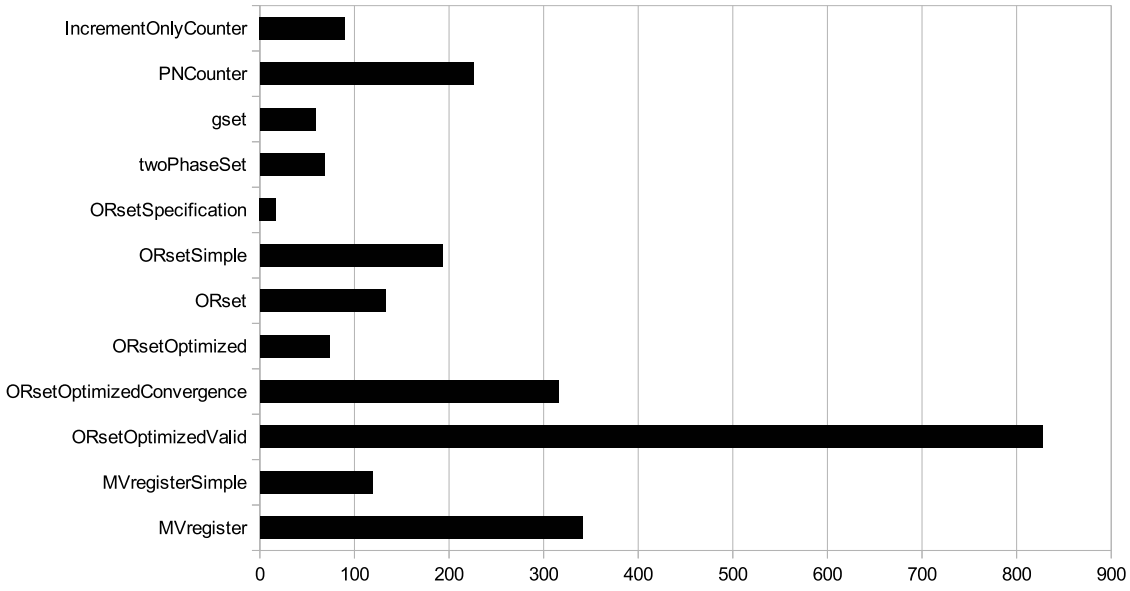

Figure 58: Lines of code in case study theories

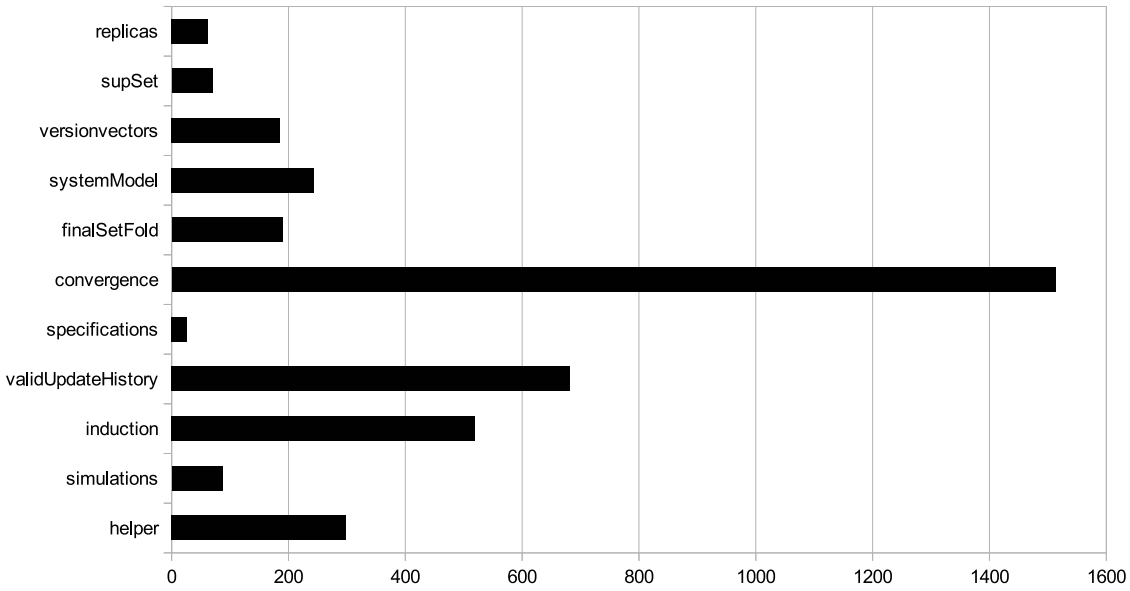

Figure 59: Lines of code in general theories

References

[1] 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.

[2] Sebastian Burckhardt, Alexey Gotsman, and Hongseok Yang. Understanding Eventual Consistency. 2013.

[3] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. 2013.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[5] Kyle Kingsbury. The trouble with timestamps. http://aphyr.com/posts/299-the-trouble-with-timestamps, October 2013.

[6] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[7] Christopher Meiklejohn. Distributed data structures with Coq. http://christophermeiklejohn.com/coq/2013/06/11/distributed-data-structures.html, June 2013.

[8] Christopher Meiklejohn. Distributed data structures with Coq: PN-Counters. http://christophermeiklejohn.com/coq/2013/06/20/distributed-data-structures-pn-counters.html, June 2013.

[9] Christopher Meiklejohn. Github repository - Distributed data structures. https://github.com/cmeiklejohn/distributed-data-structures, 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] Nuno M. Preguiça, Annette Bieniusa, Marc Shapiro, Marek Zawirski, Carlos Baquero, Valter Balegas, and Sérgio Duarte. Conflict-free replicated sets. 2013.

[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] Nikolai Tillmann and Jonathan Halleux. Pex – White Box Test Generation for .NET. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin Heidelberg, 2008.

Appendices

A. Isabelle theories

The Isabelle theories are available on the attached CD and on request (`p_zeller@cs.uni-kl.de`). The theory files were created with version *Isabelle2013: February 2013*. The following table gives an overview of the theory files with a short description explaining how they relate to the sections in this thesis.

Theory	**Description**
finalSetFold	no corresponding section, helper lemmas about the fold function for finite sets
supSet	no corresponding section, helper lemmas about taking supremum over a finite set.
replicas	System model (section 3.2)
versionvectors	System model (section 3.2)
systemModel	System model (section 3)
convergence	Consistency (section 4), proof that the semilattice properties are sufficient for convergence
specifications	Specifications, formalization of the last approach (section 5.3.2)
validUpdateHistory	Valid update history (section 6.1)
induction	Section about Verifying CRDT behavior, theory of the induction scheme (section 6)
simulations	Section about Verifying CRDT behavior, verification by showing equivalence (section 6.3)
helper	no corresponding section, contains various helper lemmas used by the case studies
IncrementOnlyCounter	Case study: Increment-Only counter
PNCounter	Case study: PN-Counter
gset	Case study: Grow-Set
twoPhaseSet	Case study: Two-Phase-Set
ORsetSpecification	Case study: OR-Set (specification)
ORsetSimple	Case study: OR-Set (simple implementation)
ORset	Case study: OR-Set (original implementation)
ORsetOptimized	Case study: OR-Set (optimized implementation)
ORsetOptimizedConvergence	Case study: OR-Set (optimized implementation, verification of semilattice properties)
ORsetOptimizedValid	Case study: OR-Set (optimized implementation, verification of behavioral specification)
MVregisterSimple	Case study: MV-Register (simple implementation)
MVregister	Case study: MV-Register (optimized implementation)