On Effective Verification of Replicated Data Types

A thesis submitted in partial fulfilment of the requirement of the degree of Doctor of Philosophy (Ph.D.) in Computer Science

by

Gautham Shenoy R
Chennai Mathematical Institute, Chennai

Supervisors

Prof. Madhavan Mukund Prof. S. P. Suresh

Declaration

I declare that the thesis "On Effective Verification of Replicated Data Types" submitted by me for the degree of Doctor of Philosophy is the record of work carried out by me from August 2012 to September 2021 under the guidance of Prof. Madhavan Mukund and Prof. S. P. Suresh from the Chennai Mathematical Institute. This work has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this or any other university or other similar institution of higher learning.

September 2021 Gautham Shenoy R.

Chennai Mathematical Institute
H1, SIPCOT IT Park
Siruseri, Kelambakkam
Tamil Nadu 603103, India

Certificate

This is to certify that the Ph.D. thesis titled "On Effective Verification of Replicated Data Types" submitted by Gautham Shenoy R. to Chennai Mathematical Institute is a record of bona fide research work done during the period August 2012 – September 2021 under our guidance and supervision. The research work presented in this thesis has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this or any other university or other similar institution of higher learning.

It is further certified that the thesis represents independent work by the candidate and collaboration when existed was necessitated by the nature and scope of the problems dealt with therein.

Madhavan Mukund
Chennai Mathematical Institute
H1, SIPCOT IT Park
Siruseri, Kelambakkam
Tamil Nadu 603103, India

S. P. Suresh
Chennai Mathematical Institute
H1, SIPCOT IT Park
Siruseri, Kelambakkam
Tamil Nadu 603103, India

Abstract

In this thesis, we investigate formal methods for providing cogent specifications and performing effective verification of replicated data types such as the CRDTs adhering to weaker notions of consistency. We analyse a particular CRDT known as the *Observed-Remove Set (OR-Set)* where the conflicts between concurrent *add* and *remove* operations are handled in a coordination free manner, even when updates are presented out of order. We provide an optimized implementation of this data type. Further, we provide a framework for defining declarative specifications for these data types. We present two methodologies to construct bounded reference implementations of these data types. We show how these reference implementations can be used for effective verification of given implementations of replicated data types. Finally we turn our attention to the verification of read-write key-value stores that support multiple consistency criteria. Some of the novel contributions in this thesis include

- *Interval Version Vectors*, which allow us to robustly keep track of concurrent updates in the absence of causal delivery. We use these to provide an optimized implementation of OR-set in the absence of causal-delivery.

- Formulation of a generalization of the *Gossip Problem* [Mukund and Sohoni, 1997; Mukund et al., 2003] and a *bounded solution* to this *Generalized Gossip Problem* that provides a principled approach towards synthesizing distributed reference implementations of CRDTs from their specifications.

- A technique for synthesizing a simple global reference implementation of CRDTs from their specification using the concept of *Later Appearance Records (LAR)* introduced in [Gurevich and Harrington, 1982].

- A formal framework for defining the correctness of behaviours of read-write data-stores which provide multiple consistency levels.

- A *systematic methodology* for deriving bad patterns characterizing a wide range of consistency models and combinations thereof.

- An *effective algorithmic framework* for testing the behaviours of modern data-stores that providing multiple levels of consistency.

Acknowledgements

First and foremost I would like to thank my advisors Prof. Madhavan Mukund and Prof. S. P. Suresh for their unwavering support and invaluable guidance during the course of my Ph.D. degree. My interaction with them on Formal Methods began in the year 2012 with a reading assignment on a paper related to *Gossiping, Asynchronous Automata, and Zielonka's theorem*. These interactions eventually resulted in the work that I did on *Bounded Version Vectors* for my Master's thesis under the supervision of Madhavan and Suresh. Since then, I have been fortunate to work alongside them to explore several interesting problems in the areas of *weak consistency* and *conflict-free replicated data types (CRDTs)*. As someone with no prior grounding in Formal Methods, it was not an easy task for me to get acquainted with the fundamental principles of Automata Theory and Formal Verification. Madhavan and Suresh were instrumental in ensuring that I got the necessary exposure to the relevant literature in this field. They also helped me grasp the finer aspects of several key results in Automata Theory. My interactions with them over the years have helped me to better understand and appreciate the role of Formal Methods in Computer Science. In addition to their tutelage and their ability to create a friendly atmosphere where it was possible to have interesting technical discussions, both Madhavan and Suresh were very supportive when, in 2014, I decided to pursue my Ph.D. part-time. I remain ever-grateful for their patience and understanding which made it very easy for me to divide my time between my career, personal life, and my Ph.D. studies.

I would like to thank Prof. Narayan Kumar for being a part of my Doctoral Committee and providing valuable feedback during the Doctoral Committee meetings. I would also like to express my sincere gratitude to our collaborators, Prof. Ahmed Bouajjani and Prof. Constantin Enea from the University of Paris Diderot (Paris 7), for several stimulating discussions on the behaviours of data stores that satisfy weak consistency. These discussions eventually led to our work on formalizing and testing multilevel consistency.

I am very grateful to Tata Consultancy Services (TCS) for offering me a research scholarship from 2012 to 2014. I would like to thank the administrative staff of CMI, in particular, Rajeshwari Nair and S. Sripathy, for helping me with the administrative issues and also for the interesting conversations on books, movies, and music.

While in CMI, I was fortunate to be surrounded by a vibrant set of peers who ensured that there was never a dull moment in their company. For this, I thank Nikhil Balaji, Abhishek Bhrushundi, Nitesh Jha, Prakash Saivasan, Prateek Karandikar, Suryajith Chillara, Vaishnavi Sundararajan, Gopakumar Mohandas and Mithilesh Kumar.

I would like to thank Suresh Mallya and Sandesh Prabhu, who were my roommates in Chennai after I moved out of the CMI Hostels. Since both of them were from my hometown, staying with them was akin to staying at home! I would also like to offer special thanks to my cousin Naveen Shenoy, his wife Sandhya Shenoy, my good friend Madhusudan Rao, and his wife Arathi Rao for inviting me over to their homes during the weekends for good food and good times!

I am extremely grateful to my friends Vijay and Trupti Sukthankar, and Shuaib Shukur, for hosting me during my frequent trips to Bangalore between 2012 and 2013. More importantly, I would like to thank them for supporting me in every way possible when I made the not-so-easy decision to pursue my Ph.D. part-time. There are a few others who, despite not being around physically, provided the much-needed support in the form of phone and text conversations during this time. In particular, I want to thank Kevin Pais, Pooja Gundlur, Sathyaparathpara Chakravarthy, Balagopala Nair, Neethu Pillai, and Luvlyn and Lester D'Souza for being there for me. I would be remiss if I forgot to mention the help and encouragement provided by Maya and Jagadish Shenoy, who were our neighbours and good friends in Bangalore. When my wife was pregnant with our first child, thanks to the support provided by Maya and Jagadish Shenoy, I was able to continue visiting Chennai at regular intervals to discuss with my advisors.

In the year 2014, I joined the IBM Linux Technology Center (LTC) Bangalore, as a full-time Linux Kernel Developer. The LTC leadership team provided me with all the necessary support that enabled me to pursue my Ph.D. studies. They allowed me to spend one day every week in Chennai so that I did not miss out on any interactions with my advisors. They also provided me with the flexibility to take time off when I was working on a paper or travelling to a conference. I want to thank my team members from IBM LTC, especially my technical leads Dipankar Sarma, Vaidyanathan Srinivasan, Aneesh Kumar, my managers Krishna Prabhu, Tarundeep Singh Kalra, and Nilesh Joshi, and my colleagues Ananth Narayan MG, Bharata B. Rao, Shivaprasad Bhat, and Nikunj Dadhania for their consistent support.

I would like to acknowledge the influence that two remarkable individuals have had over my decision to pursue Doctoral studies. First, I want to thank my mathematics teacher Prof. Sudhakar Shetty, whose passion for exploring new ideas and for discovering new connections between existing ideas is something that I greatly admire. I would also like to thank Paul E. McKenney, a former Distinguished Engineer at IBM, who was my first collaborator at IBM LTC when I started my career as a rookie out of college in the year 2006. Paul has made several important contributions towards improving the scalability of the Linux Kernel on large concurrent systems. My interest in concurrent and distributed systems was piqued by

his work on the different variants of the *Read-Copy-Update (RCU)* synchronization primitive in the Linux Kernel. I am grateful for the advice and guidance that I have received from him at various points over the last fifteen years.

Finally, I want to thank my family for their unconditional support and their belief in me over all these years. In particular, I would like to express my gratitude to my parents Dr. Girish Shenoy and Nirmala Shenoy, my sister Sapna Shenoy and brother-in-law Chaitanya Nayak, for being a constant source of encouragement amid the various transformations in my professional and personal life. I am also greatly indebted to wife Shwetha Pai for her support, patience, and understanding, through all these years while I was pursuing the research work for this thesis. In the end, I want to thank our three children, Bhargav, Raghav, and Vaishnav, for kindly putting up with my absence in the late evenings and over the weekends in these last few months, which provided me with the much-needed time to complete this thesis.

|| *Śrī Kṛṣṇārpaṇamastu* ||

Contents

List of Figures

List of Tables

1

Introduction

The Internet hosts many services that maintain replicated copies of data across distributed servers which support local updates and queries. An early example is the Domain Name Service (DNS) that maintains a distributed mapping of Internet domain names to numeric IP addresses. This map is replicated across multiple servers across the different geographies. Clients typically access the server that is geographically closest to them. Since the same mapping is replicated across different servers, when the owner of a website chooses to change the mapping, it takes up to 72 hours for the change to be propagated across all the replicas which have cached this map.

The users of such services implemented in a distributed manner expect the following properties.

- **Consistency**: The distributed service behaves as if there is a single up-to-date copy of the data. Thus the queries are always expected to return the most up-to-date information and the updates are expected to be serialized against each other.
- **Availability**: Every request made by the client to the distributed service is eventually honored with a response.
- **Partition Tolerance**: The service continues to remain operational even when some of the replicas experience crash failures, or the network connectivity between the replicas is disrupted.

In the year 2000, Eric Brewer conjectured that there is a fundamental trade-off between the *consistency*, *availability* and *partition tolerance* properties: any distributed web-service, where the servers share data, can simultaneously satisfy at most two of these three properties [Brewer, 2000]. For example, single-site data bases and LDAP forgo *partition tolerance* and satisfy *consistency* and *availability* [Brewer, 2000] while distributed databases, services that implement distributed locking, and majority based protocols forgo *availability* and satisfy *consistency* and *partition tolerance* [Brewer, 2000]. *Brewer's Conjecture* was formally

proven by Gilbert and Lynch in [Gilbert and Lynch, 2002] and the trade-off has been popularly known as the *CAP theorem* ever since.

Given that several distributed services such as virtual shopping carts of online merchants like Amazon and Content Distribution Networks (CDNs) such as Netflix have data that is replicated across the globe, these service need to be robust against network partitions and replicas becoming inaccessible. Hence, *partition tolerance* is a critical feature that ought to be supported by such services. Furthermore, high availability is critical to the business model of many of the services such as content distribution networks and social networks. Thus, it may appear that such services will not be able to guarantee the *consistency* property.

However, it must be noted that the definition of *consistency* in both [Brewer, 2000] and [Gilbert and Lynch, 2002] is a strong consistency criterion, namely *linearizability* [Herlihy and Wing, 1990; Herlihy, 2008]. Thus, distributed data services which need to satisfy *availability* and *partition tolerance* only need to forgo strong consistency, where local queries about distributed objects return answers consistent with the most recent update [Gilbert and Lynch, 2002]. They can instead support weaker consistency criteria. There are many well understood weaker consistency criteria including, but not limited to, *Basic Eventual Consistency* [Shapiro and Kemme, 2009; Saito and Shapiro, 2005; Vogels, 2008], *Session Guarantees* such as *Read-Your-Writes*, *Monotonic Reads*, *Write Follows Reads*, *Monotonic Writes* [Terry et al., 1994], *Strong Eventual Consistency* [Shapiro et al., 2011a,b], *FIFO Consistency* [CS551, 2001] and *Causal Consistency* [Lamport, 1979; Perrin et al., 2016; Bouajjani et al., 2017] that are used in distributed data stores. We shall now explain some of the weak consistency criteria using the example of a *timeline* of the popular social network Twitter.

The Twitter service allows users to post short messages, known as *tweets*, on their website or app. It allows users to *follow* other users. A user U_1's *timeline* is a sequence of *tweets* from U_1 as the *tweets* of those other users whom U_1 follows. These *tweets* are arranged in the decreasing order of the global timestamps. The user U_1 can *retweet* the tweet made by some other user U_2 (whom U_1 may or may not follow). When this happen, that *tweet* of U_2 becomes visible in the timeline of U_1 as well as in the timelines of all the users U_3 who follow U_1. Similarly user U_1 can *comment* on a tweet made by some other user U_2. Each *comment* itself is another *tweet* which is linked to the original tweet. We can imagine that the Twitter service is implemented as a distributed system where the users *tweets* are replicated across various servers across the globe. We refer to these servers as *replicas*. The users of the service are unaware of the existence of the replicas. Thus their interaction with the service could be handled by any one of the replicas spread across the world.

- **Basic Eventual Consistency**: This criteria states that if no new updates are made to the data-service, then all the queries made to the data-service will return the last updated value. This is more of a *liveness* property. If Twitter were to satisfy *Basic Eventual Consistency*, it would have to ensure that if a user U_1 stops posting any more

tweets then eventually all the followers of U_1 will see all of U_1's tweets and retweets in their timeline.

- **Read-Your-Writes**: This is a *session-guarantee* that requires that effects of prior operations in the session are visible to later operations in the same session. A session in the context of twitter is an instance where the user has logged into the twitter service from a browser window or a mobile app. Each distinct login on a different window or a differnt app on a device is treated as a different session. If Twitter were to satisfy *Read-Your-Writes*, it would have to ensure that every tweet made by a user U_1 in a given session would be visible in the timeline of that session. It makes no guarantee of those tweets being visible on any other session that the user U_1 may have open on the browser or on the Twitter app on another device.

- **Monotonic-Reads**: This is also a *session-guarantee* that requires that once the effect of an operation becomes visible within a session, it remains visible to all subsequent operations in that session. If Twitter were to satisfy *Monotonic-Reads*, it would have to ensure that once some tweet is visible in user U_1's session, it would continue to remain visible for the rest of that session.

- **Strong Eventual Consistency**: This requires that replicas which have received the same set of updates should have equivalent states, no matter in what order they received those updates. If Twitter were to satisfy *Strong Eventual Consistency*, then for any two users U_1 and U_2 who follow each other and who follow the same set of other users, if the timelines of U_1 and U_2 are respectively constructed from replicas r_1 and r_2, and both r_1 and r_2 have received the same set of *tweets*, then the timelines of U_1 and U_2 should be identical.

- **Causal Consistency**: This requires that effects of prior operations in a session are always visible to later operations. Further, if the effect of an operation is visible to another operation, then every operation that has seen the effects of the latter would have seen the effects of the former. Suppose a user U_1 posts a tweet and another user U_2 comments on it. Another user U_3 comments on U_2's comment. If Twitter were to implement *Causal Consistency* then any user U_4 who follows U_3 and see's U_3's comment on their timeline should also see U_2's comment and U_1's tweet.

Among these, *Strong Eventual Consistency* is a variant of *Eventual Consistency* that requires that all replicas which have received the same update messages have equivalent states, irrespective of the order in which they received those update messages. Hence, data stores that satisfy *Strong Eventual Consistency* are robust and conflict-free. Shapiro et al. have studied a class of data structures called the *Conflict-free Replicated Data Types (CRDTs)* are designed to satisfy strong eventual consistency by construction [Shapiro et al., 2011a,b]. This

class includes widely used data types such as replicated counters, sets, registers, and certain kinds of graphs. In the next section we provide an introduction to conflict free replicated data types using some of the popular data types.

1.1 Replicated Datatypes

Consider the abstract data type *counter*. It provides three methods for the users to interact with it.

- Inc() : Increments the value of the counter by one.
- Dec() : Decrements the value of the counter by one.
- Fetch() : Get the current value of the counter.

In a sequential setting, an implementation of the counter would typically have a memory location modelling the counter. The implementation would also provide functions corresponding to the Inc(), Dec() and Fetch() methods to respectively increment, decrement and read the value of that memory location. Such an implementation would be referred to as a *sequential implementation* of the counter. The behaviour of the sequential implementation of the counter should satisfy the well understood sequential specification of the counter.

However the abstract counter can also be implemented over a distributed system with N nodes, where each node would contain a replica of the counter. Such an implementation is referred to as a *replicated implementation* of the counter. When we say *replicated counter*, we mean this replicated implementation of the counter. We shall now see one such replicated counter named the *Positive Negative Counter (PN-Counter)* [Shapiro et al., 2011b].

In the *PN-Counter*, when a client issues an Inc(), Dec(), or a Read() request, these requests could be redirected to any of the N replicas. The client has no say in the matter. When a replica receives a Fetch() request from a client, it responds to the request based on its current local state. We say that this replica is the *source replica* for that Fetch() request.

When a replica receives an Inc() or a Dec() request from some client, it updates its local state to cause the value of the counter to increment by one (in case of a Inc() request) or decrement by one (in case of a Dec() request) . However, since this is a replicated counter, the other replicas must be informed of this Inc() or Dec() operation. For this purpose, the replicas can follow one of two strategies to inform the other replicas about these requests.

- It can send a broadcast message to all the other replicas indicating that it has received an increment/decrement request. The other replicas on receiving this broadcast will update their respective states in order to incorporate the effect of this increment/decrement.

- Otherwise, the replica can periodically share its entire state with the others replicas in the system. The other replicas, on receiving this state update their respective local states by *merging* the contents of the remote state that they have just received into their local state.

If the *PN-counter* implementation follows the first strategy of broadcasting a message after every Inc() or Dec(), then it needs to rely upon the underlying network that the broadcast messages are not lost. It also needs to take precaution to handle the cases when the same broadcast message is delivered more than once unless the underlying network guarantees that there will not be duplicated delivery.

However sending a broadcast message after every operation will flood the network and may not be desirable in certain scenarios. In such cases, the *PN-counter* implementation could follow the second strategy and periodically broadcast their entire state to the remote replicas. This state would contain information along with some meta-data about the increments and decrements. The recipient replicas can use this information to update their own states. While the network will have considerably lesser traffic in this strategy compared to the earlier one, the size of the paylaod that needs to be communicated to the other replicas is relatively higher. This is because the state could contain additional meta data which will help the recipient replicas to safely merge this state into their states without causing over-accounting or under-accounting of the counter value.

Suppose the underlying network provides a guarantee against message loss and duplicate message delivery. Then, a *PN-counter* that follows the first strategy can maintain a individual counter at every replica. When the replica receives an Inc() (respectively a Dec()) request from the client, it can increment (respectively decrement) its local counter value. It can then send a broadcast pertaining to the increment (respectively decrement) to the other replicas. Those replicas, on receiving this increment broadcast (respectively decrement broadcast), can increment (respectively decrement) their respectively local counters. When a replica receives a Fetch() request from some client, it returns the value of its local counter. It can be observed that with this implementation, when all the replicas become aware of all the increment and decrement requests, either through the clients or through the respective broadcast messages, their local counters will have the same value. Thus this implementation satisfies the requirement of *strong eventual consistency*. Such an implementation where the replicas send broadcast messages after every Inc() and Dec() request is known as an *operation based PN-counter*.

On the other hand, if the *PN-counter* implementation chooses not to send a broadcast message on every Inc() or Dec() they can go for the following implementation.

- Every replica $i \in [1, \ldots, N]$ maintains two arrays $I_i[1:N]$ and $D_i[1:N]$. The value $I_i[j]$ denotes the number of increments at the replica $j \in [1, \ldots, N]$ that i is aware

of. Similarly the value $D_i[j]$ denotes the number of decrements at a replica j that i is aware of. Initially $I_i[j] = D_i[j] = 0$ for all i, j.

- On receiving an Inc() request from the client, replica i increments the value $I_i[i]$ by one, i.e
$$I_i[i] := I_i[i] + 1$$
- On receiving an Dec() request from the client, replica i increments the value $D_i[i]$ by one. Thus
$$D_i[i] := D_i[i] + 1$$
- Periodically replica i broadcasts its state $(I_i[1:N], D_i[1:N])$ to the other replicas in the system.
- Replica j on receiving $(I_i[1:N], D_i[1:N])$ will update its own state as follows
  - For all $k \in [1, \ldots, N]$, set $I_j[k] := max(I_i[k], I_j[k])$
  - For all $k \in [1, \ldots, N]$, set $D_j[k] := max(D_i[k], D_j[k])$
- On receiving a Fetch() request from the client, replica i would return the value
$$(\Sigma_{k=1}^{N} I_i[k]) - (\Sigma_{k=1}^{N} D_i[k])$$

Thus, in this case, the state of each replica of the replicated counter keeps a count of the number of increment and decrement requests sent by the client to every replica in the system. Periodically the replica shares its entire state to the other replica in the system. The recipient replicas can safely update their knowledge of the number of increment and decrement requests at every other replica based on information of the state that it just received. In this implementation, it can be noted that once all the replicas share their states with every other replica, their local states will be identical. Thus this implementation of the *PN-counter* satisfies *strong eventual consistency*. Since the replicas send their entire states to the other replicas, this kind of an implementation is called as a *state based PN-counter*.

Now, since by definition, the Inc() and the Dec() operations of the *PN-counter* commute, the order in which a replica receives the broadcast message corresponding to these requests will not matter to the final state of the replica of *operation based replicated data type*. However, this need not be true for some of the other replicated data type implementations that satisfy strong eventual consistency. In order to illustrate this, we provide the example of a replicated ipmplementation of a *Read-Write register* known as *Last Writer Wins Register (LWW Register)*.

A *register* is an abstract data type that provides two methods for the users to interact with it.

- Write() method, which allows the users to update the value of the register with a new value. Let us assume that the values come from the set of natural numbers $\mathbb{N}$.

- Read() method, which allows the users to query the current value of the register.

The *LWW Registers* is a replicated implementation of a *register*. As in the case of *PN-Counter*, the *LWW-Register* is implemented over a distributed system with N replicas $[1, \ldots, N]$ where replica contains a copy of the register. When a replica receives a Read() request from some client, it returns a value based on its current local state. When a replica receives Write() request from a client, it will update its local state. Like in the case of the *PN-Counter*, it can either choose to broadcast this information to all the other replicas (*operation based LWW Register*), or, it can periodically broadcast its entire state to the other replicas, which on receiving the state will update their own state by merging the contents of the received state into their local states (*state based LWW Register*).

Unlike the *operation based PN-Counter*, it is not sufficient for an *operation based LWW-Register* to just maintain a register as a part of its local state. We explain this with an example. Consider the case where a client sends a Write(3) request to a replica i. Replica i updates its local register $value^i = 3$. It then broadcasts the information that it has received a Write(3) to every other replica in the system. Concurrently, another replica j receives the Write(4) request from a client. It processes the Write(4) request by updating its local state to $value^j := 4$. It then broadcasts the information pertaining to this Write(4) to all the other replicas. Eventually, replica i will receive the broadcast message from replica j saying that it has applied Write(4) locally. Likewise, replica j will receive the broadcast message from i regarding Write(3). Thus the replica i and j will have to decide if they want to overwrite their local register with the value corresponding to the broadcast or if they want to ignore the value corresponding to the broadcast. If they choose to overwrite, the local state of replica i will be $value^i = 4$ and the local state of replica j will be $value^j = 3$. Thus the states have diverged. Similarly if they both choose to ignore the values corresponding to the broadcast, then the local states of i and j will respectively be $value^i = 3$ and $value^j = 4$. In this case too, the states have diverged. Thus, even though both the replicas i and j have received the same set of updates, their states not equivalent.

However if replica i were to overwrite its local value with the value corresponding to the broadcast and replica j were to ignore the value corresponding to the broadcast, the local states of both the replicas would have converged to the same value. Thus, what is needed in this case is a strategy that allows the replicas to arbitrate between concurrent Write() requests in a uniform manner. We present one such strategy below.

Operation-based LWW Register

Assume that each replica has a clock that produces a monotonically increasing sequence of timestamps from TS. We can assume that the clocks across the different replicas progress more or less at the same rate, but are allowed to drift and will periodically synchronise using protocols such as NTP.

- State at replica $i \in [1, \ldots, N]$ is an array of pairs $[(value^i_j, t^i_j) \mid j \in [1, \ldots, N]]$ where $value^i_j \in \mathbb{N} \cup \{\perp\}$ denotes the latest value written by replica j that replica i is aware of. $t^i_j \in TS$ is the timestamp corresponding to the latest Write() request at replica j that the replica i is aware of. Initial value $[(value^i_j = \perp, t^i_j = 0) \mid j \in [1, \ldots, N]]$.
- When replica i receives Write($value$) request from the client it performs the following.
  - Let $t = getTS()$.
  - Set $(value^i_i, t^i_i) := (value, t)$
  - Broadcast $(value, i, t)$ to all the replicas.
- When a replica j receives $(value, i, t)$, it does the following
  - If $t > t^j_i$
    - Set $(value^j_i, t^j_i) := (value, t)$
- When the replica i receives a Read() request from the clients, it computes the response and returns the value as follows.
  - Let $j \in [1, \ldots, N]$ be such that $\forall k \in [1, \ldots, N] : (t^i_j > t^i_k) \vee ((t^i_j = t^i_k) \wedge j > k)$
  - Return $value^i_j$

When the replica i receives a Write request from some client to write a new value $value$, it generates a new timestamp t based on the current value of its local clock. Replica i then updates $(value^i_i, t^i_i)$ to $(value, t)$ to take into account the latest Write method that it locally applied. It then broadcasts the tuple $(value, i, t)$ to inform all the other replicas about this Write request.

When a replica j receives this broadcast message $(value, i, t)$, it will check if the timestamp t is greater than the timestamp t^j_i corresponding to the latest Write at i that it is currently aware of. If so, it will update $(value^j_i, t^j_i)$ to $(value, t)$.

Finally, when a replica i receives a Read request from the client, it determines the latest timestamp corresponding to any Write performed in the system that it is currently aware of. Suppose t^i_j is such a timestamp for some replica j. Suppose there is another replica k which, as per i's knowledge, has performed a write at the exact timestamp as j, then we pick the

replica with the larger index. Thus having determined the replica which has performed the most recent Write, the Read method at i returns the value $value^i_j$ written by that Write.

It can be observed that any pair of replicas i, j which have received the same information pertaining to the same set of Write operations will have $(value^i_k, t^i_k) = (value^j_k, t^j_k)$ for all $k \in [1, \ldots, N]$. Thus, they both will return the same value when a Read method is applied at them. Hence this *operation based LWW-Register* implementation satisfies *strong eventual consistency*. We can also describe a *state based LWW-Register* implementation which would be useful when it is desirable to keep the network traffic to a minimum.

State-based LWW Register

- State at replica $i \in [1, \ldots, N]$ is an array of pairs $[(value^i_j, t^i_j) \mid j \in [1, \ldots, N]]$ where $value^i_j \in \mathbb{N} \cup \{\perp\}$ denotes the latest value written by replica j that replica i is aware of. $t^i_j \in TS$ is the timestamp corresponding to the latest Write() request at replica j that the replica i is aware of. Initial value $[(value^i_j = \perp, t^i_j = 0) \mid j \in [1, \ldots, N]]$.
- When replica i receives Write($value$) request from the client it performs the following.
  - Let $t = getTS()$.
  - Set $(value^i_i, t^i_i) := (value, t)$.
- Periodically replica i sends its entire current state $[(value^i_k, t^i_k) \mid k \in [1, \ldots, N]]$.
- Suppose a replica j receives the state of i , i.e $[(value^i_k, t^i_k) \mid k \in [1, \ldots, N]]$. It will update its local state as follows.
  - For $k \in [1, \ldots, N]$ such that $t^i_k > t^j_k$
    - Set $(value^j_k, t^j_k) := (value^i_k, t^i_k)$
- Read() method applied at replica i:
  - Let $j \in [1, \ldots, N]$ be a replica such that

$$\forall k \in [1, \ldots, N] : (t^i_j > t^i_k) \vee ((t^i_j = t^i_k) \wedge j > k)$$

  - Return $value^i_j$

In the *state-based* implementation, the local state of the replicas is the same as that in the case of the *operation-based* implementation described earlier. When a replica i gets the Read() request from a client, it computes the response in the same manner as the *operation-based* implementation. On receiving a Write request, the replica updates its local state in a manner similar to the earlier *operation-based* implementation. The only difference is that

after updating the local state, it doesn't broadcast the message. Instead, the replicas can periodically send their current state to other replicas. When a replica j receives the state of replica i, then, for every replica $k \in [1, \ldots, N]$, j checks if i has a more recent knowledge of the latest Write at k by comparing the timestamps t_k^i with t_k^j. If so, it will update $(value_k^j, t_k^j)$ to $(value_k^j, t_k^j)$. Thus, in this implementation, once all the replicas have sent their states to all the other replicas, the local states of every replica will be the same. The monotonically increasing timestamps along with the replica indices help every replica to determine which was the *last writer* in an unambiguous manner. In particular, every replica will order a pair of concurrent Write() requests in a similar way.

The *PN-Counter* and *LWW-Register* implementations described here are examples of a class of replicated implementations of data types known as *Conflict-free Replicated Data Types (CRDTs)*. Prior works such as [Shapiro et al., 2011a,b] provide a comprehensive survey of the replicated implementations of various kinds of conflict-free replicated data types including counters, registers, sets and graphs. All these replicated implementations satisfy the *strong eventual consistency* property. Apart from the simpler CRDTs mentioned above, there also exists replicated implementations of complex data types such as JSON [Kleppmann and Beresford, 2017; Grosch et al., 2020; Brocco, 2021], and word sequences, which are the building blocks of collaborative editing tools [Attiya et al., 2016; Briot et al., 2016; Grishchenko and Patrakeev, 2020; Nicolas et al., 2020]. There are frameworks that allow decompostion of complex CRDTs as a semi-direct product of simpler ones, and construction of novel CRDTs by combining the operations of pairs of CRDTs [Weidner et al., 2020]. Real world data stores such as RiakDB support several CRDTS including counters, flags, register, sets and maps [Riak, 2015]. There are programming models for large scale distributed programming such as Lasp [Meiklejohn and Van Roy, 2015] and CScript [De Porre et al., 2020] that support replicated data types as first class objects. There is also active work in the fields of formal specifications [Burckhardt, 2014; Burkhardt et al., 2014] and formal verification of these replicated data types [Burkhardt et al., 2014; Mukund et al., 2015a,b; Gomes et al., 2017; Blau, 2020].

In this thesis we undertake the study the replicated implementations of data types (henceforth referred to as replicated data types) from the following perspectives:

1. Constructing replicated implementations of abstract data types that satisfy strong eventual consistency.

2. Formulating concurrent specifications for abstract data types independent of implementation details.

3. Verifying the correctness of a given replicated data type with respect to its concurrent specification. The replicated implementations of data type describe the behaviour of the data-types from the perspective of the communicating replicas.

4. Verifying the correctness of the behaviours of the replicated data types as seen by the clients. Since the clients are oblivious to the implementation details such as the number of replicas involved, the frequency at which they communicate, the constraints on message delivery, the behaviours as observed by them are modelled as the sequences of method-invocations made by the clients and the responses they receive from the data types.

In this thesis we also explore the applicability of concepts and tools from traditional trace theory and automata theory, and wherever required, adapt them and extend them in the study and analysis of relatively modern replicated data types.

The thesis is organized as follows. In the next chapter we provide a formal model for replicated data types and define some of the terminology associated with them. We also prove two key results from [Shapiro et al., 2011b] that describe the sufficient conditions for replicated data types to satisfy *strong eventual consistency*. In Chapter 3 we study a well known CRDT named *Observed-Remove Sets (OR-Sets)*. We describe three different implementations of OR-Sets that require different kinds of data to be replicas to correctly model the OR-Set for different delivery guarantees. We also provide a formal specification for OR-Set which captures the behaviours of all the three implementations. We demonstrate the construction of an optimized implementation of OR-Sets using a novel object known as *Interval Version Vector*. This chapter is based on our published work [Mukund et al., 2014] which describes an optimized OR-Set in the absence of any delivery constraints on the network. In Chapter 4, we introduce a formal framework for describing the behaviours of replicated data types independent of their implementations. Using this we provide declarative specifications of some of the well known CRDTs. We also provide a principled approach for constructing a reference implementation from their declarative specification. In Chapter 5 we show how a bounded distributed reference implementation can be constructed for certain classes of replicated data types. In order to achieve this, we define a generalization of the classical *gossip problem*, whose bounded solution is used to construct the bounded reference implementations. Chapters 4 and 5 are based on our published work on bounded implementations of replicated data types [Mukund et al., 2015a]. In Chapter 6 we provide methodology to construct a simpler global reference implementation from its declarative specification whose state space is bounded for certain classes of declarative specification. We use the concept of the *Later Appearance Record(LAR)* from the automata theory to arrive at the global reference implementation. We describe how these bounded reference implementations can help in the formal verification of a given implementation of a CRDT. This chapter is based on our published work on effective verification of replicated data types using *Later Appearance Records(LAR)* [Mukund et al., 2015b]. Chapters 5 and 6 focus on the verification of the correctness of the behaviours replicated data types from the perspective of the communicating replicas. In Chapter 7 we turn our attention towards checking the

correctness of the behaviours of replicated data types as seen by the clients. In particular, we look at distributed read-write stores which offer the choice of multiple consistency levels to the clients. We formalize the notion of multilevel-consistency for these data stores and provide an algorithm to test the correctness of such data stores against the consistency level and their combinations. This chapter is based on our published work on formalizing and checking multilevel consistency [Bouajjani et al., 2020]. In the final chapter of the thesis we will summarize our findings and describe a few new problems that have emerged from our study of replicated data types.

2

A Formal Model for CRDTs

In this chapter we describe a formal model for replicated data types in general and Conflict-Free Replicated Data Types (CRDTs) in particular. In the next section, formally define *replicated data types* and some terminology associated with them. While the terminology used in this chapter is from [Shapiro et al., 2011a,b], the formalism uused here is a synthesis of the formalism presented in our prior works [Mukund et al., 2014, 2015a,b]. We will be using this formalism throughout this thesis. In section 2.2 of this chapter, we provide new rigorous proofs for the two theorems from [Shapiro et al., 2011a] which describe the sufficient conditions for two flavours of CRDTs namely the *Convergent Replicated Data Types (CvRDTs)* and the *Commutative Replicated Data Types (CmRDTs)*.

2.1 Definitions

Replicated data types are replicated implementations of abstract data types over a distributed system with a finite number of servers, each of which hosts a replica of the data type. We shall use the term servers or replicas interchangeably to refer to the replica of a distributed system. The set of replicas is denoted by $\mathcal{R}$ which is a finite set of natural numbers $[1..N]$. These numbers identify the distinct replicas. We shall use the variables r, r', r'', r_i, r_j to refer to the individual replicas. We shall use the example of *LWW Register* to introduce the technical terminology associated with CRDTs.

The *LWW Register* is a replicated implementation of the abstract data type known as *Register*. We had previously mentioned that each value written to the register is a natural number from $\mathbb{N}$. We assumed that the initial value of the register is a special value denoted by $\perp$. Thus, $\mathbb{N} \cup \{\perp\}$ forms the *universe* of the underlying set of values that can be present in the register. Further, $\mathbb{N} \cup \{\perp\}$ also forms the set of values that can be *returned* as a response to the Read method. As we have seen, the Read() method returns a response but does not modify the state of the replica. In general we shall denote such methods as *queries*. The

Write() method modifies the state of the replica. However it does not return a value. Such methods are known as *updates*. In this thesis, we shall restrict our attention to only those data types whose whose *update* methods do not return any value. With this, we present the formal definition of an abstract data type below.

Definition 1 (Abstract Data Type (ADT))**.** *An abstract data type $\mathcal{D}$ is a tuple* (Univ, Queries, Updates, Rets) *where*

- Univ *is the underlying set of values stored in the datatype and is called the* universe *of the datatype. For example $\mathbb{N} \cup \{\perp\}$ forms the universe of the* LWW Register*.*
- Queries *denotes the set of query methods exposed by the data type. The* Read *method is the query method of the* LWW Register*.*
- Updates *denotes the set of update methods. The* Write *method is the update method of the* LWW Register*.*
- Rets *is the set of all return values for queries. In the case of* LWW Registers*, the set of return values is $\mathbb{N} \cup \{\perp\}$.*

We assume that $\perp$ is a designated "empty value", belonging to both Univ *and* Rets*.*

In the case of a replicated data type, when a client invokes a *query* (Read in the case of *LWW Register*) or an *update* (Write in the case of *LWW Register*) method, it is performed on any one of the N replicas of the replicated data type. That replica is referred to as the *source replica* for that method. In the case of the *query* method, the source replica returns a response based on its local state. In the case of an *update* method, the replica first updates its local state and subsequently communicates with the other replicas. The replicas communicate with each other in a couple of ways.

1. Each time a replica r receives an *update* request, it applies the update locally and generates some auxiliary information pertaining to the update. It then broadcasts that auxiliary information to all the other replicas. We will model this broadcast to be performed via a method known as *update-send*. Any of those other replicas, say r', on receiving this broadcast will update their local states using the auxiliary information. We shall model this as via the *update-receive* method associated with that particular *update* request.
2. From time to time, a replica r can share its current state to some other replica r'. We model this using a *merge-send* method. The other replica r' on receiving this state, merges it into its current state, thereby updating the state through the *merge-receive* method.

A replicated data type where the source replica communicates with the other replicas after every *update* via the *update-send* and the *update-receive* methods is known as an *operation-based replicated data type*. A replicated data type where the replicas communicate with the other replicas by sending and incorporating the entire state via the *merge-send* and *merge-receive* are known as *state-based replicated data types*.

We shall now define some technical terms which will be used to reason about the behaviours of the *state-based* and *operation based* replicated data types. Towards this, we model the query and update methods, the update send and receive methods, and the merge send and receive methods as *operations*.

Definition 2 (Operations)**.** *An operation of a Replicated Data Type*

$$\mathcal{D} = (\mathsf{Univ}, \mathsf{Queries}, \mathsf{Updates}, \mathsf{Rets})$$

is a tuple $o = (op, r, args, ret)$ *where*

- $op \in \mathsf{Queries} \cup \mathsf{Updates} \cup \mathsf{UComm} \cup \mathsf{MComm}$ *is the method or action where*
  - $\mathsf{UComm} = \{\mathbf{usend}, \mathbf{ureceive}\}$ *where* **usend** *is the corresponding* update-send *method for an update operation and* **ureceive** *is the corresponding update-receive methods for an update operation. If* o *is an* update *operation and* o_{snd} *and* o_{rcv} *respectively are the corresponding* update-send *and* update-receive *operations then we say that* o *is the* matching update operation *of* o_{snd} *and* o_{rcv}*.*
  - $\mathsf{MComm} = \{\mathbf{msend}, \mathbf{mreceive}\}$ *are the send and receive operations for merge. If* o_{msnd} *is a* merge-send *operation and* o_{mrcv} *is the corresponding* merge-receive *operation at a remote replica, then* o_{msnd} *is said to be the* matching merge-send *operation of* o_{mrcv}*.*
- $r \in \mathcal{R}$ *is the source replica where the operation is performed.*
- $args$ *is a tuple of arguments from* Univ*,*
- $ret \in \mathsf{Rets}$ *is the return value*

Operations satisfy the following conditions:

- *if* $op \in \mathsf{Updates}$*,* $ret = \perp$*. This captures that fact that* update *methods do not return a value to the clients.*
- *if* $op \in \mathsf{UComm} \cup \mathsf{MComm}$*,* $args = ret = \perp$*. The auxilliary information sent to all the replicas by the* update-send *method varies from implementation to implementation. Similarly the arguments received by the* update-receive *method varies across implementations. We shall model them separately later in this chapter. Similarly the*

information sent and received by merge-send *and* merge-receive *methods correspond to the state of the replica performing the* merge-send. *This varies across implementations. We shall model these separately.*

For an operation $o = (op, r, args, ret)$, we define $Op(o) = op$, $Args(o) = args$, $Rep(o) = r$, and $Ret(o) = ret$. We call o a query operation *if $op \in \mathsf{Queries}$, an* update operation *if $op \in \mathsf{Updates}$, an* update-send operation *if $op =$* **usend**, *an* update-receive operation *if $op =$* **ureceive**, *a* merge-send operation *if $op =$* **msend** *and a* merge-receive operation *if $op =$* **mreceive**.

We denote the set of operations of $\mathcal{D}$ by $\Sigma(\mathcal{D})$.

In the previous section we saw that the *operation-based LWW-Register* implementation maintains some local state. It further provides a function to update the local state on receiving a $\mathsf{Write}()$ request from the client. It determines that on every $\mathsf{Write}()$ the tuple consisting of the value to be written, the replica identifier and the timestamp should be broadcast to all the other replicas. It also provides a function to update the local state on receiving this broadcast message. Finally it provides a function that computes the response to a $\mathsf{Read}()$ request by inspecting the local state of the replica. In general an operation based implementation of a replicated data type should define what the local states should be and what should be performed on receiving a query or an update request from the client, what should be the auxiliary information generated for every update and how should a remote replica incorporate this auxiliary information which would be presented to it in via an *update-receive*. We define an *operation-based* implementation of a replicated data type as follows.

Definition 3 (Operation-based Replicated Data Type Implementation)**.** *An* operation-based *implementation of the* replicated data type $\mathcal{D} = (\mathsf{Univ}, \mathsf{Queries}, \mathsf{Updates}, \mathsf{Rets})$ *is a tuple $\mathcal{D}_I = (\mathcal{S}, S_\perp, \Gamma, F_{\mathsf{Queries}}, F_{\mathsf{Updates}})$ where*

- *$\mathcal{S}$ is set of states that each of the replicas can take.*
- *$S_\perp \in \mathcal{S}$ is the initial state at every replica.*
- *Γ is the set of auxiliary information generated by the update to be sent to all the other replicas.*
- *$F_{\mathsf{Queries}} = \{f_q : q \in \mathsf{Queries}\}$ is a set of functions corresponding to the query methods in* $\mathsf{Queries}$, *where each function $f_q : \mathsf{Univ}^* \times \mathcal{S} \to \mathsf{Univ}$ is an implementation of the query method q as described earlier*
- *$F_{\mathsf{Updates}} = \{(f_u, f_u^{rcv}) : u \in \mathsf{Updates}\}$ is a set of pairs of computable functions corresponding to the update methods in* $\mathsf{Updates}$. *The function $f_u : \mathsf{Univ}^* \times \mathcal{S} \to \mathcal{S} \times \Gamma$*

is an implementation of the update method $u \in \mathsf{Updates}$ *that take arguments from* Univ^* *to transforms the state of its source replica into a new state. This function also generates auxiliary information to be propagated to the other replicas as a part of the* update-send *method.* $f_u^{rcv} : \Gamma \times \mathcal{S} \to \mathcal{S}$ *is a computable function corresponding to the* update-receive *method which takes as argument the auxiliary information generated by the* f_u *and transforms the state of a remote replica to a new state.*

- *Given an update method* $u \in \mathcal{U}$*, the corresponding functions* (f_u, f_u^{rcv}) *are such that for any state* $S_i \in \mathcal{S}$ *and any argument* $args \in \mathsf{Univ}^*$ *and* $aux \in \Gamma$*, if* $f_u(args, S_i) = (S_i', aux)$ *and* $f_u^{rcv}(aux, S_j) = S_j'$*, then* $S_i = S_j \implies S_i' = S_j'$*. This captures the requirement that given a pair of replicas which are at the same state, if an update operation is applied to one of them and the* update-receive *operation at the other will result in the same state at both the replicas.*

The difference between an *operation-based LWW Registers* and *state-based LWW Registers* is that instead of sending auxiliary information on every $\mathsf{Write}()$ *update* request the replicas periodically send their entire state. Thus the state based implementation should provide a function that allows the replicas to merge into their local state, the remote state which they receive via a *merge-receive* method. We now formally describe the *state-based* implementation of a replicated data type.

Definition 4 (State-based Replicated Data Type Implementation)**.** *A* state-based *implementation of the* replicated data type $\mathcal{D} = (\mathsf{Univ}, \mathsf{Queries}, \mathsf{Updates}, \mathsf{Rets})$ *is*

$$\mathcal{D}_I = (\mathcal{S}, S_\perp, F_{\mathsf{Queries}}, F_{\mathsf{Updates}}, f_{\mathcal{M}})$$

where

- $\mathcal{S}$ *is set of states that each of the replicas can take.*
- $S_\perp \in \mathcal{S}$ *is the initial state at every replica.*
- $F_{\mathsf{Queries}} = \{f_q : q \in \mathsf{Queries}\}$ *is a set of functions corresponding to the query methods in* $\mathsf{Queries}$*. Each* $f_q : \mathsf{Univ}^* \times \mathcal{S} \to \mathsf{Univ}$ *is an implementation of the query method* q *that computes the return value of that query with arguments from* Univ^* *when applied at a state in* $\mathcal{S}$*.*
- $F_{\mathsf{Updates}} = \{f_u : u \in \mathsf{Updates}\}$ *is a set of computable functions corresponding to the update methods in* $\mathsf{Updates}$*. The function* $f_u : \mathsf{Univ}^* \times \mathcal{S} \to \mathcal{S}$ *is an implementation of the update method* $u \in \mathsf{Updates}$ *that take arguments from* Univ^* *to transforms a given state of a replica to a new state.*

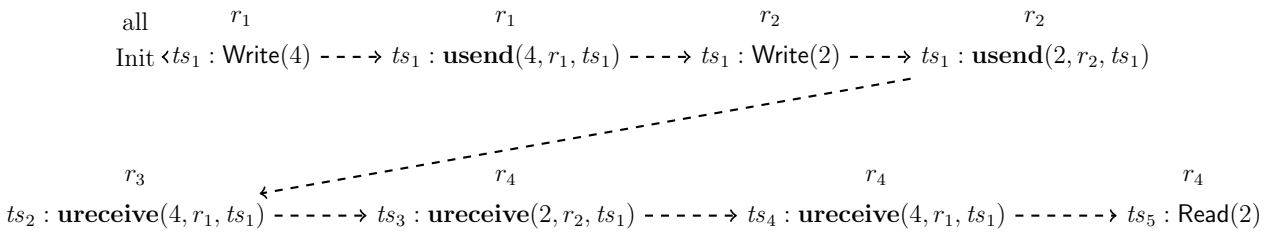


Figure 2.1: Behaviour of LWW register

- *$f_{\mathcal{M}}$ is the implementation of the* merge-receive *method which computes the merger of a pair of states to produce a new state. Formally $f_{\mathcal{M}} : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$.*

In this thesis, we are interested in analyzing the *behaviours* of implementations of replicated data types, and reasoning about their correctness. An example of an observable behaviour of a *LWW Register* is presented in Figure 2.1. There is an initialization operation *Init* which initializes the states of all the replicas. At timestamp ts_1, replica r_1 gets a Write(4) request from the client. r_1 would update its local state and propagate $(4, r_1, ts_1)$ to all the replicas via **usend**. Concurrently, at timestamp ts_1, the replica r_2 receives a Write(2) request from the client. It would also update its local state and broadcast $(2, r_2, ts_1)$ to all the replicas via **usend**. At ts_2, replica r_3 receives the broadcast sent by r_1. It then updates its local state via **ureceive** method to incorporate this new information. At ts_3 and ts_4 the replica r_4 receives the broadcast sent by r_2 and r_1 respectively. The replica r_4 will update its local state via the **ureceive** method. Finally at ts_5, replica r_4 receives a Read() query from the client. The local state of r_4 at this point is as follows:

- $(value_1^4, t_1^4) = (4, ts_1)$
- $(value_2^4, t_2^4) = (2, ts_1)$
- $(value_3^4, t_3^4) = (\perp, 0)$
- $(value_4^4, t_4^4) = (\perp, 0)$

Thus, the maximum timestamp that r_4 is aware of is ts_1 and there are two replicas r_1 and r_2 which performed the Write() operation at ts_1. In this case the arbitration is performed based on the maximality of the replica-id. Hence the replica r_2 arbitrates in favour of the value written by r_2. Hence the Read() request at r_4 returns the value 2.

In general, the behaviour of a replicated data type can be defined from the perspective of the replicas, in the order in which they process the *queries* and the *updates* from the clients, and the order in which they propagate information about the *updates* to other replicas. The behaviour should also capture the communication between the replicas via *update-send*,

update-receive in the case of *operation based replicated data types* or *merge-send* and *merge-receive* in the case of *state based replicated data types*. Furthermore, from the behaviour of an *operation based* replicated data type, we should be able to unambiguously determine the *matching update* operation for every *update-receive* operation. Likewise in the case of *state based replicated data types*, we should be able to unambiguously determine the *matching merge-send* operation for every *merge-receive* operation. We model the behaviour of the replicated data type an *abstract run*.

Definition 5 (Abstract Run)**.** *An* Abstract Run *of a replicated data type* $\mathcal{D} = (\mathsf{Univ}, \mathsf{Queries}, \mathsf{Updates}, \mathsf{Rets})$ *is a pair* (ρ, φ) *where*

- ρ *is a sequence* $Io_1o_2 \ldots o_n$ *of operations from* $\Sigma(\mathcal{D})$ *such that*
  - I *is an initialization operation which is uniformly applied to all the replicas in* $\mathcal{R}$*.*
  - $\forall i \leq |\rho|$ *if* o_i *is an* update-send *operation at a replica* r*, then the previous operation* o_{i-1} *has to be the corresponding update operation at that replica* r*. Thus,*

$$Op(o_i) = \textbf{usend} \implies Op(o_{i-1}) \in \mathsf{Updates} \wedge Rep(o_i) = Rep(o_{i-1})$$

- φ *is a partial function from* $[1..n]$ *to* $[1..n]$ *such that*
  - *The domain of* φ *is the set of* update-receive *operations and the set of* merge-receive *operations*

$$dom(\varphi) = \{i \leq n \mid o_i \textit{ is a } \textbf{ureceive} \textit{ or an } \textbf{mreceive} \textit{ operation}\}$$

  - *For an* update-receive *operation* o_i*,* $o_{\varphi(i)}$ *is the matching update operation at a remote replica.*

$$\varphi(i) = j \wedge Op(o_i) = \textbf{ureceive} \implies j < i \wedge Op(o_j) \in \mathsf{Updates} \wedge Rep(o_i) \neq Rep(o_j)$$

  - *Any pair of distinct* update-receive *operations* o_i, o_j *at the same source replica have distinct matching* update *operations.*

$$Op(o_i) = Op(o_j) = \textbf{ureceive} \wedge Rep(o_i) = Rep(o_j) \wedge i \neq j \implies \varphi(i) \neq \varphi(j)$$

  - *For a* merge-receive *operation* o_i*,* $o_{\varphi(i)}$ *is the matching* merge-send *operation at a remote replica.*

$$\varphi(i) = j \wedge Op(o_i) = \textbf{mreceive} \implies j < i \wedge Op(o_j) = \textbf{msend} \wedge Rep(o_i) \neq Rep(o_j)$$

  - *Any pair of distinct* merge-receive *operations* o_i, o_j *at the same source replica have distinct matching* merge-send *operations.*

$$Op(o_i) = Op(o_j) = \textbf{mreceive} \wedge Rep(o_i) = Rep(o_j) \wedge i \neq j \implies \varphi(i) \neq \varphi(j)$$

For a sequence $\rho = Io_1o_2 \cdots o_n$, we denote by $\rho[0]$ the initialization operation I, by $\rho[i]$ the operation o_i, and we denote by $\rho[i:j]$ the subsequence $o_i o_{i+1} \cdots o_j$.

For a run $\alpha = (\rho, \varphi)$ we say that another run $\alpha' = (\rho', \varphi')$ is a prefix of α, if $\rho' = \rho[0:n]$ for some integer n and for all $1 \leq i \leq n$, $\varphi'(i)$ is defined iff $\varphi(i)$ is defined such that $\varphi'(i) = \varphi(i)$. We say that $\alpha' \in$ Prefixes(α).

We denote the k^{th} operation at replica r in the run (ρ, φ) by $\rho_r[k]$.

The associated update *of an* update-receive *operation $\rho[i]$ is its* matching update *$\rho[\varphi(i)]$.*

The associated update *of an* update *operation $\rho[i]$ is $\rho[i]$ itself.*

The set of all operations in the run $\alpha = (\rho, \varphi)$ is denoted by $\Sigma((\rho, \varphi))$.

We denote the set of all runs of $\mathcal{D}$ by Runs$(\mathcal{D})$.

Given an abstract run (ρ, φ), ρ is a global sequence of operations across all the replicas of the data type. φ is a function that associates every *update-receive* operation, denoted by **ureceive**, with the matching *update* operation, and associates every *merge-receive* operation, denoted by **mreceive**, with its matching *merge-send* operation denoted by **msend**.

In order to reason about the states of multiple replicas at any given point in time, it is useful to consider a *snapshot* of the states of all the replicas of an implementation of a replicated data type. This snapshot is formally defined to be a *configuration* of the implementation.

Definition 6 (Configuration of an Implementation)**.** *Let $\mathcal{S}$ be the set of states of an implementation $\mathcal{D}_I$ a replicated data type on a system with $\mathcal{R} = [1, \ldots, N]$ replicas.*

A configuration of this implementation is a subset of $\underbrace{\mathcal{S} \times \mathcal{S} \times \cdots \times \mathcal{S}}_{N \text{ times}}$. The set of all configurations is denoted by $\mathcal{C}$. If $C \in \mathcal{C}$ is a configuration then the state of the replica $r \in$ Reps in C is denoted by $C[r]$.

Note that the set of abstract runs of a replicated data type is the set of all well-formed sequences of operations, which model the behaviours of all possible implementation of that data type. However a given implementation may only exhibit a subset of those behaviours. When reasoning about the correctness of an implementation with respect to a given specification, we would need to show that the subset of behaviours exhibited by the implementation are allowed by the specification. Towards this, we formally define the criteria for determining if a behaviour is exhibited by an implementation.

Definition 7 (Run of a replicated data type implementation)**.** *A run $\alpha = (\rho, \varphi)$ where $\rho = Io_1 \cdots o_n$ is accepted by a state-based implementation $\mathcal{D}_I = (\mathcal{S}, S_\perp, F_{\mathsf{Queries}}, F_{\mathsf{Updates}}, f_{\mathcal{M}})$ (resp. by a operation-based implementation $\mathcal{D}_I = (\mathcal{S}, S_\perp, \Gamma, F_{\mathsf{Queries}}, F^{op}_{\mathsf{Updates}})$ if there exists a sequence of configurations $C_0 C_1 \cdots C_n$ such that*

- $C_0 = (\underbrace{S_\perp, S_\perp, \ldots, S_\perp}_{N \text{ times}})$

- *For every* $1 \leq i \leq n$*, with* $Rep(\rho[i]) = r$*,* $Args(\rho[i]) = args$*,* $Ret(\rho[i]) = \text{ret}$*,* $C_i[r]$ *satisfies the following:*

  Case $Op(\rho[i]) =$ **msend (resp.** $Op(\rho[i]) =$ **usend)** *:*

  $$C_i[r] = C_{i-1}[r]$$

  The merge-send *(resp.* update-send*) operation should not change the state of its source replica.*

  Case $Op(\rho[i]) = q \in \mathsf{Queries}$ *:*

  $$C_i[r] = C_{i-1}[r] \text{ and } \text{ret} = f_q(args, C_{i-1}[r])$$

  The query operation should not change the state at the source replica. However the value returned by the query operation in the run should be the same as the value computed by the corresponding query function f_q*.*

  Case $Op(\rho[i]) = u \in \mathsf{Updates}$ *:*

  $$C_i[r] = f_u(args, C_{i-1}[r])$$

  (resp. in the case of operation-based implementation

  $$(C_i[r], aux) = f_u(args, C_{i-1}[r])$$

).

  The update operation should change state at the source replica as computed by the corresponding update function f_u*.*

  Case $Op(\rho[i]) =$ **mreceive** *: Let* $j = \varphi(i)$ *and let* $Op(\rho[j]) =$ **msend** *be the corresponding* message-send *operation. Let* $r' = Rep(\rho[j])$ *be the source replica of the* message-send *operation. Then,*

  $$C_i[r] = f_{\mathcal{M}}(aux, C_{i-1}[r])$$

  . Thus, the merge-receive *operation* $\rho[i]$ *should update the state at replica* r *to be the merge of the states* $C_j[r']$ *and* $C_{i-1}[r]$ *as computed by the function* $f_{\mathcal{M}}$*.*

  Case $Op(\rho[i]) =$ **ureceive** *: Let* $j = \varphi(i)$ *and let* $Op(\rho[j]) = u \in \mathsf{Updates}$ *be the corresponding update operation. Let* $args' = Args(\rho[j])$ *be the arguments of that update operation. Let* $r' = Rep(\rho[j])$ *be the source replica of that update operation. Let* aux *be the auxiliary information generated by* f_u *when applied to the state* $C_{j-1}[r']$*, i.e* $f_u(args', C_{j-1}[r']) = (C_j[r'], aux)$*. Then*

  $$C_i[r] = f_u^{\text{rcv}}(aux, C_{i-1}[r])$$

  Thus, the update-receive *operation* $\rho[i]$ *should update the state at replica* r *as per the function* f_u^{rcv} *when applied to the state* $C_{i-1}[r]$ *with the argument* aux*.*

Finally for $r' \in \mathcal{R} \wedge r' \neq r$], it is the case that $C_i[r'] = C_{i-1}[r']$, i.e. the states of the replicas should not change when the operations are performed at some remote replica.

We define the following terminology associated with runs of implementations.

- *We denote the set of all runs of $\mathcal{D}_I$ by $Runs(\mathcal{D}_I)$.*
- *If $\alpha = (\rho, \varphi) \in Runs(\mathcal{D}_I)$ then, the state of replica r at the end of the prefix $\rho[0:i]$ is denoted by $S_r(\alpha, i)$.*
- *We say that an operation o* extends *the run $\alpha = (\rho, \varphi)$ if there exists a φ' such that $\varphi'|_\rho = \varphi$ and $\alpha' = (\rho.o, \varphi')$ is a valid run of $\mathcal{D}_I$.*
- *We say that a valid run (ρ', φ') is an* extension *of (ρ, φ) iff (ρ, φ) is a prefix of (ρ', φ').*
- *In the run $\alpha = (\rho, \varphi)$, we shall denote by ρ_r to be the maximal subsequence of operations from ρ whose source replica is $r \in \mathcal{R}$.*
- *We shall use the notation $\rho_r|_{\mathsf{Updates} \cup \{\mathbf{ureceive}\}}$ to refer to the maximal subsequence of operations in ρ_r that are* updates *or* update-receive *operations. Note that these are the only two classes of operations that can modify the state of the replica.*
- *Suppose $\rho[i]$ is an* update *operation at replica r with arguments $args$ with $Op(\rho[i]) = u$. Let aux be the auxiliary information generated when f_u is applied to $S_r(\alpha, i-1)$, i.e $f_u(args, S_r(\alpha, i-1)) = (S_r(\alpha, i), aux)$. Then, we shall use the notation $Aux(\rho[i])$ to denote aux.*
- *Suppose $\rho[j]$ is an* update-receive *operation such that $\varphi(j) = i$. Then we write $Aux(\rho[j])$ to mean $Aux(\rho[i])$ since the auxiliary information that is received by the* update-receive *operation is the same as the auxiliary information generated by its corresponding* update *operation.*

In any replicated data type, it is the *update* request from the client which initiates a change in the local state. The *update* could further issue broadcasts via *update-send* or could send the updated-state itself via *merge-send* in order to trigger change in the state of a remote replica. Since we are studying replicated data types which are supposed to satisfy *strong eventual consistency*, we are interested in what set of *update* operations have impacted the state of any replica either directly via an *update* request from a client, or indirectly through *update-receive* or *merge-receive* any point in the run. We shall formally model this as the *causal past of a replica*. We shall use this in the next section to reason about the correctness of replicated data types that satisfy *strong eventual consistency*.

Definition 8 (Causal Past of a replica)**.** *Let $\alpha = (\rho, \varphi)$ be a run of a replicated data-type $\mathcal{D}$.*

The causal past of a replica r is a function $\mathsf{Past}^r_\alpha : \text{Prefixes}(\alpha) \to 2^{\Sigma(\alpha)|_{\mathsf{Updates}}}$ which associates with every prefix of the run, the set of all the update operations that are visible to r in that prefix. We define it inductively as follows.

$$\bullet\ \mathsf{Past}^r_\alpha(\rho[0:i]) = \begin{cases} \{\rho[0]\} & \text{if } i = 0 \\ \mathsf{Past}^r_\alpha(\rho[0:i-1]) & \text{if } Op(\rho[i]) \in \mathsf{Queries} \cup \{\mathbf{msend}, \mathbf{usend}\} \\ & \quad \vee\ Rep(\rho[i]) \neq r \\ \mathsf{Past}^r_\alpha(\rho[0:i-1]) \cup \{\rho[i]\} & \text{if } Op(\rho[i]) \in \mathsf{Updates} \wedge Rep(\rho[i]) = r \\ \mathsf{Past}^r_\alpha(\rho[0:i-1]) \cup \{\rho[j]\} & \text{if } Op(\rho[i]) = \mathbf{ureceive} \\ & \quad \wedge\ Rep(\rho[i]) = r \wedge \varphi(i) = j \\ \mathsf{Past}^r_\alpha(\rho[0:i-1]) \cup \mathsf{Past}^{r'}_\alpha(\rho[0:j])) & \text{if } Op(\rho[i]) = \mathbf{mreceive} \\ & \quad \wedge\ Rep(\rho[i]) = r \\ & \quad \wedge\ \varphi(i) = j \\ & \quad \wedge\ Rep(\rho[\varphi(i)]) = r' \end{cases}$$

We denote by $\mathsf{Past}_\alpha(\rho[i])$ the causal past $\mathsf{Past}^{Rep(\rho[i])}_\alpha(\rho[0:i])$.

Thus, the *causal past* of a replica in a prefix of a run will not change when the latest operation in the prefix is a *query* or a *update-send* or a *merge-send* or any operation that is performed on some other replica. If the latest operation in the prefix of the run is an *update* operation performed at the replica, then the *causal past* of replica includes that *update* operation. If the latest operation in the prefix of the run is an *update-receive* operation, then the *matching update* operation will be included into *causal past* of the replica. Finally, if the latest operation in the prefix of the run is a *merge-receive* operation, then the causal past of the replica will be extended with the causal past of the remote replica when that remote replica performed the *matching merge-send*.

The notion of the causal past helps us formally define when an *update* operation gets *delivered* at another replica in a *run*. This helps us reason about the delivery of updates across both *state-based* and *operation-based* implementations in a uniform manner, even though *update* operations are not explicitly propagated to the other replicas in a *state-based* implementation.

Definition 9 (Delivery of an update)**.** *An update operation $\rho[i]$ is said to be* delivered *at a replica $r \neq Rep(\rho[i])$ in the run $\alpha = (\rho, \varphi)$, if there exists a j such that $Rep(\rho[j]) = r$ and $\rho[i] \in \mathsf{Past}_\alpha(\rho[j])$.*

An update operation $\rho[i]$ is said to be delivered *in the run (ρ, φ) if it is delivered at every replica $r \neq Rep(\rho[i])$.*

A run $\alpha = (\rho, \varphi)$ is said to be a complete run *iff all the updates in the run are delivered.*

We say that a complete run (ρ', φ') is a completion *of (ρ, φ) iff (ρ', φ') extends (ρ, φ) and for all $i > |\rho| : Op(\rho'[i]) \in \mathsf{UComm} \cup \mathsf{MComm}$.*

Given a pair of *update operations* $\rho[i], \rho[j]$ in a *run* (ρ, φ), there are only three relations possible between them. It may be the case that $\rho[i]$ has *happened-before* $\rho[j]$, in which case, the source replica of the $\rho[j]$ would already be aware of $\rho[i]$. Or it may be the case that $\rho[j]$ has *happened-before* $\rho[i]$. If neither of the source replicas of the two updates is aware of the other *update* operation at the time when it is performing its *update* operation locally, then the two *update* operations are said to be *concurrent* in the run. Formally we can define this as follows.

Definition 10 (Concurrency and Happened-Before)**.** *Let (ρ, φ) be a run of a replicated data type. Let $\rho[i], \rho[j]$ be a pair of update operations.*

Then we say that $\rho[i]$ has happened before $\rho[j]$ (denoted by $\rho[i] \xrightarrow{\mathsf{hb}} \rho[j]$) if $\rho[i] \in \mathsf{Past}_{(\rho,\varphi)}(\rho[j])$. They are said to be concurrent *if neither $\rho[i]$ happened-before $\rho[j]$ nor vice-versa. We denote this by $\rho[i] \parallel \rho[j]$.*

Certain replicated data types require that any pair of updates which are related by the *happened-before* relation be delivered to all the replicas in that same order. This is known as the *causal delivery* of the updates. Formally,

Definition 11 (Causal Delivery of an updates)**.** *We say that updates are* causally delivered *in a run $\alpha = (\rho, \varphi)$, iff for every $i, j : 0 < i \leq j \leq |\rho|$, if $\rho[i]$ is an update operation such that $\rho[i] \in \mathsf{Past}_\alpha(\rho[j])$, then, $\mathsf{Past}_\alpha(\rho[i]) \subseteq \mathsf{Past}_\alpha(\rho[j])$.*

Note that it is possible that two distinct states are indistinguishable for an end user because they always respond to queries in a similar manner. We shall define this property to be the *query equivalence* of those states.

Definition 12 (Query Equivalence)**.** *A pair of states $S, S' \in \mathcal{S}$ of an implementation*

$$\mathcal{D}_I = (\mathcal{S}, S_\perp, F_{\mathsf{Queries}}, F_{\mathsf{Updates}}, f_{\mathcal{M}})$$

(resp. by a operation-based implementation $\mathcal{D}_I = (\mathcal{S}, S_\perp, \Gamma, F_{\mathsf{Queries}}, F^{op}_{\mathsf{Updates}})$) are said to be query-equivalent*, or simply* equivalent*, they return the same return-value for the same query operations with the same arguments. For every query operation $q \in \mathsf{Queries}$ and every valid argument $args \in \mathsf{Univ}^*$, $f_q(args, S) = f_q(args, S')$.*

If S and S' are query equivalent, we write it as $S \cong S'$.

We formally define the consistency criteria known as *strong eventual consistency* which requires that replicas which have received the same set updates are query equivalent.

Definition 13 (Strong Eventual Consistency). *An implementation $\mathcal{D}_I$ of a replicated data type $\mathcal{D}$ is said to satisfy* Strong Eventual Consistency *if for every run $(\rho, \varphi) \in Runs(\mathcal{D}_I)$, for any pair of operations $\rho[i]$, $\rho[j]$, with $Rep(\rho[i]) = r_i$ and $Rep(\rho[j]) = r_j$, it is the case that, $\mathsf{Past}_{(\rho,\varphi)}(\rho[i]) = \mathsf{Past}_{(\rho,\varphi)}(\rho[j]) \implies S_{r_i}(\rho, \varphi, i) \cong S_{r_j}(\rho, \varphi, j)$*

2.2 Convergent and Commutative Replicated Data Types

In their work [Shapiro et al., 2011b], Shapiro et al. define *Convergent Replicated Data Types (CvRDTs)* to be the *State-based* implementations of replicated data types that satisfy *strong eventual consistency* and as [Shapiro et al., 2011b] and *Commutative Replicated Data Types (CmRDTs)* to be the *Operation-based* implementations of replicated data types that satisfy *strong eventual consistency*. They also provide sufficient conditions for a state based or an operation based replicated data type to satisfy strong eventual consistency. The gist is that an operation based replicated data type satisfies strong eventual consistency if the underlying network guarantees causal delivery of updates and if the concurrent update commute, i.e it does not matter in which order the replicas receive the broadcast messages corresponding to concurrent updates since the resultant state will be equivalent no matter in what order they are applied. On the other hand, state-based replicated data types satisfy strong eventual consistency if i) the set of states along with a partial order defined on the set forms a join-semilattice, and ii) as a result of every update operation, the state move up in the join-semilattice and the result of a merge function applied on a pair of states produces the least upper bound of those two states in the join-semilattice. In this section we formalize these sufficient conditions and provide a rigorous proof for these sufficient conditions pertaining to CvRDTs and CmRDTs.

Consider an *operation-based* implementation $\mathcal{D}_I = (\mathcal{S}, S_\perp, \Gamma, F_{\mathsf{Queries}}, F_{\mathsf{Updates}})$. Let $u \in \mathsf{Updates}$ be an *update* method and (f_u, f_u^{rcv}) be the functions implementing the *update* method and its *update-receive* methods. Let $args \in \mathsf{Univ}^*$ be the argument to u. Let $S, S_u \in \mathcal{S}$ be a pair of states of the implementation and let $aux \in \Gamma$ be an auxiliary information such that $f_u(args, S) = (S_u, aux)$. Then we write $S.f_u(args)$ to denote the state S_u. Furthermore, for a pair of states state $S', S'' \in \mathcal{S}$, if $f_u^{rcv}(aux, S') = S''$, then we write $S'.f_u^{rcv}(aux)$ to refer to the state S'' which is obtained as a result of $f_u^{rcv}(aux, S')$.

Note that the sufficient condition for CmRDTs requires the concurrent updates to commute. So we first formally define commutative updates as follows.

Definition 14 (Commutative Updates). *Let $u, u' \in \mathsf{Updates}$ be a pair of update methods of a replicated data type $\mathcal{D} = (\mathsf{Univ}, \mathsf{Queries}, \mathsf{Updates}, \mathsf{Rets})$.*

Let $\mathcal{D}_I = (\mathcal{S}, S_\perp, \Gamma, F_{\mathsf{Queries}}, F^{op}_{\mathsf{Updates}})$ be an operation-based implementation of $\mathcal{D}$ with (f_u, f_u^{rcv}) and $(f_{u'}, f_{u'}^{\text{rcv}})$ being the functions implementing the update *and* update-receive *methods of u and u' respectively. Let $S, S', S'' \in \mathcal{S}$ be states of a replica r. Let $args, args' \in$*

Univ^* *respectively be valid arguments to* f_u *and* $f_{u'}$ *respectively. Let* aux, aux' *respectively be the auxiliary information generated by applying* $f_u(args)$ *at* $S \in \mathcal{S}$ *and* $f_{u'}(args')$ *at* $S' \in \mathcal{S}$. *Then we say that* u *and* u' *are* commutative updates *in* $\mathcal{D}_I$ *iff*

$$S''.f_{u'}(args').f_u^{\text{rcv}}(aux) = S''.f_u^{\text{rcv}}(aux).f_{u'}^{\text{rcv}}(aux')$$

Thus, when a pair of commutative updates are applied to replicas with the same state, the resultant state remains the same, irrespective of the order in which the information pertaining to the updates were delivered.

For an operation based implementation, the state of a replica at the end of a run is obtained by applying the sequence of corresponding *update* and the *update-receive* methods from the run to the initial state $S_\perp$. Due to the property that an *update* method applied to a state S produces the same result state as when the corresponding *update-receive* method applied at the state S, it is possible to represent the state of a replica at the end of a run as a sequence of *update-receive* methods applied to the starting state. This property helps us reason about the evolution of the state of a replica in a run using only the *update-receive* methods, which simplifies some of the technical proofs later on. We prove this result in the proposition below.

Proposition 15. *Let* $\mathcal{D}_I = (\mathcal{S}, S_\perp, \Gamma, F_{\mathsf{Queries}}, F^{op}_{\mathsf{Updates}})$ *be an* operation-based *implementation of a replicated data type* $\mathcal{D} = (\mathsf{Univ}, \mathsf{Queries}, \mathsf{Updates}, \mathsf{Rets})$. *Let* $\alpha = (\rho, \varphi)$ *be a run of* $\mathcal{D}_I$.

Let the update method *associated with an update operation* $\rho[i]$ *be* $Op(\rho[i])$ *and the update method associated with an* update-receive *operation* $\rho[j]$ *be* $Op(\rho[\varphi(j)])$.

Suppose $r \in \mathcal{R}$ *be a replica. Let* ρ' *be the maximal sequence of just the update and the update receive operations from* ρ *whose source replica is* r. *Let* $|\rho'| = n$. *For* $i \in [1, \ldots, n]$, *let the update operation associated with* $\rho'[i]$ *be* u_i *and let* $Aux(\rho_r[i])$ *be* aux_i. *Then,*

$$S_r(\alpha) = S_\perp.f^{rcv}_{u_1}(aux_1).f^{rcv}_{u_2}(aux_2).\ldots.f^{rcv}_{u_n}(aux_n)$$

Proof. Since the only operations which modify the states of a replica are the *update* and the *update-receive* operations, the state of a replica r at the end of the run $\alpha = (\rho, \varphi)$ is exactly the state of the replica after applying the sequence operations in ρ' to the initial state. Let the state of r be S^i_r at the end of $\rho'[i]$. Then $S_r(\alpha) = S^n_r$.

We know that,

$$S^{i+1}_r = \begin{cases} S^i_r.f_{u_i}(args_i) & \text{if } \; Op(\rho_r[i]) = u_i \wedge Args(\rho_r[i]) = args_i \\ S^i_r.f^{rcv}_{u_i}(aux_i) & \text{if } \; Op(\rho_r[i]) = \textbf{ureceive} \wedge \\ & \qquad \text{the associated update is } u_i \wedge Aux(\rho_r[i]) = aux_i \end{cases}$$

By definition of the *operation-based* replicated data type, for any state S, $S.f_{u_i}(args_i) = S.f^{rcv}_{u_i}(aux_i)$.

Thus, if $Op(\rho'[i])$ is an *update* operation, $S^i_r.f_{u_i}(args_i) = S^i_r.f^{rcv}_{u_i}(aux_i)$. Thus, we can unconditionally write $S^{i+1}_r = S^i_r.f^{rcv}_{u_i}(aux_i)$.

Since $S^0_r = S_\perp$, we can see that the state of r at the end of ρ' is

$$S^n_r = S_\perp.f^{rcv}_{u_1}(aux_1).f^{rcv}_{u_2}(aux_2).\ldots.f^{rcv}_{u_n}(aux_n)$$

.

From this, and the fact that $S_r(\alpha) = S^n_r$, the result follows. □

We shall now prove a result pertaining to the commutativity of updates.

Proposition 16. *Let $\alpha = (\rho, \varphi)$ be a run of an operation-based implementation*

$$\mathcal{D}_I = (\mathcal{S}, S_\perp, \Gamma, F_{\mathsf{Queries}}, F^{op}_{\mathsf{Updates}})$$

. Let $r \in \mathcal{R}$ be a replica. Let ρ' be the maximal sequence of update *and* update-receive *operations from ρ whose source replica is r. Let*

- *S^j_r be the state of r at the end of $\rho'[j]$.*
- *u_j be the update method of the update operation associated with the operation $\rho'[j]$. (Note: $\rho'[j]$ could be an* update *or an* update-receive *operation and u_j is the associated* update method*).*
- *$x_j = Aux(\rho'[j])$.*

Suppose there exists some i and m, such that for all $k \in [1, \ldots, m]$, $\rho'[i]$ commutes with $\rho'[i+k]$. Then, for all $k \in [0, \ldots, m]$,

$$S^{i+m}_r = S^{i-1}_r.f^{rcv}_{u_{i+1}}(x_{i+1}).f^{rcv}_{u_{i+2}}(x_{i+2}).\ldots.f^{rcv}_{u_{i+k}}(x_{i+k}).f^{rcv}_{u_i}(x_i).f^{rcv}_{u_{i+k+1}}(x_{i+k+1}).\ldots.f^{rcv}_{u_{i+m}}(x_{i+m})$$

Proof. We shall prove this by induction over k. The case with $k = 0$, is trivially proved since the RHS is

$$S^{i-1}_r.f^{rcv}_{u_i}(x_i).f^{rcv}_{u_{i+1}}(x_{i+1}).\ldots.f^{rcv}_{u_{i+m}}(x_{i+m})$$

which is the same as S^{i+m}_r by definition.

We assume that the result holds for $k-1$

Let $S = S^{i-1}_r.f^{rcv}_{u_{i+1}}(x_{i+1}).f^{rcv}_{u_{i+2}}(x_{i+2}).\ldots.f^{rcv}_{u_{i+k-1}}(x_{i+k-1})$.

Then since $\rho'[i]$ commutes with $\rho'[i+k]$,

$$S.f^{rcv}_{u_{i+k}}(x_{i+k}).f^{rcv}_{u_i}(x_i) = S.f^{rcv}_{u_i}(x_i).f^{rcv}_{u_{i+k}}(x_{i+k})$$

Thus,

$$\begin{aligned}
&S_r^{i-1}.f^{rcv}_{u_{i+1}}(x_{i+1}).\ldots.f^{rcv}_{u_{i+k-1}}(x_{i+k-1}).f^{rcv}_{u_{i+k}}(x_{i+k}).f^{rcv}_{u_i}(x_i).f^{rcv}_{u_{i+k+1}}(x_{i+k+1}).\ldots.f^{rcv}_{u_{i+m}}(x_{i+m})\\
&\quad = S.f^{rcv}_{u_{i+k}}(x_{i+k}).f^{rcv}_{u_i}(x_i).f^{rcv}_{u_{i+k+1}}(x_{i+k+1}).\ldots.f^{rcv}_{u_{i+m}}(x_{i+m})\\
&\quad = S.f^{rcv}_{u_i}(x_i).f^{rcv}_{u_{i+k}}(x_{i+k}).f^{rcv}_{u_{i+k+1}}(x_{i+k+1}).\ldots.f^{rcv}_{u_{i+m}}(x_{i+m})\\
&\quad = S_r^{i+m} \quad \text{(by induction hypothesis)}
\end{aligned}$$

Thus the results holds for k if it holds for $k-1$. By the principle of mathematical induction, the results holds for all k. □

We next show that the final state of a replica in an *operation-based* implementation remains unchanged when the commutative updates are reordered with respect to each other.

Lemma 17. *Let $\alpha = (\rho, \varphi)$ be a run of an operation-based implementation*

$$\mathcal{D}_I = (\mathcal{S}, S_\perp, \Gamma, F_{\mathsf{Queries}}, F^{op}_{\mathsf{Updates}})$$

where the updates are causally delivered. Let $r \in \mathcal{R}$ be a replica. Let ρ' be the maximal sequence of update *and* update receive *operations from ρ whose source replica is r. Let*

- $|\rho'| = n$
- *u_j be the update method of the update operation associated with $\rho'[j]$*
- *$x_j = Aux(\rho'[j])$.*

We say that a permutation $\pi : [1, \ldots, n] \to [1, \ldots, n]$ is a causality-preserving permutation *if for any pair of integers $i, j \in [1, \ldots, n]$, such that the updates associated with $\rho'[i]$ and $\rho'[j]$ are not concurrent, then $i < j$ iff $\pi(i) < \pi(j)$.*

Suppose all the concurrent updates of the replicated data type commute, then the state of replica r, at the end of the run α is

$$S_r(\alpha) = S_\perp.f^{rcv}_{u_{\pi(1)}}(x_{\pi(1)}).f^{rcv}_{u_{\pi(2)}}(x_{\pi(2)}).\ldots.f^{rcv}_{u_{\pi(n)}}(x_{\pi(n)})$$

where $\pi : [1, \ldots, n] \to [1, \ldots, n]$ is a causality-preserving permutation.

Proof. We need to prove that

$$S_\perp.f^{rcv}_{u_{\pi(1)}}(x_{\pi(1)}).f^{rcv}_{u_{\pi(2)}}(x_{\pi(2)}).\ldots.f^{rcv}_{u_{\pi(n)}}(x_{\pi(n)}) = S_\perp.f^{rcv}_{u_1}(x_1).f^{rcv}_{u_2}(x_2).\ldots.f^{rcv}_{u_n}(x_n)$$

We shall prove this by induction over $|\rho'|$. If $|\rho'| = 0$, then the result is trivially true as both LHS and RHS are $S_\perp$.

Assume that the result holds for all the cases when $|\rho'| < n$. We shall now prove the result when $|\rho'| = n$.

Suppose $\pi(n) = n$. Let $\pi' : [1, \ldots, n-1] \to [1, \ldots, n-1]$ be a permutation such that $\pi'(i) = \pi(i)$. Thus π' is a valid order-preserving permutation over $\rho'[1 : n-1]$. By induction hypothesis,

$$S_\perp.f^{rcv}_{u_{\pi'(1)}}(x_{\pi'(1)}).f^{rcv}_{u_{\pi'(2)}}(x_{\pi'(2)}).\ldots.f^{rcv}_{u_{\pi'(n-1)}}(x_{\pi'(n-1)}) = S_\perp.f^{rcv}_{u_1}(x_1).f^{rcv}_{u_2}(x_2).\ldots.f^{rcv}_{u_{n-1}}(x_{n-1})$$

Thus,

$$\begin{aligned}
&S_\perp.f^{rcv}_{u_{\pi(1)}}(x_{\pi(1)}).f^{rcv}_{u_{\pi(2)}}(x_{\pi(2)}).\ldots.f^{rcv}_{u_{\pi(n)}}(x_{\pi(n)}) \\
&\quad = S_\perp.f^{rcv}_{u_{\pi'(1)}}(x_{\pi'(1)}).f^{rcv}_{u_{\pi'(2)}}(x_{\pi'(2)}).\ldots.f^{rcv}_{u_{\pi'(n-1)}}(x_{\pi'(n-1)}).f^{rcv}_{u_{\pi(n)}}(x_{\pi(n)}) \\
&\quad = S_\perp.f^{rcv}_{u_1}(x_1).f^{rcv}_{u_2}(x_2).\ldots.f^{rcv}_{u_{n-1}}(x_{n-1}).f^{rcv}_{u_{\pi(n)}}(x_{\pi(n)}) \\
&\quad = S_\perp.f^{rcv}_{u_1}(x_1).f^{rcv}_{u_2}(x_2).\ldots.f^{rcv}_{u_{n-1}}(x_{n-1}).f^{rcv}_{u_n}(x_n)
\end{aligned}$$

Hence the result holds for $|\rho'| = n$ when $\pi(n) = n$.

Suppose $\pi(n) \neq n$. Then, let $\pi(n) = k$ and k' be such that $\pi(k') = n$. Note that since π is a causality-preserving permutation, it is the case that $\rho[n]$ and $\rho[k]$ are concurrent. Otherwise, since $k < n$ and $\pi(n) = k$ and $\pi(k') = n$, it would require $n < k'$ which is a contradiction.

Furthermore, it is the case that for any $j : k < j \leq n$, $\rho'[k]$ and $\rho'[j]$ are concurrent. To see why this is true, assume that they are not concurrent. Then, since $k < j$, it has to be the case that $\rho'[k]$ happened-before $\rho'[j]$. Let $j' \in [1..n]$ be an integer such that $\pi(j') = j$. Now, it is clear that $j' \leq n$. But then, it is easy to see that j' is distinct from n. Because if $j' = n$, then, $\pi(j') = \pi(n)$. But then $\pi(j') = j$ and $\pi(n) = k$ and j and k are distinct. Thus, j' and n are distinct. Thus, it is clear that $j' < n$. We are assuming that $\rho'[k]$ happened before $\rho'[j]$. That is the same as saying that $\rho'[\pi(n)]$ happened before $\rho'[\pi(j')]$. Since π is an causality-preserving permutation it has to be the case that $n < j'$. But this contradicts our earlier observation that $j' < n$. Hence, our original assumption that $\rho'[k]$ and $\rho'[j]$ are not concurrent is incorrect. Which proves the fact that for any $j : k < j \leq n$, $\rho'[k]$ and $\rho'[j]$ are concurrent.

Hence, by proposition 16, we can shuffle $f^{rcv}_{u_k}(x_k)$ towards the end, closer to $f^{rcv}_{u_n}(x_n)$. Thus,

$$\begin{aligned}
&S_\perp.f^{rcv}_{u_1}(x_1).\ldots.f^{rcv}_{u_n}(x_n) \\
&\quad = S_\perp.f^{rcv}_{u_1}(x_1).\ldots.f^{rcv}_{u_{k-1}}(x_{k-1}).f^{rcv}_{u_{k+1}}(x_{k+1}).\ldots.f^{rcv}_{u_n}(x_n).f^{rcv}_{u_k}(x_k)
\end{aligned}$$

By similar reasoning as above, for any $j' : k' < j' \leq n$, it is the case that $\rho'[\pi(k')]$ and $\rho'[\pi(j')]$ are concurrent. Otherwise, since π is a causality-preserving permutation, and

$k' < j'$, it is the case that $\pi(k') < \pi(j')$. But that is not possible since $\pi(k') = n$ and $\pi(j') = j$ for some $j < n$. Thus, $\pi(k') < \pi(j') \implies n < j$ which is a contradiction. Hence for all $j' : k' < j' \leq n$, we it is the case that $\rho'[\pi(k')]$ and $\rho'[\pi(j')]$ are concurrent. Once again, by proposition 16, we have

$$S_{\perp}.f^{rcv}_{u_{\pi(1)}}(x_{\pi(1)}).\ldots.f^{rcv}_{u_{\pi(n)}}(x_{\pi(n)})$$
$$= S_{\perp}.f^{rcv}_{u_{\pi(1)}}(x_{\pi(1)}).\ldots.f^{rcv}_{u_{\pi(k'-1)}}(x_{\pi(k'-1)}).f^{rcv}_{u_{\pi(k'+1)}}(x_{\pi(k'+1)}).\ldots.f^{rcv}_{u_{\pi(k')}}(x_{\pi(k')}).f^{rcv}_{u_{\pi(n)}}(x_{\pi(n)})$$

Since

$$f^{rcv}_{u_n}(x_n) = f^{rcv}_{u_{\pi(k')}}(x_{\pi(k')})$$

and

$$f^{rcv}_{u_k}(x_k) = f^{rcv}_{u_{\pi(n)}}(x_{\pi(n)})$$

in order to prove the induction result for k, it is sufficient to show that

$$S_{\perp}.f^{rcv}_{u_1}(x_1).\ldots.f^{rcv}_{u_{k-1}}(x_{k-1}).f^{rcv}_{u_{k+1}}(x_{k+1}).\ldots.f^{rcv}_{u_{n-1}}(x_{n-1})$$
$$= S_{\perp}.f^{rcv}_{u_{\pi(1)}}(x_{\pi(1)}).\ldots.f^{rcv}_{u_{\pi(k'-1)}}(x_{\pi(k'-1)}).f^{rcv}_{u_{\pi(k'+1)}}(x_{\pi(k'+1)}).\ldots.f^{rcv}_{u_{\pi(n-1)}}(x_{\pi(n-1)})$$

Now suppose α'' is the run α without any *query* operations and without any *update* and the *update-receive* operations associated with $\rho'[k]$ and $\rho'[n]$. Clearly, α'' is a well-defined run. Let ρ'' is the maximal sequence of *update* and *update-receive* operations in α'' at replica r. It is clear that $\rho'' = \rho' \setminus \{\rho'[k], \rho'[n]\}$. Thus we can write

$$\rho''[i] = \rho'[g(i)]$$

where $g : [1, \ldots, n-2] \to [1, \ldots, k-1, k+1, \ldots, n-1]$ is a the bijective function defined as follows

$$g(i) = \begin{cases} i & \text{for } 1 \leq i < k \\ i+1 & \text{for } k \leq i \leq n-2 \end{cases}$$

It can be noted that for $i < j \iff g(i) < g(j)$. Let u'_i be the update method associated with $\rho''[i]$. Then, $u'_i = u_{g(i)}$.

If the auxiliary information generated by the update associated with $\rho''[j]$ is denoted by x'_j, i.e. $x'_j = Aux(\rho''[j])$, then it is the case that $x'_j = x_{g(j)} = Aux(\rho'[g(j)])$. The reasoning is as follows.

In ρ', the update associated with $\rho'[k]$ is not in the causal past of the update associated with $\rho'[j]$ for all $j : 1 \leq j \neq k < n$. This is because for $j : k < j < n$, we have shown earlier that the update associated with $\rho'[k]$ is concurrent with the update associated with $\rho'[j]$.

For $1 \leq j < k$, due to causal-delivery, it is clear that the update associated with $\rho'[j]$ either happened before the update associated with $\rho'[k]$ or is concurrent with it. Thus, $\rho'[k]$ is not in the causal-past of any other update operation in ρ'. By similar reasoning, the update operation associated with $\rho'[n]$ is not in the causal past of any other update operation in ρ'. Since α'' is the run α without the queries and without the update and update-receive operations associated with $\rho'[k]$ and $\rho'[n]$, and the fact that the update associated with $\rho''[j]$ is exactly the update associated with $\rho'[g(j)]$ it follows that set of updates visible to $\rho''[j]$ in α'' are exactly the set of updates visible to $\rho'[g(j)]$ in α and in the exact same order. Thus, it follows that thus state of the source replica of the update associated with $\rho''[j]$ prior to performing that update in the run α'' is the same as the state of that source replica prior to performing the corresponding update associated with $\rho'[g(j)]$ in α. Thus, if $x'_j = Aux(\rho''[j])$ then, $x'_i = x_{g(i)}$.

From this, have

$$\begin{aligned}
&S_\perp.f^{rcv}_{u'_1}(x'_1).\ldots.f^{rcv}_{u'_{n-2}}(x'_{n-2}) \\
&\quad = S_\perp.f^{rcv}_{u_{g(1)}}(x_{g(1)}).\ldots.f^{rcv}_{u_{g(n-2)}}(x_{g(n-2)}) \\
&\quad = S_\perp.f^{rcv}_{u_1}(x_1).\ldots.f^{rcv}_{u_{k-1}}(x_{k-1}).f^{rcv}_{u_{k+1}}(x_{k+1}).\ldots.f^{rcv}_{u_{n-1}}(x_{n-1})
\end{aligned}$$

Recall that $k' \in [1,\ldots,n]$ is the integer such that $\pi(k') = n$. We define another bijective map $h : [1,\ldots,n-2] \to [1,\ldots,k'-1,k'+1,\ldots,n-1]$ as follows:

$$h(i') = \begin{cases} i' & \text{for } 1 \leq i' < k' \\ i'+1 & \text{for } k' \leq i' \leq n-2 \end{cases}$$

Again it is the case that $i' < j' \iff h(i') < h(j')$.

Now for the operations in ρ'', we define the permutation $\pi' : [1,\ldots,n-2] \to [1,\ldots,n-2]$ such that $\pi'(i')$ corresponds to $\pi(h(i'))$ in ρ'. Formally

$$\pi'(i') = g^{-1}(\pi(h(i')))$$

Thus $g(\pi'(i')) = \pi(h(i'))$ and by construction,

$$(u'_{\pi'(i')}, x'_{\pi'(i')}) = (u_{g(\pi'(i'))}, x_{g(\pi'(i'))}) = (u_{\pi(h(i'))}, x_{\pi(h(i'))})$$

Thus,

$$S_\perp.f^{rcv}_{u'_{\pi'(1)}}(x'_{\pi'(1)}).\ldots.f^{rcv}_{u'_{\pi'(n-2)}}(x'_{\pi'(n-2)})$$

is the same as

$$S_\perp.f^{rcv}_{u_{\pi(1)}}(x_{\pi(1)}).\ldots.f^{rcv}_{u_{\pi(k'-1)}}(x_{\pi(k'-1)}).f^{rcv}_{u_{\pi(k'+1)}}(x_{\pi(k+1)}).\ldots.f^{rcv}_{u_{\pi(n-1)}}(x_{\pi(n-1)})$$

Thus, if we prove that π' is an causality-preserving permutation function, from induction hypothesis, since $|\rho''| < n$, it follows that

$S_\perp.f^{rcv}_{u_1}(x_1).\ldots.f^{rcv}_{u_{k-1}}(x_{k-1}).f^{rcv}_{u_{k+1}}(x_{k+1}).\ldots.f^{rcv}_{u_{n-1}}(x_{n-1})$ is equal to

$S_\perp.f^{rcv}_{u_{\pi(1)}}(x_{\pi(1)}).\ldots.f^{rcv}_{u_{\pi(k'-1)}}(x_{\pi(k'-1)}).f^{rcv}_{u_{\pi(k'+1)}}(x_{\pi(k+1)}).\ldots.f^{rcv}_{u_{\pi(n-1)}}(x_{\pi(n-1)})$

Suppose $\rho''[i]$ and $\rho''[j]$ are not concurrent.

Let i', j' be such that $\pi'(i') = i$ and $\pi'(j') = j$. Since, $\pi'(i') = g^{-1}(\pi(h(i'))$ we have $i' = h^{-1}(\pi^{-1}(g(\pi'(i'))))$. Similarly, $j' = h^{-1}(\pi^{-1}(g(\pi'(j'))))$.

Now

$$
\begin{aligned}
i < j &\iff \pi'(i') < \pi'(j') \quad \text{by definition} \\
&\iff g(\pi'(i')) < g(\pi'(j')) \quad \text{(by definition of } g\text{)} \\
&\iff \pi^{-1}(g(\pi'(i'))) < \pi^{-1}(g(\pi'(j'))) \quad \text{(as } \pi \text{ is causality-preserving)} \\
&\iff h^{-1}(\pi^{-1}(g(\pi'(i')))) < h^{-1}(\pi^{-1}(g(\pi'(j')))) \text{ by definition of } h \\
&\iff i' < j' \text{ by definition of } i' \text{ and } j'
\end{aligned}
$$

Thus the permutation π' is a causality-preserving. This completes the proof of this lemma. □

We now state and prove two theorems from [Shapiro et al., 2011b] which provide the sufficient conditions for a replicated data type to be a CvRDT or a CmRDT.

Theorem 18 (Sufficient condition for CmRDT [Shapiro et al., 2011b])**.** *Assuming termination and causal delivery of updates, a sufficient condition for an op-based replicated datatype to satisfy strong eventual consistency is that all the concurrent updates should commute and all delivery preconditions are compatible with causal delivery.*

Proof. Let $\alpha = (\rho, \varphi)$ be a run of an operation-based replicated data types where the updates are causally delivered and concurrent updates commute. Let $r, s \in \mathcal{R}$ be two replicas which have the same causal past at the end of α. We need to show that the states of r and s at the end of α are query-equivalent. Let $\rho'_r = \rho_r|_{\mathsf{Updates} \cup \{\mathbf{ureceive}\}}$ and $\rho'_s = \rho_s|_{\mathsf{Updates} \cup \{\mathbf{ureceive}\}}$.

Since ρ'_r and ρ'_s contain only update operations and update-receive operations, and the fact that they have the same causal past at the end of α, it follows that they both have the same number of operations. Let $|\rho'_r| = |\rho'_s| = n$. Let $\pi : [1, \ldots, n] \to [1, \ldots, n]$ be an bijective map defined as follows.

Since r and s have the same causal past,

- If $\rho'_s[i]$ is an update operation, then, the corresponding update-receive operation should appear in ρ'_r. Suppose it is $\rho'_r[j]$. We define $\pi(i) = j$.

- If $\rho'_s[i']$ is an update-receive operation whose matching update operation occurred at r, then, that update operation appears in ρ'_r. Suppose it is $\rho'_r[j']$. We define $\pi(i') = j'$.
- If $\rho'_s[i'']$ is an update-receive operation whose associated update operation occured at neither r nor s, then, there exists a corresponding update-receive operation in ρ'_r. Let $\rho'_r[j'']$ be that operation. We define $\pi(i'') = j''$.

Thus, π is a bijective map and a permutation.

Further, suppose for some i, j, the updates associated with $\rho'_r([\pi(i))$ and $\rho'_r([\pi(j))$ are not concurrent. Since the updates are delivered in causal order, every replica sees them in the order of their happened before relation. Thus, $\pi(i) < \pi(j)$ iff $i < j$. Thus π is a causality-preserving permutation.

From this, and from Proposition 15 and Lemma 17, it follows that the state of replicas r at the end of $\rho' r$ and the state of s at the end of ρ'_s are the same. Thus, their states are query-equivalent. This proves the theorem. □

Theorem 19 (Sufficient condition for CvRDT [Shapiro et al., 2011b])**.** *Let* $\mathcal{D}_I = (\mathcal{S}, S_\perp, F_{\mathsf{Queries}}, F_{\mathsf{Updates}}, f_{\mathcal{M}})$ *be a state-based implementation of a replicated data type* $\mathcal{D} = (\mathsf{Univ}, \mathsf{Queries}, \mathsf{Updates}, \mathsf{Rets})$ *over a distributed system with replicas* $\mathcal{R}$ *such that:*

- *There exists a partial order* $\leq$ *over* $\mathcal{S}$ *such that* $(\mathcal{S}, \leq)$ *is a join-semilattice with* $\sqcup$ *denoting the least-upper-bound operation.*
- *For any state* S, $u \in \mathsf{Updates}$, *args* $\in \mathsf{Univ}^*$, $S \leq S.f_u(args)$
- *For any pair of state* S, S', $f_{\mathcal{M}}(S, S') = S \sqcup S'$.

Then $\mathcal{D}_I$ *is a CvRDT.*

Proof. Let $\alpha = (\rho, \varphi)$ be a run of $\mathcal{D}_I$. For any $i \in [0, |\rho|]$ with $Rep(\rho[i]) = r$. Let the causal past of r at the end of $\rho[0 : i]$ be $\{\rho[i_1], \rho[i_2], \ldots, \rho_{i_{k'}}]$. Let S_{i_j} denote the state of the source replica of $\rho[i_j]$ at the end of the run $\rho[0 : i_j]$. Then, we shall show that the state of r at the end of $\rho[i]$ is

$$S_r(\rho, \varphi, i) = S_\perp \sqcup S_{i_1} \sqcup S_{i_2} \sqcup \ldots \sqcup S_{i_{k'}}$$

This will prove that any pair of replicas that have the same causal past in a run will be at the same state.

We prove this by induction over i. If $i = 0$, then, $\mathsf{Past}^r_{(\rho,\varphi)}(\rho[0 : 0]) = \{I\}$. Thus, $S_r(\rho, \varphi, i) = S_\perp$ which completes the proof for $i = 0$.

Suppose the result holds for all $i < k$.

Consider $\rho[k]$. If $Op(\rho[k]) \in \mathsf{Queries} \cup \{\mathbf{msend}\}$, or $Rep(\rho[k]) \neq r$, then $\mathsf{Past}^r_{(\rho,\varphi)}(\rho[0 : i]) = \mathsf{Past}^r_{(\rho,\varphi)}(\rho[0 : i-1])$ and $S_r(\rho, \varphi, i) = S_r(\rho, \varphi, i-1)$. Thus, the result holds by induction hypothesis.

Suppose $\rho[k]$ is an update operation with *update* method u at replica r. Let $args = Args(\rho[k])$. Then, $S_r(\rho, \varphi, k) = S_r(\rho, \varphi, k-1).f_u(args)$. $\mathsf{Past}^r_{(\rho,\varphi)}(\rho[0:k]) = \mathsf{Past}^r_{(\rho,\varphi)}(\rho[0:k-1]) \cup \{\rho[k]\}$. Since it is given that, $S_r(\rho, \varphi, k-1) \leq S_r(\rho, \varphi, k-1).f_u(args)$,

$$\begin{aligned} S_r(\rho, \varphi, k-1) \sqcup S_r(\rho, \varphi, k) &\leq S_r(\rho, \varphi, k-1).f_u(args) \sqcup S_r(\rho, \varphi, k) \\ &= S_r(\rho, \varphi, k) \sqcup S_r(\rho, \varphi, k) \\ &= S_r(\rho, \varphi, k) \end{aligned}$$

By induction hypothesis, if $\mathsf{Past}^r_{(\rho,\varphi)}(\rho[0:k-1]) = \{\rho[i_1], \rho[i_2], \ldots, \rho[i_{k'}]\}$, then, $S_r(\rho, \varphi, k-1) = S_\perp . \sqcup S_{i_1} \sqcup S_{i_2} \sqcup \ldots . \sqcup S_{i_{k'}}$. Thus, $S_r(\rho, \varphi, k) = S_\perp \sqcup S_{i_1} \sqcup S_{i_2} \sqcup \cdots \sqcup S_{i_{k'}} \sqcup S_r(\rho, \varphi, k)$. Thus the result is true when $\rho[k]$ is an update operation at r.

Suppose $\rho[k]$ is an **mreceive** operation. Let $j = \varphi(k)$ and $Rep(\rho[j]) = r'$. Then, $S_r(\rho, \varphi, k) = f_{\mathcal{M}}(S_r(\rho, \varphi, k-1), S_{r'}(\rho, \varphi, j))$. It is given that, $f_{\mathcal{M}}(S_r(\rho, \varphi, k-1), S_{r'}(\rho, \varphi, j)) = S_r(\rho, \varphi, k-1) \sqcup S_{r'}(\rho, \varphi, j)$. By induction hypothesis, if
$\mathsf{Past}^r_{(\rho,\varphi)}(\rho[0:k-1]) = \{\rho[k_1], \rho[k_2], \ldots, \rho[k_n]\}$ and $\mathsf{Past}^{r'}_{(\rho,\varphi)}(\rho[0:j]) = \{\rho[j_1], \rho[j_2], \ldots, \rho[j_m]\}$ then,

$$S_r(\rho, \varphi, k-1) = S_\perp \sqcup S_{k_1} \sqcup S_{k_2} \sqcup \cdots \sqcup S_{k_n}$$

and

$$S_{r'}(\rho, \varphi, j) = S_\perp \sqcup S_{j_1} \sqcup S_{j_2} \sqcup \cdots \sqcup S_{j_m}$$

Since $\sqcup$ is the least upper bound, it is idempotent, commutative and associative. Thus, the result is true for k when $\rho[k]$ is an **mreceive** operation at r.

Thus, the result is true for all i. Hence, in any run of $\mathcal{D}_I$, any pair of replicas that have the same causal past have the same state. Thus $\mathcal{D}_I$ satisfies *strong eventual consistency*. Hence $\mathcal{D}_I$ is a CvRDT.

□

In the next chapter, we shall survey in detail a conflict-free replicated data type known as the *Observed-Remove Set (OR Set)*.

2.3 Further Reading

The study of Conflict Free Replicated Data types was undertaken in detail by [Shapiro et al., 2011a]. The formalism to provide the first proofs for the sufficient conditions for state based and operation based replicated data types to be CvRDTs and CmRDTs respectively was discussed in [Shapiro et al., 2011b]. Since then there have been some interesting developments in the fields of state-based and operation-based. We list some of them here, though their detailed study is beyond the scope of this thesis.

[Auvolat and Taïani, 2019] show how state based replicated data types can be efficiently implemented by encoding the states as specialized Merkle trees known as a Merkle Search Tree. This a balanced search tree which maintains key-ordering.

[Almeida et al., 2015] introduce δ-state based conflict free replicated data types that address the draw back of the classical state-based replicated data types which share entire states by defining delta-operators which return delta states. These delta states can be merged both locally as well as remotely to arrive at the resultant final state. [Enes et al., 2019] notices that despite the lower payload size of the δ-state based CRDTs, the synchronization algorithms induce wasteful delta propagation . In this work, they identify the two sources of ineffeciency in the state of the art synchronization algorithms and introduce the concept of join-decomponistion to state-based CRDTs in order to obtain optimal deltas.

Most of the existing CRDTs have no support for a native *undo* operation that can undo the effects of an update. To address this, [Yu et al., 2020] provides an generic "out-of-box" *undo* support for state based CRDTs with the help of an abstraction that captures the semantics of concurrent undo and redo operations through equivalence classes.

[Weidner et al., 2020] presents a new construction that allows composition and decomponsition of operation-based CRDTs using semi-direct products. This helps decompose complex operation-based CRDTs as a semi-direct product of simpler CRDTs. They also demonstrate construction of novel CRDTs by combining the operations of a pair of CRDTs by handing conflicts in a uniform way.

3

Optimized OR-Sets Without Ordering Constraints

A *Set* is a common data type used to represent a collection of elements. For example, in the social network *Twitter*, the collection of all the accounts that a particular user follows can be modelled as a *set*. Likewise the set of all the accounts who follow a user is modelled as a *set*. Furthermore, the collection of all the tweets made by a user can again be modelled by a *set*, only in this case the tweets in the collection are also ordered by the decreasing order of their timestamps. Thus *sets* form the building block of other data types that consist of a collection of elements with possible additional associations between those elements. Examples of this include *lists*, *stacks*, *trees*, *graphs* among other data types.

The abstract data type *set* provides three methods for the clients to interact with it. As per the standard semantics of the *set* data type, the method $\mathsf{add}(x)$ takes as a parameter an element x of the universe and adds it to the set. Again, as per the standard semantics, the method $\mathsf{delete}(x)$ takes as a parameter an element x of the universe and removes x from the set. Lastly the method $\mathsf{contains}(x)$ takes as a parameter an element x of the universe and informs if the element x is present or not in the set. Thus the add and delete methods update the state of the set while the $\mathsf{contains}$ methods allows clients to query the state of the set. Formally, we define the set as follows. Formally, the *Set* is an abstract datatype with $\mathsf{Queries} = \{\mathsf{contains}\}$ and $\mathsf{Updates} = \{\mathsf{add}, \mathsf{delete}\}$ over a universe Univ, and $\mathsf{Rets} = \{\mathsf{True}, \mathsf{False}\}$. The arity of $\mathsf{contains}$, add and delete is 1 since they each take 1 argument.

In this chapter, we explore the replicated implementation of the *set* data type implemented over a distributed system containing multiple replicas. We shall refer to such replicated implementations as *replicated sets*.

In a sequential setting, given a sequence of add and delete operations performed over a set, it is easy to determine at the end of the sequence of operations, whether an element x is present in the set or not by looking at the latest x-operation in the sequence. If the latest

x-operation is an $\mathsf{add}(x)$, then $\mathsf{contains}(x)$ should return True. Otherwise it should return False. However, in a distributed setting, we need a specification of the set that provides coherent semantics while taking into account concurrent update operations delivered to the replicas of the replicated set in potentially different orders.

We can observe that for a pair of distinct elements x and y, $\mathsf{add}(x)$ and $\mathsf{add}(y)$ operations commute with each other. So do the update operations in the pairs $(\mathsf{add}(x), \mathsf{delete}(y))$, $(\mathsf{delete}(x), \mathsf{add}(y))$ and $(\mathsf{delete}(x), \mathsf{delete}(y))$. Furthermore, for the same element x, a operations in the $(\mathsf{add}(x), \mathsf{add}(x))$ and $(\mathsf{delete}(x), \mathsf{delete}(x))$ are idempotent. However operations in the pair $(\mathsf{add}(x), \mathsf{delete}(x))$ do not commute with each other. Suppose one of them *happened before* the other, then the distributed specification can insist that the effect after performing this pair of operations at every replica should be the same as effect of performing these operations in their *happened-before* order, irrespective of the order in which they get delivered at the other replica. Thus if a $\mathsf{delete}(x)$ happened-before an $\mathsf{add}(x)$, then the element x is a member of the set. On the other hand, if $\mathsf{add}(x)$ happened before a $\mathsf{delete}(x)$, then, the element x is not a member of the set. However, if the $\mathsf{add}(x)$ and $\mathsf{delete}(x)$ operations are concurrent, then, the distributed specification must provide a way to arbitrate between these two operations, so that every replica which has performed both the operations, perhaps in a different orders, converges to the same state, i.e. either x is a member of the set in all the replicas which have received these two operations in any order, or x is not a member of the set in all of them.

In this chapter, we shall focus on distributed specifications which favor the $\mathsf{add}(x)$ over a concurrent $\mathsf{delete}(x)$, thereby ensuring that the element x is a member of the set. This is known as the *add-wins* strategy. A replicated set which implements the *add-wins* strategy, where a $\mathsf{delete}(x)$ operation can only *remove* the effects of the $\mathsf{add}(x)$ operations that have *happened-before* that delete (that were *observed* by the delete) is known as an *Observed-Remove Set (OR Set).*

Definition 20 (Observed-Remove Set (OR-Set))**.** *An* Observed-Remove Set *(OR-Set) is a replicated set where the conflict between concurrent* $\mathsf{add}(x)$ *and* $\mathsf{delete}(x)$ *operations is resolved by giving precedence to the* $\mathsf{add}(x)$ *operation so that x is eventually present in all the replicas when both the concurrent operations are delivered [Shapiro et al., 2011a].*

*This conflict resolution strategy is termed as "*add-wins*".*

In this chapter we shall have a detailed look into the different implementations of *Observed-Removed Sets*. These implementations are both state-based as well as operation-based. On every add and a delete operation, the source replica prepares auxiliary information that is sent to all the other replicas which perform the addrcv and delrcv operations respectively. The implementations also provide a $\mathsf{compare}$ function that allows comparison of any two states of the the replica and a merge function which allows the replicas to share their entire

state with other replicas to converge faster. Further more, all these implementations satisfy *strong eventual consistency*. Thus, these implementations are both CvRDTs and CmRDTs.

An implementation of the OR-Set can be formally defined as follows.

Definition 21 (Implementation of OR-Set)**.** *A CRDT implementation of an OR-Set over a universe* Univ *is a tuple*

$$(\mathcal{S}, S_\perp, \Gamma, \{\mathsf{contains}\}, \{(\mathsf{add}, \mathsf{addrcv}), (\mathsf{delete}, \mathsf{delrcv})\}, \mathsf{compare}, \mathsf{merge})$$

where

- $\mathcal{S}$ *is the set of states that the replicas can take.*
- $S_\perp \in \mathcal{S}$ *is the initial state of all the replicas.*
- contains *is the implementations of the query operation that returns* True *or* False *depending on whether a given element is present in the OR-Set or not.*
- add *and* delete *respectively are implementations of the add and delete update operation implementations.*
- addrcv *and* delrcv *correspond to the* **ureceive** *operations of the respective* add *and* delete *operations.*
- Γ *is a set of auxiliary information that the replicas generate during an* add *(resp.* delete*) operation to be sent send to other replicas along with corresponding* addrcv *(resp.* delrcv*).*
- compare *is a method which given a pair of states* $S, S' \in \mathcal{S}$ *will return* True *if* S' *is more up-to-date than* S *and a* False *otherwise.*
- merge *is a function that merges the state of a replica with that of another replica.*

We formally express this as follows:

- $\mathsf{contains} : \mathcal{S} \times \mathsf{Univ} \to \{\mathsf{True}, \mathsf{False}\}$
- $\mathsf{add} : \mathcal{S} \times \mathsf{Univ} \to \mathcal{S} \times \Gamma$.
- $\mathsf{delete} : \mathcal{S} \times \mathsf{Univ} \to \mathcal{S} \times \Gamma$.
- $\mathsf{addrcv} : \mathcal{S} \times \Gamma \to \mathcal{S}$
- $\mathsf{delrcv} : \mathcal{S} \times \Gamma \to \mathcal{S}$
- $\mathsf{compare} : \mathcal{S} \times \mathcal{S} \to \{\mathsf{True}, \mathsf{False}\}$

- $\mathsf{merge} : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$

The organization of this chapter is as follows.

In the next section (section 3.1) we present an implementation of the OR-Set from [Shapiro et al., 2011b], which we call the *original implementation*. This is a robust implementation of the OR-Set which works even when the updates are not delivered in a causal order. However, in order to achieve this, it keeps track of a significant amount of data which is the key drawback of this implementation. In section 3.2 we will discuss an optimized implementation presented in [Bieniusa et al., 2012] which works when the updates are delivered causally. We shall refer to this as the *causally-optimized implementation*. In section 3.4 we shall present our generalization of the *causally-optimized implementation*. We call this the *generic-optimized implementation* [Mukund et al., 2014]. This implementation is as robust as the *original implementation* and works even correctly when the updates are delivered out-of-order or duplicate delivery. Furthermore, if we define the *space complexity* of a replicated data type to be the size of the information that is maintained at each of the replicas as a function of the number of update operations in the run, then the generic optimized implementation has a smaller space complexity than the original implementation and its space complexity is comparable to that of the *causally-optimized implementation*. In section 3.3, we provide a precise formal specification for OR-sets in terms of its runs. This specification that captures the intent of definition 20.

In section 3.7 we shall discuss the space complexity of each of these implementations. Finally in section 3.5 we shall present the proofs of correctness of the generic-optimized implementations and show that it is bisimilar to the original implementation.

3.1 Original implementation of the OR-Set [Shapiro et al., 2011a,b]

Algorithm 1 is a variant of the *original implementation* from [Shapiro et al., 2011a]. This implementation distinguishes between different $\mathsf{add}(x)$ operations performed across the different replicas by labelling the elements of the universe with some meta-data which uniquely identifies that particular $\mathsf{add}(x)$. Towards this, the implementation maintains a triple (x, c, r) where r denotes the source replica where the corresponding $\mathsf{add}(x)$ operation was performed and c denotes that fact that the corresponding $\mathsf{add}(x)$ operation was the c^{th} add operation at replica r. We shall define the set of such labeled elements as the *labeled-universe* formally defined as follows:

Definition 22 (Labeled Universe)**.** *The* labeled universe *is the set of triples*

$$\mathcal{M} = \{(x, c, r) \mid x \in \mathsf{Univ}, c \in \mathbb{N}, r \in \mathcal{R}\}$$

Algorithm 1 An Original OR-Set implementation

An Original OR-set implementation for replica r

```
1  E ⊆ M, T ⊆ M, c ∈ ℕ: initially ∅, ∅, 0.
2
3  Boolean CONTAINS(x ∈ Univ):
4        return (∃m : m ∈ E ∧ data(m) = x)
5
6  ADD(x ∈ Univ):
7        m = ADD.PREPARE(x)
8        Call ADD.APPLY(m)
9        Broadcast ADD.send(m)
10 ADD.PREPARE(x ∈ Univ):
11       return (x, c, r)
12 ADD.APPLY(m ∈ M):
13       E := (E ∪ {m}) \ T
14       if (rep(m) = r)
15             c := ts(m) + 1
16 Receive ADD.receive(m ∈ M):
17       Call ADD.APPLY(m)
18
19 DELETE(x ∈ Univ):
20       Let M := DELETE.PREPARE(x)
21       Call DELETE.APPLY(M)
22       Broadcast DELETE.send(M)
23 DELETE.PREPARE(x ∈ Univ):
24       return M := {m ∈ E | data(m) = x}
25 DELETE.APPLY(M ⊆ M):
26       E := E \ M
27       T := T ∪ M
28 Receive DELETE.receive(M ⊆ M):
29       Call DELETE.APPLY(M)
30
31 Boolean COMPARE(S', S'' ∈ S):
32       Assume that S' = (E', T', c')
33       Assume that S'' = (E'', T'', c'')
34       Let b_seen := (E' ∪ T') ⊆ (E'' ∪ T'')
35       Let b_deletes := T' ⊆ T''
36       return b_seen ∧ b_deletes
37
38 MERGE(S' ∈ S):
39       Assume that S' = (E', T', c')
40       E := (E \ T') ∪ (E' \ T)
41       T := T ∪ T'
```

For a triple $m = (x, c, r) \in \mathcal{M}$, we say $data(m) = x$ (the data or payload), $ts(m) = c$ (the timestamp), and $rep(m) = r$ (the source replica).

In Algorithm 1, each replica maintains a local set $E \subseteq \mathcal{M}$. When replica r receives an $\mathsf{add}(x)$ operation from the client, it invokes the ADD.PREPARE method which tags x with a unique identifier (c, r) (line 7,10,11), where this $\mathsf{add}(x)$ operation is the c^{th} *add* operation overall at r. It then invokes the ADD.APPLY operation which will add the triple (x, c, r) to its local copy of E and increment the value of the counter c (lines 8,12-15). Finally it broadcasts the triple (x, c, r) to all the replicas via the Broadcast ADD.**send**(x, c, r) method (line 9). The other replicas, on receiving this broadcast via the Receive ADD.**receive**(x, c, r) (lines 16–17) will add (x, c, r) to their local set E via the ADD.APPLY method (line 13).

Symmetrically, deleting an element x involves identifying every triple m from E with $data(m) = x$. This is done via the DELETE.PREPARE method (lines 20,23-24) which returns the set $M \subseteq E$ containing all the triples m with $data(m) = x$. These triples in M are removed from the E set via DELETE.APPLY method (lines 21,25-27). Finally this set of triples are propagated to the other replicas via Broadcast DELETE.**send**(M) (line 22). Those replicas on receiving this set of triples via Receive DELETE.**receive**(M) (Lines 28–29) will remove the elements in M from their local E set, via DELETE.APPLY (line 25-27).

For the rest of the paper, we shall identify "Receive ADD.**receive**()" with $\mathsf{addrcv}()$ and "Receive DELETE.**receive**()" with $\mathsf{delrcv}()$.

When a replica receives an addrcv(m), it should add the corresponding triple m to its local copy of E. However, with no constraints on the delivery of operations, it is possible that a delrcv operation corresponding to a delete(x) is delivered before the addrcv(m) operation corresponding to an add(x) which *happened-before* the delete(x). For example in Figure 3.1, replica r'' receives delrcv($\{(x, c+1, r)\}$)) before it receives addrcv($(x, c+1, r)$). Alternatively, after applying a delrcv, a replica may merge its state with that of another replica that has not performed this delrcv, but has performed the corresponding addrcv(). For instance, in Figure 3.1, replica r'' merges its state with replica r when r'' has applied delrcv($\{(x, c+1, r)\}$) but r has not. In such cases, we need to ensure that the triple m which has been deleted in the past is not added back into E. Towards this, each replica maintains a set T of *tombstones*, containing every triple m ever deleted (line 27). Thus, before adding m to E during an addrcv, the replica first checks that it is not in T (line 13).

We say that a state S'' of a replica r'' has more knowledge than a state S' of replica r' if r'' has seen all the triples present in S' (either through an add or a delete) and r'' has deleted all the triples that r' has deleted. This is checked by compare (lines 31–36) where compare(S', S'') returns True if S'' has more knowledge than S'. When compare(S', S'') returns False it only means that S'' does not have more knowledge than S' and not that S' is more up-to-date than S'' (it could be the case that neither state has more knowledge than the other).

Finally, the merge function of states S and S' retains only those triples from $S.E \cup S'.E$ that have not been deleted in either S or S' (line 40). The merge function also updates the *tombstone set* to contain the triples that have been deleted in either S or S' (line 41).

We can note that if there is a pair of concurrent add(x) and delete(x) operations involving the same element $x \in$ Univ, the unique identifier (c, r) that the source replica of add(x) tags to x will be different from the identifiers of x seen by the source replica performing the delete(x). Thus, when the addrcv(m) and delrcv(M) operations of these add and delete are propagated to all the replicas, the recepient replicas will not evict the triple (x, c, r) as they won't be a part of the argument of delrcv(M) operation of delete(x). This is true even when the addrcv() operation gets delivered after the delrcv operation, since (x, c, r) won't be in the tombstone set T of the recipient of the addrcv(). Thus, in this way the add(x) operation *wins* over the concurrent delete(x) operation as its triple (x, c, r) survives in the E set at all the recipient replicas after both the add and delete are delivered. At this point a contains(x) query made at any such recipient replica shall return True.

Problem of Tombstones

Note that in the original implementation, the triple (x, c, r) introduced into the state of any replica, during an add(x) operation, will forever remain in the state even after it has been deleted, as in the latter case it is merely shifted from the E set to the tombstone-set T. The tombstone set T is never purged, and hence the size of $E \cup T$ is proportional to the number

of adds that the replica has received (directly or through other replicas). A solution to this was proposed in [Bieniusa et al., 2012], where they implemented an optimized version of the OR-set, but it works only in the presence of causal delivery of updates.

3.2 Optimized OR-Set with causal-delivery [Bieniusa et al., 2012]

Algorithm 2 An optimized OR-Set implementation with causal-delivery

Optimized OR-set implementation for the replica r with causal delivery constraint

```
1  E ⊆ M, V : R → N, c ∈ N: initially ∅, [0, ··· , 0], 0
2
3  Boolean CONTAINS(e ∈ Univ):
4      return (∃m : m ∈ E ∧ data(m) = e)
5
6  ADD(x ∈ Univ):
7      Let m := ADD.PREPARE(x)
8      Call ADD.APPLY(m)
9      Broadcast ADD.send(m)
10 ADD.PREPARE(x ∈ Univ):
11     return (x, c, r)
12 ADD.APPLY(m ∈ M):
13     if ts(m) > V[rep(m)]
14         E := E ∪ {m}
15         V[rep(m)] := ts(m)
16         if (rep(m) = r)
17             c := ts(m) + 1
18 Receive ADD.receive(m ∈ M):
19     Call ADD.APPLY(m)
20
21 DELETE(x ∈ Univ):
22     Let V' := DELETE.PREPARE(x)
23     Call DELETE.APPLY((x, V'))
24     Broadcast DELETE.send(x, V')
25 DELETE.PREPARE(x ∈ Univ):
26     Let V' := [0, ..., 0]
27     for r' ∈ R
28         Let C := {ts(m) | m ∈ E∧
                        (data(m), rep(m)) = (x, r')}
29         V'[r'] := max(C)
30     return V'
31
32 DELETE.APPLY(x ∈ Univ, V' : R → N)
33     Let M := {m ∈ E | ts(m) ≤ V'[rep(m)]∧
                         data(m) = x}
34     E := E \ M
35 Receive DELETE.receive(x ∈ Univ, V' : R → N):
36     Call DELETE.APPLY(x, V')
37
38 Boolean COMPARE(S', S'' ∈ S):
39     Assume that S' = (E', V')
40     Assume that S'' = (E'', V'')
41     b_seen := ∀i(V'[i] ≤ V''[i])
42     b_deletes := ∀m ∈ E'' \ E'
                (ts(m) > V'[rep(m)])
43     // If m is deleted from E' then
44     // it is also deleted in E''.
45     // So anything in E'' \ E'
46     // is not even visible in S'.
47     return b_seen ∧ b_deletes
48
49 MERGE(S' ∈ S):
50     Assume that S' = (E', V')
51     E := (E ∩ E') ∪
             {m ∈ E \ E' | ts(m) > V'[rep(m)} ∪
             {m ∈ E' \ E | ts(m) > V[rep(m)}
52     // We retain m if it is either
53     // in the intersection, or if it is fresh
54     // (so one of the states has not seen it).
55     ∀i.(V[i] := max(V[i], V'[i]))
```

When updates are delivered in a causal manner, every replica that has seen a $\mathsf{delete}(x)$ would have also seen all the $\mathsf{add}(x)$ operations that causally precede the $\mathsf{delete}(x)$. In particular, they would have already seen all the $\mathsf{add}(x)$ whose corresponding entries in the E set are evicted by $\mathsf{delete}(x)$. Thus, in the presence of causal delivery, it suffices for an operation-

based implementation of OR-Set to maintain only the E set. Algorithm 2 from [Bieniusa et al., 2012] does not maintain the T set. Whenever an element is added, it is labeled with the unique (c, r) pair and added to the E set (lines 7,11 in Algorithm 2). On the other hand, when an element x is to be deleted, all the triples $m \in E$ with $data(m) = x$ are marked for removal (lines 22,26–30). At the source replica, these elements are removed from the E set (lines 23,32–34) and propagated to all the replicas(line 24). When the replicas receives this (line 35–36), they will evict all these triples m from their E set(lines 36,32–34). Thus, in a operation-only implementation where the operations are causally delivered, it suffices to maintain only the E set.

However, in the presence of a merge, things can get a little complicated. Suppose a replica r' receives a $\mathsf{delrcv}(\{m = (x, c, r)\})$ due to which it evicts the triple m from its E set. Soon after this, suppose it receives a merge from another replica r''. Further, suppose r'' hadn't received the corresponding delrcv before it issued the merge, but has received the $\mathsf{addrcv}(m)$. (This is possible since merges may occur between the causal-delivery of updates) Thus the triple m would be present in the E set of r''. On receiving the merge request, r' won't be able to judge based only on the contents of the $r'.E$ and $r''.E$ as to whether the triple m is a new one in $r''.E$ which it hasn't seen before, or is it something that it has already seen and removed.

To address this problem, [Bieniusa et al., 2012] proposes the use of a version vector [Almeida et al., 2004] at each replica to keep track of the latest add operation that it has received from every other replica. Thus every time an element is to be added (either via $\mathsf{add}(x)$ or via the corresponding $\mathsf{addrcv}(m = x, c, r)$), the replica that is doing the add will update the version vector corresponding to the replica from which it is receiving the add operation to the value of counter of the triple (line 15). Since updates are causally delivered, all the adds are causally delivered. In particular, the add operations of a particular replica are causally delivered. Thus it suffices to record the counter corresponding to the latest add operation of every replica in the system. The version-vector isn't updated during the delete operation, since it only tracks the add operations from all the replicas.

When a merge happens, the replica r' inspects the set $r''.E$ and retains only those triples m which are either already in $r'.E$ (implying that r' knows about this $\mathsf{add}(x, c, r)$ and hasn't yet deleted it) or $(x, c, r) \notin r'.E$ but $c > r'.V[r]$ (implying that r' hasn't yet seen the $\mathsf{add}(x, c, r)$) (line 51). The remaining triples from $r''.E$ are discarded since they have already been seen and deleted at r'. The version vectors are also updated by taking the point-wise maximum, since after the merge, r' has seen all the adds that r'' has seen (line 55).

Note that a triple (x, c, r) is said to be deleted in a state S if $(x, c, r) \notin S.E$ and $S.V[r] > c$. This tells us that the state S has already seen the triple, and has not retained it in its E set, which implies that it has deleted it.

Thus, the $\mathsf{compare}(S', S'')$ method (lines 38–47) returns True if the state S'' has more knowledge than S'. This happens when

- S'' has seen all the adds that S' has seen. This can be inferred by point-wise comparing the respective version vectors $S''.V[i]$ and $S'.V[i]$ for every replica $i \in \mathcal{R}$ (line 41).
- S'' performed all the deletes that S' has. This can be inferred by checking if all the surviving triples in $S''.E \setminus S'.E$ are those that S' hasn't even seen, which implies that there are no undeleted triples in S'' that S' has seen and deleted (line 42).

We can observe that while the original algorithm keeps data for every add operation, the optimized algorithm from [Bieniusa et al., 2012] keeps data for only those add operation which haven't been observed by a delete, in addition to a version vector corresponding to all the add-operations.We shall formally show in section 3.7 that optimized algorithm requires lesser space to store the state of a replica. However, this comes at the price of causal-delivery of updates.

Critique of Causal Delivery for OR-Sets

Causal delivery imposes unnecessary restrictions on the delivery of independent updates. For example, updates at a source replica of the form $\mathsf{add}(x)$ and $\mathsf{delete}(y)$, for distinct elements x and y, need not be delivered in the same order to all other replicas. The ideal case would be to have causal delivery of updates involving the same element x. While this is weaker than causal delivery across all updates, it puts an additional burden on the underlying delivery subsystem to keep track of the partial order of updates separately for each element in the universe. A weaker delivery constraint is FIFO, which delivers updates originating at the same source replica in the order seen by the source. However, this is no better than out-of-order delivery since causally related operations on the same element that originate at different sources can still be delivered out-of-order.

Additionally, it can be noted that the *original implementation* is quite robust and works correctly even when updates are delivered out-of-order. Thus if there is a way to efficiently track the deleted elements even in the absence of causal-delivery, we will have an optimization over the original implementation.

However, before going towards the optimized implementation which is robust against out-of-order delivery, we recognize the non-trivial challenges pertaining to reasoning about the state of the replicas in the absence of any delivery guarantees. So we shall first illustrate the challenges posed by out-of-order delivery and then formalize a concurrent specification for OR-Sets that is independent of delivery guarantees.

3.3 Distributed Specification of OR-Sets

If we assume causal delivery of updates, then it is easy to see that all replicas apply non-concurrent operations in the same order. The *happens-before* relation $\xrightarrow{\mathsf{hb}}$ is thus transitive

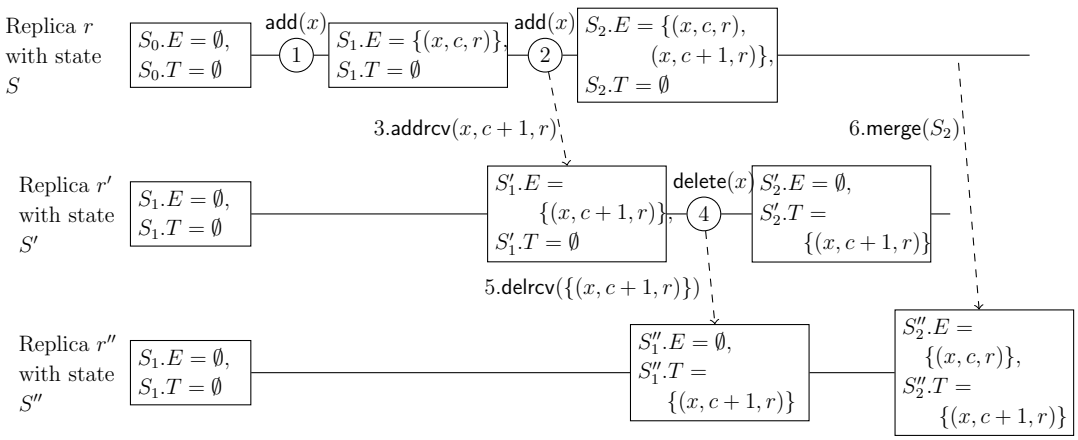


Figure 3.1: Non-transitivity of the happened-before relation.

and a partial order. At the end of a run $\alpha = (\rho, \varphi)$ for a replica r, we can look at the x-update operations in $\mathsf{Past}^r_{(\rho,\varphi)}(\rho)$ partially ordered by $\xrightarrow{\mathsf{hb}}$. We say that an element x is present at a replica if the maximal set of x-operations with respect to the $\xrightarrow{\mathsf{hb}}$ relation in the causal past contains at least one $\mathsf{add}(x)$. Because that $\mathsf{add}(x)$ would be concurrent with all the other maximal x operations, and even if all those others are $\mathsf{delete}(x)$ operations, by the *add-wins* semantics, x will be deemed to be *present* in the set. On the other hand, if the set of maximal x update operations in the causal past of a replica does not contain an $\mathsf{add}(x)$ operation, then the element x is treated to be not present at that replica. Thus, when the updates are delivered in an order consistent with their happened-before relation, reasoning about the state of a replica is fairly straightforward.

However, this straightforward specification fails to hold when the updates are delivered out-of-order. The reason being, in the absence of causal delivery, even non-concurrent operations can exhibit counter-intuitive behaviours. We identify a couple of them in Examples 23 and 24.

Example 23. *In the absence of causal-delivery, the happened-before relation need not be transitive. For instance, in Figure 3.1, if we denote the add operations at* 1 *and* 2 *as* $\mathsf{add}_1(x)$ *and* $\mathsf{add}_2(x)$*, respectively, then we can observe that* $\mathsf{add}_1(x) \xrightarrow{\mathsf{hb}} \mathsf{add}_2(x)$ *and* $\mathsf{add}_2(x) \xrightarrow{\mathsf{hb}} \mathsf{delete}(x)$*. However, it is not the case that* $\mathsf{add}_1(x) \xrightarrow{\mathsf{hb}} \mathsf{delete}(x)$ *since the source replica of* $\mathsf{delete}(x)$*, which is* r'*, has not processed the* addrcv *of* $\mathsf{add}_1(x)$ *before processing the* $\mathsf{delete}(x)$ *update request at* 4*. Hence when the merge operation at* 6 *occurs, the triple added by* $\mathsf{add}_1(x)$ *survives in the state* S'' *while the triple added by subsequent add operation* $\mathsf{add}_2(x)$ *remains evicted. Thus, at* S'' *even though* $\mathsf{add}_1(x)$, $\mathsf{add}_2(x)$ *and* $\mathsf{delete}(x)$ *are present in the causal past of the replica* r''*, and despite the fact that there is no* $\mathsf{add}(x)$ *operations among the maximal* x*-update operations in the causal past, the* $\mathsf{contains}(x)$ *at* S'' *would return* True*.*

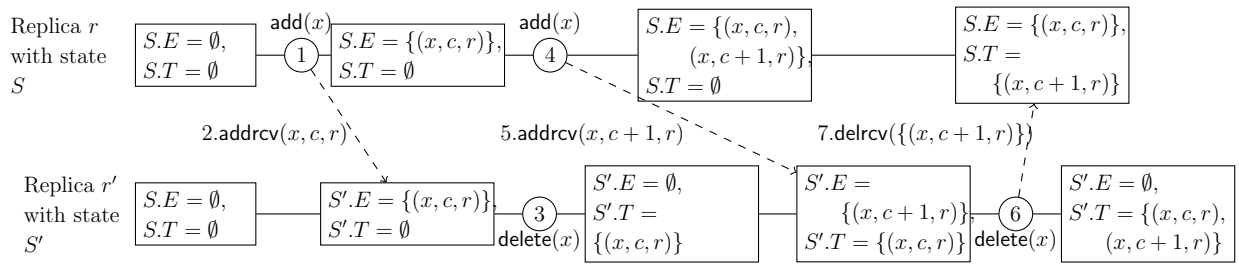


Figure 3.2: Non-intuitive behaviour of *deletes* in the absence of causal delivery.

Example 24. *In the absence of causal delivery, sometimes a* delrcv(x) *may not remove all copies of x from the set—even copies corresponding to* add(x) *operations that happened before. Consider Figure 3.2. Say (x, c, r) is added at r at 1, propagated to r' at 2, and subsequently deleted at r' at 3. Suppose $(x, c+1, r)$ is later added at r at 4, propagated to r' at 5, and subsequently deleted at r' at 6. If the delete at 6 is propagated from r' to r before the delete at 3, then the* delrcv$(\{(x, c+1, r)\})$ *at r removes only $(x, c+1, r)$ while retaining (x, c, r), as illustrated in Figure 3.2.*

Thus, the non-transitivity of the *happened-before* relation, and the non-intuitive behaviour of delrcv operations not necessarily getting rid of all the elements added by *happened-before* add operations, introduce challenges for reasoning about the correctness of the distributed specification of OR-Sets from its sequential specification.

Hence, to address these issues, we need a more precise formulation of the concurrent specification that captures the intent of Definition 20 (for any element $x \in \mathsf{Univ}$, when concurrent updates add(x) and delete(x) are applied at any state, the add(x) should win over delete(x)) and allows us to uniformly reason about the states of the replicas of OR-Sets independent of the order of delivery of updates. Towards this, we define the *covering-delete* operation of an *add* operation.

Definition 25 (Covering Delete of an Add)**.** *Let (ρ, φ) be a run of an OR-Set. Let $\rho[i]$ be an* add *operation with $Args(\rho[i]) = x$. Then we define the covering delete of $\rho[i]$ to be the following set:*

$$\mathsf{CoveringDel}(\rho[i]) = \{\rho[j] \mid \rho[j] = \mathsf{delete}(x) \wedge \rho[i] \xrightarrow{\mathsf{hb}} \rho[j] \wedge \forall k : \rho[k] = \mathsf{delete}(x) \wedge \rho[i] \xrightarrow{\mathsf{hb}} \rho[k] \implies \neg(\rho[k] \xrightarrow{\mathsf{hb}} \rho[j])\} \quad (3.1)$$

Thus, intuitively we can now say that at the end of the run $\alpha = (\rho, \varphi)$, at a replica r, contains(x) should return True iff there exists an add(x) operation in the causal past of r, which has no *covering-delete*. We formally define this as follows:

Definition 26 (Specification of OR-Set [Mukund et al., 2014]). *For any reachable state S in a run (ρ, φ) and element $x \in \mathsf{Univ}$, $S.\mathsf{contains}(x) = \mathsf{True}$ iff there exists a $\rho[i] = \mathsf{add}(x) \in \mathsf{Past}(S)$ such that $\mathsf{CoveringDel}(\rho[i]) \cap \mathsf{Past}(S) = \emptyset$.*

We will now show that this definition captures the intention of definition 20.

Let $\{u_1, u_2, \ldots, u_n\}$ be a set of update operations where every pair of operations in the set is concurrent with each other. We denote this by $u_1 \parallel u_2 \parallel \cdots \parallel u_n$. Assume that these operations are applied at a replica with state S which has no other undelivered operation. Let S' denote the state obtained by applying $u_1, \cdots, u_n$ to S. We shall write this as $S' = S \circ (u_1 \parallel u_2 \parallel \cdots \parallel u_n)$. If one of the u_i's is $\mathsf{add}(x)$, it is clear that $\mathsf{CoveringDel}(u_i) \cap \mathsf{Past}(S') = \emptyset$. Thus, $S' \circ \mathsf{contains}(x) = \mathsf{True}$. Thus, the concurrent add wins.

This specification also explains Examples 23 and 24. In Example 23, the $\mathsf{add}(x)$ operation at 1 does not have a covering delete in $\mathsf{Past}(S_2'')$, which explains why $S_2''.\mathsf{contains}(x) = \mathsf{True}$. Similarly in the other example, the $\mathsf{add}(x)$ operation at 1 does not have a covering delete in the history of replica r, but it has a covering delete (operation 3) in the history of replica r'. This explains why x is *present* in the final state of r but does not belong to the final state of r'.

In the next section we shall provide an optimized implementation of OR-Set that doesn't require updates to be delivered in a causal manner 3. In section 3.5 we prove that this optimized implementation satisfies the specification given in definition 26. We shall also show that this optimized implementation(Algorithm 3) is *bisimilar* to the original implementation(Algorithm 1), thereby showing that the specification captures the intuitive idea of *add-wins* resolution envisioned by the creators of the original implementation.

3.4 Generic Optimized Implementation [Mukund et al., 2014]

Algorithm 3 describes our optimized implementation of OR-sets that does not require causal ordering and yet uses space comparable to the solution provided in [Bieniusa et al., 2012]. Our main observation is that tombstones are only required to track $\mathsf{delete}(x)$ operations that overtake $\mathsf{add}(x)$ operations from the same replica. Since a source replica attaches a timestamp (c, r) with each $add(x)$ operation, all we need is a succinct way to keep track of those timestamps that are "already known". We define *Interval Version Vectors* for this purpose. We first define *Interval Sequences* which is a representation of a set of non-overlapping intervals of integers.

Definition 27 (Interval Sequences). *For a pair of integers $s \leq \ell$, $[s, \ell]$ denotes the* interval *consisting of all integers from s to ℓ. A finite set of intervals $\{[s_1, \ell_1], \ldots, [s_n, \ell_n]\}$ is*

Algorithm 3 An optimized OR-Set implementation without any ordering constraints

Optimized OR-set implementation without ordering constraints for the replica r

```
1  E ⊆ M, V : R → I, c ∈ ℕ: initially ∅, [∅, ..., ∅], 0
2
3  Boolean CONTAINS(x ∈ Univ):
4      return (∃m : m ∈ E ∧ data(m) = x)
5
6  ADD(x ∈ Univ):
7      Let m = ADD.PREPARE(x)
8      Call ADD.APPLY(m)
9      Broadcast ADD.send(m)
10 ADD.PREPARE(x ∈ Univ):
11     return (x, c, r)
12 ADD.APPLY(m ∈ M):
13     if (ts(m) ∉ V[rep(m)])
14         E := E ∪ {m}
15         V[rep(m)] :=
               add(V[rep(m)], {ts(m)})
16     if (rep(m) = r)
17         c = ts(m) + 1
18 Receive ADD.receive(m ∈ M):
19     Call ADD.APPLY(m)
20
21 DELETE(x ∈ Univ):
22     Let V' := DELETE.PREPARE(x)
23     Call DELETE.APPLY(V')
24     Broadcast DELETE.send(V')
25 DELETE.PREPARE(x ∈ Univ):
26     Let V' : R → I = [0, ..., 0]
27     for m ∈ E with data(m) = x
28         add(V'[rep(m)], {ts(m)})
29     return V'
30 DELETE.APPLY(V' : R → I):
31     Let M = {m ∈ E |
                   ts(m) ∈ V'[rep(m)]}
32     E := E \ M
33     for i ∈ R
34         V[i] := V[i] ∪ V'[i]
35 Receive DELETE.receive(V' : R → I):
36     Call DELETE.APPLY(x, V')
37
38 Boolean COMPARE(S', S'' ∈ S):
39     Assume that S' = (E', V')
40     Assume that S'' = (E'', V'')
41     b_seen := ∀i(V'[i] ⊆ V''[i])
42     b_deletes := ∀m ∈ E'' \ E'
               (ts(m) ∉ V'[rep(m)])
43     // If m is deleted from E' then
44     // it is also deleted in E''.
45     // So anything in E'' \ E'
46     // is not even visible in S'.
47     return b_seen ∧ b_deletes
48
49 MERGE(S' ∈ S):
50     Assume that S' = (E', V')
51     E := {m ∈ E ∪ E' |
           m ∈ E ∩ E' ∨
           ts(m) ∉ V[rep(m)] ∩ V'[rep(m)]}
52     // We retain m if it is either
53     // in the intersection, or if it is fresh
54     // (so one of the states has not seen it).
55     ∀i.(V[i] := V[i] ∪ V'[i])
```

nonoverlapping *if for all distinct* $i, j \leq n$, *either* $s_i > \ell_j$ *or* $s_j > \ell_i$. *An* interval sequence *is a finite set of nonoverlapping intervals. We denote by* $\mathcal{I}$ *the set of all interval sequences.*

We define some basic operations that can be performed on Interval Sequences below. The function $\textsc{pack}(X)$ collapses a set of numbers X into an interval sequence. The function $\textsc{unpack}(A)$ expands an interval sequence to the corresponding set of integers. Fundamental set operations can be performed on interval sequences by first unpacking and then packing.

Definition 28 (Operations on Interval Sequences)**.** *For* $X \subseteq \mathbb{N}$, $A, B \in \mathcal{I}$, *and* $n \in \mathbb{N}$*:*

- $\textsc{pack}(X) = \{[i, j] \mid \{i, i+1, \ldots, j\} \subseteq X, i-1 \notin X, j+1 \notin X\}$.
- $\textsc{unpack}(A) = \{n \mid \exists [n_1, n_2] \in A \wedge n_1 \leq n \leq n_2\}$.
- $n \in A$ *iff* $n \in \textsc{unpack}(A)$.
- $\boldsymbol{add}(A, X) = \textsc{pack}(\textsc{unpack}(A) \cup X)$.
- $\boldsymbol{delete}(A, X) = \textsc{pack}(\textsc{unpack}(A) \setminus X)$.
- $\boldsymbol{max}(A) = max(\textsc{unpack}(A))$.
- $A \cup B = \textsc{pack}(\textsc{unpack}(A) \cup \textsc{unpack}(B))$.
- $A \cap B = \textsc{pack}(\textsc{unpack}(A) \cap \textsc{unpack}(B))$.
- $A \subseteq B$ *iff* $\textsc{unpack}(A) \subseteq \textsc{unpack}(B)$.

We can now define Interval Version Vector, which is a generalization of Version Vectors from [Almeida et al., 2004]. Recall that if V is a regular interval vector, then for a replica r, $V[r]$ would denote the latest r-message that is known. With Interval Version Vectors, $V[r]$ denotes the Interval Sequence corresponding to the messages received from r. Thus, we can keep track of the messages received from r even if they are delivered out of order. Formally,

Definition 29 (Interval Version Vector)**.** *An* Interval Version Vector *over* $\mathcal{R}$ *is a vector* V *of length* $|\mathcal{R}|$ *where for each* $r \in \mathcal{R}$, $V[r] \in \mathcal{I}$.

As in the algorithm of [Bieniusa et al., 2012], when replica r receives an $\mathsf{add}(x)$ operation, it tags x with a unique identifier (c, r) and propagates (x, c, r) to be added to E (Lines 7,11 in Algorithm 3). In addition, each replica r maintains the set of all timestamps c received from every other replica r' as an interval sequence $V[r']$. Thus, the vector V is an *Interval Version Vector*. Since all the **ureceive** operations with source replica r are applied at r in causal order, $r.V[r]$ contains a single interval $[1, c_r]$ where c_r is the index of the latest add operation received by r from a client. Notice that if $\mathsf{delete}(x)$ at a source replica r'

is a *covering delete* for an $\mathsf{add}(x)$ operation, then the unique identifier $(ts(m), rep(m))$ of the triple m generated by the add operation will be included in the interval version vector prepared via DELETE.PREPARE() (lines 26-29) and will be broadcast to the other replicas by the delete operation (Line 24). When this vector arrives at a replica r as a part of delrcv, r updates the interval sequence $V[rep(m)]$ to record the missing add operation (lines 33–34) so that, when m eventually arrives to be added through the addrcv operation, it can be ignored (line 13).

Thus, we can avoid maintaining tombstones altogether without insisting on causal-delivery. The price we pay is maintaining a collection of interval sequences, but these interval sequences will eventually get merged once the replica receives all the pending updates, collapsing the representation to contain at most one interval per replica.

In [Bieniusa et al., 2012], the authors suggest a solution in the absence of causal delivery using *version vectors with exceptions (VVwE)*, proposed in [Malkhi and Terry, 2007]. A VVwE is an array each of whose entries is a pair consisting of a timestamp and an *exception set*, and is used to handle out-of-order message delivery. For instance, if replica r sees operations of r' with timestamps $1, 2$, and 10, then it will store $(10, \{3, 4, 5, 6, 7, 8, 9\})$, signifying that 10 is the latest timestamp of an r'-operation seen by r, and that $\{3, 4, \ldots, 9\}$ is the set of operations that are yet to be seen. The same set of timestamps would be represented by the interval sequence $\{[1, 2], [10, 10]\}$. In general, it is easy to see that interval sequences are a more succinct way of representing timestamps in systems that allow out-of-order delivery.

In the next section, we prove the correctness of the generic optimized implementation.

3.5 Correctness of the Generic Optimized Implementation

In this section we prove the correctness of our optimized solution by showing that the solution satisfies the concurrent specification of OR-Sets. We also show that the implementation is a CvRDT as well as a CmRDT. In the next section, we provide a proof of equivalence between the original implementation and our optimized implementation.

We shall define a few terms that we shall be referring to in the proofs.

We use r to denote replicas in $\mathcal{R}$, x, y to denote elements of Univ, and m to denote elements of $\mathcal{M}$, with superscripts and subscripts as needed. For $m = (x, c, r) \in \mathcal{M}$, recall that we have defined $data(m) = x$ (the data or payload), $ts(m) = c$ (the timestamp), and $rep(m) = r$ (the source replica). We define a valid set of labeled elements as follows:

Definition 30 (Valid set of labelled elements)**.** *A set of labelled elements $M \subseteq \mathcal{M}$ is said to be* valid *if it does not contain distinct items from the same replica with the same timestamp.*

Formally,

$$\forall m, m' \in M : (ts(m) = ts(m') \wedge rep(m) = rep(m')) \implies m = m'$$

Given a reachable state S of some replica r, we define the following notations.

- $v = S \circ \mathsf{contains}(x)$ denotes that the return value of application of the query $\mathsf{contains}(x)$ to the state S is v.
- $S' = S \circ u$ denotes the fact that the state of the replica after application of the update u is S' where $u \in \{\mathsf{add}(x), \mathsf{delete}(x) \mid x \in \mathsf{Univ}\} \cup \{\mathsf{addrcv}(m) \mid m \in \mathcal{M}\} \cup \{\mathsf{delrcv}(M) \mid M \in 2^{\mathcal{M}}, M \text{ is valid}\}$.
- $S' = S \circ \mathsf{merge}(S'')$ denotes that the resultant state obtained by merging S'' into S is S'.

We say that two states S and S' are *equivalent* and write $S \equiv S'$ iff $S.E = S'.E$ and $S.V = S'.V$. It is easy to see that if S and S' are equivalent then they are also query-equivalent. Note that S and S' can differ in the $S.count$ and $S'.count$ values because for a local replica with state S, $S.count$ gets incremented only when a replica receives an $\mathsf{add}(x)$ from a client. However, a remote replica might receive the corresponding $\mathsf{addrcv}(m)$ which results in the $S'.E$ and the $S'.V$ of the remote replica to be the same as $S.E$ and $S.V$ respectively. However, since the $S'.count$ is not updated during an *update-receive* operation, $S'.count$ may not be the same as $S.count$.

We now show that when an $\mathsf{add}(x)$ operation is applied at a state S, the resultant state is equivalent to one where the corresponding $\mathsf{addrcv}(m)$ operation is applied at S. This will help us uniformly reason about changes made by an $\mathsf{add}(x)$ operation and its corresponding $\mathsf{addrcv}(m)$.

Proposition 31. *Let S be state of some replica. Let m be the triple generated by the* ADD.PREPARE(x) *helper method of of the add operation* ADD(x) *of Algorithm 3 when applied to S. Let the state after applying the* ADD(x) *be S', i.e. $S' = S \circ \mathsf{add}(x)$. Let S'' be the state of the replica if $\mathsf{addrcv}(m)$ were applied at S, i.e $S'' = S \circ \mathsf{addrcv}(m)$.*

Then $S' \equiv S''$

Proof. We observe that the invocation of ADD.PREPARE(x) at line 7 in algorithm 3 has no side-effects on the state of the replica. On line 8, the algorithm invokes ADD.APPLY(m) where m is the triple generated by ADD.PREPARE(x). This updates the state of S to S' where $S'.E$ and $S'.V$ are updated in lines 14,15 to incorporate the triple m generated by ADD.PREPARE(x).Subsequently, the broadcast operation on line 9, doesn't modify the state at the replica. So after the broadcast operation, and in effect after the add operation, the state of the replica,$S' = S \circ \mathsf{add}(x)$

If on the other hand, with the triple m generated by ADD.PREPARE(x), if

Receive ADD.**receive**(m)

were applied at a state S, it would invoke ADD.APPLY(m) on line 19. Note that when applied to the same state, the operation ADD.APPLY(m) updates the state in the same fashion in lines 14,15 to incorporate m into $S.E$ and $S.V$ to yield a state S''.

Since the generated triple m is same in both cases, and the code incorporating m into the starting state S is the same, it is clear that $S'.E = S''.E$ and $S'.V = S''.V$..

Hence, $S' \equiv S''$. □

We can show a similar result for the deletes as well. Since the proof is quite similar, we omit it, and present the result below:

Proposition 32. *Let S be state of some replica. Let V' be the interval version vector generated by* DELETE.PREPARE(x) *of the add operation* DELETE(x) *of Algorithm 3 when applied to S. Let $S' = S \circ \mathsf{delete}(x)$ and $S'' = S \circ \mathsf{delrcv}(V')$. Then $S' = S''$*

From propositions 31 and 32 we can see that without loss of generality an $\mathsf{add}(x)$ operation (respectively a $\mathsf{delete}(x)$ operation) at a replica r can be treated as invoking a ADD.PREPARE(x) (respectively DELETE.PREPARE(x)) which generates an argument without having any side-effect on the state. The argument is then broadcast to all the replicas including r where it is *applied* soon after the broadcast is sent. Thus the state at r gets updated when the corresponding receive operation $\mathsf{addrcv}(m)$ (respectively $\mathsf{delrcv}(V')$) is broadcast and applied on the same replica soon-after the $\mathsf{add}(x)$ operation (respectively the $\mathsf{delete}(x)$ operation). This allows us to reason about the state changes caused by the $\mathsf{add}(x)$ and $\mathsf{delete}(x)$ operations in a uniform manner via their respective receive operations.

A receive operation u is said to be an x-add-receive operation (respectively, x-delete-receive operation) if it is a receive operation of an $\mathsf{add}(x)$ (respectively, $\mathsf{delete}(x)$) operation. If $\mathcal{O}$ is a collection of commutative update-receive operations then for any state S, $S \circ \mathcal{O}$ denotes the state obtained by applying these operations to S in any order.

If u is an $\mathsf{add}(x)$ or a $\mathsf{delete}(x)$ operation and $u \in \mathsf{Past}(S)$ where S is a reachable state at some replica r, and u' is the corresponding receive operation of u at replica r, then we say $u' \in \mathsf{Past}(S)$. Thus, unless otherwise specified, in the proofs that follow, all the update operations in the causal-past of a state refer to the receive operations. Similarly, if u is an $\mathsf{add}(x)$ operation and u' is the corresponding $\mathsf{addrcv}(m)$ operation, when we write $\mathsf{CoveringDel}(u')$, we mean $\mathsf{CoveringDel}(u)$, which is the set of the covering deletes of the add operation u.

For an $\mathsf{add}(x)$ operation u if the triple prepared by the ADD.PREPAREx is m, and if u' is the corresponding $\mathsf{addrcv}(m)$ operation, then, we write $\arg(u')$ to mean m which is $Aux(u)$.

Similarly, for an $\mathsf{delete}(x)$ operation u if the version vector prepared by the

$$\textsc{delete.prepare}(x)$$

is M, and if u' is the corresponding $\mathsf{delrcv}(M)$ operation, then, we write $\arg(u')$ to mean M which is $Aux(u)$.

We first prove that the addrcv and delrcv operations can be simulated using the merge operation with appropriate arguments.

Proposition 33. *Let S be a state of some replica. Let $u \in \{\mathsf{addrcv}, \mathsf{delrcv}\}$ be a receive operation. Let $a = \arg(u)$ be the argument of u.*

$$\textit{If } S_{op} = S \circ u(a) \textit{ then } S_{op} \equiv S \circ \mathsf{merge}(S_\perp \circ u(a))$$

Proof. Let $S_{one} = S_\perp \circ u(a)$ and $S_{merge} = S \circ \mathsf{merge}(S_{one})$. We have to consider the following two cases:

Case $u(a) = \mathsf{addrcv}(m)$: In this case, from the code for $\textsc{add.apply}(m)$, it is seen that

$$S_{op}.E = \begin{cases} S.E \cup \{m\} & \text{if } ts(m) \notin S.V[rep(m)] \\ S.E & \text{otherwise} \end{cases}$$

Also,

$$S_{op}.V[r] = \begin{cases} S.V[r] \cup \{ts(m)\} & \text{if } r = rep(m) \text{ and } ts(m) \notin S.V[r] \\ S.V[r] & \text{otherwise} \end{cases}$$

Simplifying, we get

$$S_{op}.V[r] = \begin{cases} \mathbf{add}(S.V[r], \{ts(m)\}) & \text{if } r = rep(m) \\ S.V[r] & \text{otherwise} \end{cases}$$

From the code of $\textsc{add.apply}$, it is immediately seen that $S_{one}.E = \{m\}$,

$$S_{one}.V[r] = \begin{cases} \{[ts(m), ts(m)]\} & \text{if } r = rep(m) \\ \emptyset & \text{otherwise} \end{cases}$$

Again from the code of $\textsc{merge}$, it follows that for any triple m', $m' \in S_{merge}.E$ iff

$$(m' \neq m \text{ and } m' \in S.E) \text{ or } \{m' = m \text{ and } (m \in S.E \text{ or } ts(m) \notin S.V[rep(m)])\}.$$

Simplifying, we see that $m' \in S_{merge}.E$ iff

$$m' \in S.E \text{ or } (m' = m \text{ and } ts(m) \notin S.V[rep(m)]).$$

Thus it follows that $S_{merge}.E = S_{op}.E$.

Again from the code of MERGE, we see that

$$S_{merge}.V[r] = \begin{cases} \mathbf{add}(S.V[r], \{ts(m)\}) & \text{if } r = rep(m) \\ S.V[r] & \text{otherwise} \end{cases}$$

Thus $S_{merge}.V = S_{op}.V$. Therefore $S_{op} \equiv S_{merge}$ in this case.

Case $u(a) = \mathsf{delrcv}(V')$: From the code, V' is the interval version vectors containing timestamps of triples m that have to be deleted. Now, it is easy to see that

$$S_{one}.E = \emptyset$$

and

$$S_{one}.V = V'$$

From the code for the merge, for any r',

$$S_{merge}.V[r'] = S.V[r'] \cup S_{one}.V[r']$$

.

Similarly from the code for delrcv,

$$S_{op}.V[r'] = S.V[r'] \cup V'[r']$$

Hence $S_{merge}.V = S_{op}.V$.

Since $S_{one}.E = \emptyset$,

$$\begin{aligned} S_{merge}.E &= S.E \setminus \{m' \mid ts(m') \in S_{one}.V[rep(m')]\} \\ &= S.E \setminus \{m' \mid ts(m') \in V'[rep(m')]\} \\ &= S_{op}.E \end{aligned}$$

From this and the fact that $S_{op}.V = S_{merge}.V$, we can conclude that $S_{op} \equiv S_{merge}$ in this case.

□

Any reachable state of the OR-Set is obtained by applying some sequence of addrcv, delrcv and merge operations to the initial state $S_\perp$. From Proposition 33, since any addrcv or delrcv operation can be simulated using the merge operation, the structure of the reachable states can be reasoned about using the properties of the merge operations. In the following result, we prove two important properties of merges namely *associativity* and *commutativity*. In the subsequent result we prove the *idempotence* of merges.

Proposition 34. *If S_1, S_2, S_3 are three reachable states of some run then*

$$(S_1 \circ \mathsf{merge}(S_2)) \circ \mathsf{merge}(S_3) \equiv S_1 \circ \mathsf{merge}(S_2 \circ \mathsf{merge}(S_3)) \equiv (S_1 \circ \mathsf{merge}(S_3)) \circ \mathsf{merge}(S_2)$$

Proof. Let

- $S_{12} = S_1 \circ \mathsf{merge}(S_2)$, $S_{23} = S_2 \circ \mathsf{merge}(S_3)$ and $S_{13} = S_1 \circ \mathsf{merge}(S_3)$,
- $S_{(12)3} = S_{12} \circ \mathsf{merge}(S_3)$, $S_{1(23)} = S_1 \circ \mathsf{merge}(S_{23})$ and $S_{(13)2} = S_{13} \circ \mathsf{merge}(S_2)$.

For any r, $S_{(12)3}.V[r] = S_{1(23)}.V[r] = S_{1(32)}.V[r] = S_1.V[r] \cup S_2.V[r] \cup S_3.V[r]$.

For any distinct $i, j \in \{1, 2, 3\}$ one can observe from the code of merge that for any triple m, $m \in S_{ij}.E$ iff $(m \in S_i.E \cap S_j.E) \vee (m \in S_i.E \wedge ts(m) \notin S_j.V[rep(m)]) \vee (m \in S_j.E \wedge ts(m) \notin S_i.V[rep(m)])$. Using repeated application of this principle, we can show that $m \in S_{(12)3}.E$ iff one of the following is satisfied:

- $(m \in S_1.E \cap S_2.E \cap S_3.E)$
- $\bigvee_{i,j,k \in \{1,2,3\} \wedge i \neq j \neq k} (m \in (S_i.E \cap S_j.E) \wedge ts(m) \notin S_k.V[rep(m)])$
- $\bigvee_{i,j,k \in \{1,2,3\} \wedge i \neq j \neq k} (m \in S_i.E \wedge ts(m) \notin (S_j.V[rep(m)] \cup S_k.V[rep(m)]))$.

By symmetry, it can be seen that these are also the conditions for m to belong to $S_{1(23)}.E$ or $S_{1(32)}.E$. Thus we have $S_{(12)3}.E = S_{1(23)}.E = S_{1(32)}.E$.

Hence $S_{(12)3} \equiv S_{1(23)} \equiv S_{1(32)}$. □

We now prove that merges are idempotent.

Proposition 35. *If S_1 and S_2 are two reachable states then*

$$(S_1 \circ \mathsf{merge}(S_2)) \circ \mathsf{merge}(S_2) \equiv S_1 \circ \mathsf{merge}(S_2)$$

Proof. Let $S = S_1 \circ \mathsf{merge}(S_2)$ and $S' = S \circ \mathsf{merge}(S_2)$. For any r, $S'.V[r] = S.V[r] \cup S_2.V[r] = (S_1.V[r] \cup S_2.V[r]) \cup S_2.V[r] = S_1.V[r] \cup S_2.V[r] = S.V[r]$.

For any element m, $m \in S'.E$ iff $m \in S.E \cap S_2.E$ or $m \in S.E \wedge ts(m) \notin S_2.V[rep(m)]$ or $m \in S_2.E \wedge m \notin S.V[rep(m)]$. Again, we have $m \in S.E$ iff $m \in S_1.E \cap S_2.E$ or $m \in S_1.E \wedge ts(m) \notin S_2.V[rep(m)]$ or $m \in S_2.E \wedge ts(m) \notin S_1.V[rep(m)]$.

On combining and simplifying these two conditions we can see that $m \in S'.E$ iff $m \in S_1.E \cap S_2.E$ or $m \in S_1.E \wedge ts(m) \notin S_2.V[rep(m)]$ or $m \in S_2.E \wedge ts(m) \notin S_1.V[rep(m))$. This happens iff $m \in S.E$.

Thus $S \equiv S'$ and hence $(S_1 \circ \mathsf{merge}(S_2)) \circ \mathsf{merge}(S_2) \equiv S_1 \circ \mathsf{merge}(S_2)$. □

Having proved that the merge operations are associative and idempotent, we now use these results to prove that when a pair of replicas having the same state which apply the same set of update-receive operations, their resultant state would be the same, irrespective of the order in which they applied those update-receive operations.

Lemma 36. *Let S be some reachable state of the* OR-Set, $\mathcal{O} = \{u_1, u_2, \ldots u_n\}$ be a set of receive operations along with their arguments. Let π_1 and π_2 be any two permutations of $[1 \cdots n]$. If $S_1 = S \circ u_{\pi_1(1)} \circ u_{\pi_1(2)} \ldots u_{\pi_1(n)}$ and $S_2 = S \circ u_{\pi_2(1)} \circ u_{\pi_2(3)} \ldots u_{\pi_2(n)}$ then $S_1 \equiv S_2$.

Proof. We prove the result by induction on $|\mathcal{O}|$. If $|\mathcal{O}| = 1$, the result follows trivially. Assume that the result holds for all $\mathcal{O}$ of size smaller than n. Now consider $\mathcal{O} = \{u_1, u_2, \ldots, u_n\}$. Let π_1 and π_2 be any two permutations of $[1 \ldots n]$. Let $j \in [1 \ldots n]$ such that $\pi_1(j) = \pi_2(n)$.

$$
\begin{aligned}
S_1 &= (S \circ u_{\pi_1(1)} \circ \cdots \circ u_{\pi_1(n-1)}) \circ u_{\pi_1(n)} \\
&= (S \circ u_{\pi_1(1)} \circ \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j)} \circ u_{\pi_1(j+1)} \cdots \circ u_{\pi_1(n-1)}) \circ u_{\pi_1(n)} \\
&\equiv (S \circ u_{\pi_1(1)} \circ \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j+1)} \cdots u_{\pi_1(n-1)} \circ u_{\pi_1(j)}) \circ u_{\pi_1(n)} \\
&\qquad\qquad \text{(from induction hypothesis)} \\
&\equiv (S \circ u_{\pi_1(1)} \circ \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j+1)} \cdots u_{\pi_1(n-1)}) \circ u_{\pi_1(j)} \circ u_{\pi_1(n)} \\
&\equiv (S \circ u_{\pi_1(1)} \circ \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j+1)} \cdots u_{\pi_1(n-1)}) \circ \mathsf{merge}(S_\perp \circ u_{\pi_1(j)}) \circ \mathsf{merge}(S_\perp \circ u_{\pi_1(n)}) \\
&\qquad\qquad \text{(from Proposition 33)} \\
&\equiv (S \circ u_{\pi_1(1)} \circ \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j+1)} \cdots u_{\pi_1(n-1)}) \circ \mathsf{merge}(S_\perp \circ u_{\pi_1(n)}) \circ \mathsf{merge}(S_\perp \circ u_{\pi_1(j)}) \\
&\qquad\qquad \text{(merges commute from Proposition 34)} \\
&\equiv (S \circ u_{\pi_1(1)} \circ \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j+1)} \cdots u_{\pi_1(n-1)} \circ u_{\pi_1(n)}) \circ u_{\pi_1(j)} \\
&\qquad\qquad \text{(from Proposition 33)}
\end{aligned}
$$

Now, since

$$\{\pi_1(1), \ldots, \pi_1(j-1), \pi_1(j+1), \ldots \pi_1(n)\} = \{\pi_2(1), \ldots, \pi_2(n-1)\}$$

by the induction hypothesis, we have,

$$S \circ u_{\pi_1(1)} \circ \cdots u_{\pi_1(j-1)} \circ u_{\pi_1(j+1)} \cdots u_{\pi_1(n-1)} \circ u_{\pi_1(n)} \equiv S \circ u_{\pi_2(1)} \circ u_{\pi_2(2)} \cdots u_{\pi_2(n-1)}$$

Further since $\pi_1(j) = \pi_2(n)$, we have

$$
\begin{aligned}
S_1 &\equiv (S \circ u_{\pi_2(1)} \circ u_{\pi_2(2)} \cdots u_{\pi_2(n-1)}) \circ u_{\pi_2(n)} \\
&= S_2
\end{aligned}
$$

□

The following lemma shows the relationship between any reachable state and its causal past. From this result we can also conclude that any two reachable states with the same causal history are identical.

Before that, we make the following observation.

Let S' and S'' be any two equivalent states. i.e $S' \equiv S''$. Then, for any state S, $S \circ \mathsf{merge}(S') = S \circ \mathsf{merge}(S'')$. The reason for this is that merge only updates the E set and the version vector V. It does not modify the $S.count$ in any way in either case.

Lemma 37. *Let S be any reachable state of replica. Then, we can write $S \equiv S_\perp \circ \mathsf{Past}(S)$.*

Proof. Let S be a reachable state of some replica at the end of a run of OR-Set

The proof follows by induction over the length of the run.

If S is the state at the end of an empty run, $\mathsf{Past}(S) = \emptyset$ then by definition, $S = S_\perp$ and the result follows.

Suppose the result is true for all runs of length smaller than k. Let S be the state of some replica at the end of k^{th} operation.

Let the receive operations associated with $\mathsf{Past}(S) = \{u_1, u_2, \ldots, u_n\}$.

Since S is a reachable state of some replica, we can find a reachable state $S' \neq S$ of the same replica and an operation u with $Op(u) \in \{\mathsf{addrcv}, \mathsf{delrcv}, \mathsf{merge}\}$ such that $S = S' \circ u$. By definition, S' is a reachable state in a run of length smaller than k. Hence, by the induction hypothesis, $S' \equiv S_\perp \circ \mathsf{Past}(S')$.

Consider the case when $u = \mathsf{merge}(S'')$ where $S'' \neq S_\perp$ and $S'' \neq S$. By the induction hypothesis $S'' \equiv S_\perp \circ \mathsf{Past}(S'')$.

Let the receive operations associated with $\mathsf{Past}(S'')$ be $\{u''_1, u''_2, \ldots, u''_l\}$. Now $S = S' \circ \mathsf{merge}(S'')$. From the observation earlier, this can be written as

$$S = S' \circ \mathsf{merge}(S_\perp \circ u''_1 \circ u''_2 \cdots u''_m)$$

By appealing to Propositions 33 and 34, this is the same as

$$S \equiv S' \circ \mathsf{merge}(S_\perp \circ u''_1) \circ \mathsf{merge}(S_\perp \circ u''_2) \cdots \mathsf{merge}(S_\perp \circ u''_l) \circ \mathsf{merge}(S_\perp)$$

.

Appealing to Proposition 33 once again, we can rewrite this as

$$S \equiv S' \circ u''_1 \circ u''_2 \cdots u''_l . \circ \mathsf{merge}(S_\perp)$$

Since for any state S''', $S''' \circ \mathsf{merge}(S_\perp) \equiv S_\perp \circ \mathsf{merge}(S''') \equiv S'''$, we have

$$S \equiv S' \circ u''_1 \circ u''_2 \circ \cdots u''_l$$

By induction hypothesis, we can write this as

$$\begin{aligned} S &\equiv S_\perp \circ \mathsf{Past}(S') \circ u''_1 \circ u''_2 \cdots u''_l \\ &\equiv S_\perp \circ \mathsf{Past}(S') \circ \mathsf{Past}(S'') \\ &\equiv S_\perp \circ (\mathsf{Past}(S') \cup \mathsf{Past}(S'')) \\ &= S_\perp \circ \mathsf{Past}(S) \end{aligned}$$

hus, in this case $S \equiv S_\perp \circ \mathsf{Past}(S)$.

If u were an addrcv or a delrcv operation, appealing to Proposition 33, we can reduce it to the merge case. □

We now analyse the structure of the state of the optimized implementation. In particular, we state the necessary and sufficient conditions for a certain integer to be present in the integer version vector of the state.

Proposition 38. *Let S be any reachable state. $c \in S.V[r]$ iff there exists a receive operation u of an update operation $u' \in \mathsf{Past}(S)$ such that*

- *Either u is an $\mathsf{addrcv}(m)$ with $ts(m) = c$ and $rep(m) = r$ or*
- *u is a $\mathsf{delrcv}(V')$ operation with $c \in V'[r]$.*

Proof. Proof follows by induction over $|\mathsf{Past}(S)|$. The base case when $S = S_\perp$ is trivial. If $|\mathsf{Past}(S)| = 1$, then from Lemma 37, $S = S_\perp.u$. If u is an $\mathsf{addrcv}(m)$ such that $ts(m) = c$ and $rep(m) = r$, then $c' \in S.V[r']$ iff $(c' = c)$ and $(r' = r)$. If $u = \mathsf{delrcv}(V')$ then, $c \in S.V[r]$ iff $c \in V'[r]$. Thus the result is true for all S with $|\mathsf{Past}(S)| = 1$. Assume that the result holds for all states S with $|\mathsf{Past}(S)| < n$. Now consider a state S such that $|\mathsf{Past}(S)| = n$.

From proposition 33 we can write $S \equiv S' \circ \mathsf{merge}(S'')$ with $\mathsf{Past}(S') \subsetneq \mathsf{Past}(S)$ and $\mathsf{Past}(S'') \subsetneq \mathsf{Past}(S)$ for appropriate S' and S''. Now $c \in S.V[r]$ iff $c \in S'.V[r] \cup S''.V[r]$. Since $|\mathsf{Past}(S')| < n$ and $|\mathsf{Past}(S'')| < n$, by induction hypothesis, this happens iff there exists such an receive operation u of an update $u' \in \mathsf{Past}(S') \cup \mathsf{Past}(S'')$ such that u is an $\mathsf{addrcv}(m)$ for which $ts(m) = c$ and $rep(m) = r$ or u is a $\mathsf{delrcv}(V')$ operation with $c \in V'[r]$. By definition, $\mathsf{Past}(S) = \mathsf{Past}(S') \cup \mathsf{Past}(S'')$. Hence $u' \in \mathsf{Past}(S)$ and u is the corresponding receive operation. □

We now establish the necessary and sufficient conditions for a $\mathsf{delrcv}()$ operation to be a *covering-delete* of an $\mathsf{addrcv}()$ operation by inspecting their respective arguments. This result along with Proposition 38 yields the necessary and sufficient condition characterising the structure of the Interval Version Vectors of any reachable state through the *covering-delete* relations between the $\mathsf{addrcv}()$ and $\mathsf{delrcv}()$ operations present in the state-history.

Proposition 39. *If u is an $\mathsf{addrcv}(m)$ operation and u' is a $\mathsf{delrcv}(V')$ operation then $u' \in \mathsf{CoveringDel}(u)$ iff $ts(m) \in V'[rep(m)]$.*

Proof. Suppose $data(m) = x$. Let p and p' be the corresponding ADD.PREPARE() and DELETE.PREPARE() methods of u and u' respectively. Let r' be the source replica of the delete whose prepare and receive operations are (p', u'). Let S be the state of r' before applying u and let S' be the state of r' before applying p'. Let $\rightarrow$ be a total order on the set of all the update-receive operations of OR-Set such that $\xrightarrow{\mathsf{hb}} \subseteq \rightarrow$. Let u' be the i^{th} x-delrcv operation in $\rightarrow$. The proof follows from induction over i.

The base case occurs when $i = 1$. Suppose $ts(m) \in V'[rep(m)]$. Since V' is prepared by p', from the code of DELETE.PREPARE(), we know that $m \in S'.E$. This is possible only if $u \in \mathsf{Past}(S')$, as every addrcv() operation has a unique argument. Hence $u \xrightarrow{\mathsf{hb}} u'$. Since u is the earliest x-delrcv operation in $\rightarrow$ and since $\rightarrow$ is consistent with $\xrightarrow{\mathsf{hb}}$, there is no other x-delrcv operation u'' such that $u \xrightarrow{\mathsf{hb}} u'' \xrightarrow{\mathsf{hb}} u'$. Hence by definition, $u' \in \mathsf{CoveringDel}(u)$. Conversely suppose $u' \in \mathsf{CoveringDel}(u)$. Then since $\mathsf{Past}(S')$ contains u and cannot not contain any x-delrcv() operations without contradicting the minimality of u' in $\rightarrow$, we can conclude that $m \in S'.E$. From the code of DELETE.PREPARE(), $ts(m) \in V'[rep(m)]$.

Assume that the result holds for all $i < n$. Suppose u' is the n^{th} x-delrcv operation in $\rightarrow$. If $ts(m) \in V'[rep(m)]$ then we know that $m \in S'.E$ which implies that $u \in \mathsf{Past}(S')$ and hence $u \xrightarrow{\mathsf{hb}} u'$. If $u' \notin \mathsf{CoveringDel}(u)$ then there exists an x-delrcv operation u'' such that $u \xrightarrow{\mathsf{hb}} u'' \xrightarrow{\mathsf{hb}} u$ and $u'' \in \mathsf{CoveringDel}(u)$. Since $\rightarrow$ is consistent with $\xrightarrow{\mathsf{hb}}$, u'' occurs before u' in $\rightarrow$. Let $V'' = \arg(u'')$. By induction hypothesis, $ts(m) \in V''[rep(m)]$. If S'' is the state of r' after applying u'' then, from the code of delrcv, we know that $m \notin S''.E$. No addrcv or merge operation applied by the replica to attain a state S' from S'' can reintroduce m into the E set of the replica since $ts(m) \in S''.V[rep(m)]$ (from proposition 38). Hence $m \notin S'.E$ which implies that $ts(m) \notin V'[rep(m)]$ which is a contradiction. Hence $u' \in \mathsf{CoveringDel}(u)$.

Conversely suppose $u' \in \mathsf{CoveringDel}(u)$. Suppose there is an x-delrcv operation $u'' \in \mathsf{Past}(S')$ such that $\arg(u'') = V''$ and $ts(m) \in V''[rep(m)]$. Since $u'' \xrightarrow{\mathsf{hb}} u'$, u'' appears before u' in the total order $\rightarrow$. By induction hypothesis, $u'' \in \mathsf{CoveringDel}(u)$ which implies $u \xrightarrow{\mathsf{hb}} u'' \xrightarrow{\mathsf{hb}} u'$ which contradicts the fact that $u' \in \mathsf{CoveringDel}(u)$. Hence no such u'' exists. Since $u \in \mathsf{Past}(S')$, $m \in S'.E$. From the code of DELETE.PREPARE(), m is in the argument set prepared by p'. Hence $ts(m) \in V'[rep(m)]$. □

Once a replica has applied a delrcv operation, any subsequent addrcv operation for which the earlier delrcv operation was a *covering-delete* has no effect on the state of the replica. We formally prove this through the following proposition.

Proposition 40. *Let S be a reachable state, u be a* addrcv *operation and u' a* delrcv *operation such that $u' \in \mathsf{CoveringDel}(u)$. Let S be any reachable state with $\{u, u'\} \cap \mathsf{Past}(S) = \emptyset$.*

Then $S \circ u' \circ u \equiv S \circ u'$.

Proof. Let $S' = S \circ u'$. We need to show that $S' \circ u \equiv S'$. Let $m = \arg(u), V' = \arg(u')$. From proposition 39, $ts(m) \in V'[rep(m)]$. From proposition 38, $ts(m) \in S'.V[rep(m)]$.

Hence from the code of addrcv, the operation u will not affect any change in the state. Hence $S' \circ u \equiv S'$. □

We now prove the necessary and sufficient conditions characterising the structure of the E set of any reachable state. With this result we are closer to showing that the optimized OR-Set implementation satisfies the concurrent-specification.

Lemma 41. *Let S be any state and u be an x-*addrcv *operation such that $u \notin \mathsf{Past}(S)$ and $\arg(u) = m$, and let $S' = S \circ u$. Then $m \in S'.E$ iff $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$.*

Proof. We first note that since if $\arg(u) = m$, and $u \notin \mathsf{Past}(S)$, then $m \notin S.E$. This is because, the auxiliary information m is unique to u and thus, since $S = S_\perp \circ \mathsf{Past}(S)$, no other add or addrcv operation from $\mathsf{Past}(S)$ could have added m into $S.E$. Thus $m \notin S.E$.

Suppose $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$. Suppose u' is some covering-delete of u, i.e. $u' \in \mathsf{CoveringDel}(u)$. Let $V' = \arg(u')$. From Proposition 39, it is the case that $ts(m) \in V'[rep(m)]$. Since it is given that $u \notin \mathsf{Past}(S)$ and since $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$, from proposition 38, we know that $ts(m) \notin S.V[rep(m)]$. Since $S' = S \circ u$, from the code of $\mathsf{addrcv}(m)$, we can conclude that $m \in S'.E$.

Thus if $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$ then $m \in S'.E$.

Conversely, suppose $m \in S'.E$. We need to show that $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$. Suppose not. Let $u' \in \mathsf{CoveringDel}(u) \cap \mathsf{Past}(S)$.

Let $\mathcal{O} = \mathsf{Past}(S) \setminus \{u'\}$. If $S_{old} \equiv S_\perp \circ \mathcal{O}$, then it is clear that $S \equiv S_{old} \circ u'$.

Now it is given that $S' = S \circ u$ which is to say, $S' \equiv S_{old} \circ u' \circ u$. But from Proposition 40 we know that $S_{old} \circ u' \circ u \equiv S_{old} \circ u' \equiv S$. Thus $S' \equiv S$. We have shown above that $m \notin S.E$. Since $S' \equiv S$, it implies that $m \notin S'.E$. But this contradicts our premise that $m \in S'.E$. The contradiction is due to our assumption that there exists a u' such that $u' \in \mathsf{CoveringDel}(u) \cap \mathsf{Past}(S)$. Hence no such u' exists, which implies $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$.

Thus we have shown that if $m \in S'.E$ then, $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$.

From these two cases, we can conclude that

$$m \in S'.E \iff \mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$$

.

□

Theorem 42. *The optimized OR-set implementation satisfies the concurrent specification for* OR-Sets provided in Definition 26

Proof. Given a state S and an element x, let $\mathcal{O}_{add}$ be the set of all x-add-receive operations u in $\mathsf{Past}(S)$ such that $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$. Let $\mathcal{O}_{others} = \mathsf{Past}(S) \setminus \mathcal{O}_{add}$ and $S' \equiv S_\perp \circ \mathcal{O}_{others}$. Then, $S \equiv S' \circ \mathcal{O}_{add}$.

Since $\mathcal{O}_{others}$ contains no x-addrcv operation u such that $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$, from Lemma 41, we can conclude that for all $m \in S'.E$, $data(m) \neq x$. Hence $S' \circ \mathsf{contains}(x) = \mathsf{False}$. Now $S \circ \mathsf{contains}(x) = \mathsf{True}$ iff $\exists m \in S.E$ such that $data(m) = x$. Since $S \equiv S' \circ \mathcal{O}_{add}$, from Lemma 41, this is possible iff there exists an x-addrcv operation $u \in \mathcal{O}_{add}$ such that $\arg(u) = m$ and $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S) = \emptyset$ iff $\mathcal{O}_{add} \neq \emptyset$. □

Thus, we have shown that our generic optimized implementation is correct with respect to the distributed specification.

We now prove that the implementation is a CRDT. In particular, it is both a CvRDT and a CmRDT. Towards this, we define the notion of *Seen*() and *Deletes*() which capture which are the elements that have been observed in a state, and which elements have been deleted.

Definition 43 (Seen and Delete for a state)**.** *Let S be a reachable state of the optimized implementation.*

We define the set of timestamps of all the elements added and deleted in a state S, denoted by $Seen(S)$, as

$$Seen(S) = \{(c, r) \mid c \in S.V[r]\}$$

.

The set of timestamps for all the elements that are currently present in a state S, is denotedy by $Exists(S)$. This is defined to be

$$Exists(S) = \{(c, r) \mid \exists x : (x, c, r) \in S.E\}$$

Similarly, we denote the set of all the timestamps of elements deleted in a state S as $Deletes(S)$ defined as

$$Deletes(S) = Seen(S) \setminus Exists(S)$$

.

We then define a binary-relation that allows us to compare a pair of states of the optimized implementation.

Definition 44 (compare relation)**.** *For states S, S' we say $S \leq_{compare} S'$ to mean that* $\mathsf{compare}(S, S')$ *returns* True*.*

To show that the optimized OR-Set implementation is a *CvRDT* we need to define a partial order on the set of reachable states of the implementation.

We can observe that for states S_1 and S_2, if $Seen(S_1) = Seen(S_2)$ and $Deletes(S_1) = Deletes(S_2)$ then $S_1.E = S_2.E$ and $S_1.V = S_2.V$. Thus the $(Seen(S), Deletes(S))$ pair uniquely identifies a state S for all query-purposes. We use this definition to show that there exists a well-defined partial order on the set of states of the optimized OR-Set implementation through the following Proposition.

Proposition 45. *For states S_1 and S_2 , $S_1 \leq_{compares} S_2$ iff $Seen(S_1) \subseteq Seen(S_2)$ and $Deletes(S_1) \subseteq Deletes(S_2)$. $\leq_{compare}$ induces a partial order on $\mathcal{S}$.*

Proof. ($\Longrightarrow$)Suppose $S_1 \leq_{compares} S_2$.

From the code, $\forall r' : S_1.V[r'] \subseteq S_r.V[r']$. Now,

$$\begin{aligned} \forall r' : S_1.V[r'] \subseteq S_r.V[r'] &\iff \{(c', r') \mid c \in S_1.V[r']\} \subseteq \{(c', r') \mid c \in S_2.V[r']\} \\ &\iff Seen(S_1) \subseteq Seen(S_2) \end{aligned}$$

Let m be a triple such that $ts(m) = c$ and $rep(m) = r$. Suppose $(c, r) \in Deletes(S_1)$. Then, by definitions $(c, r) \in Seen(S_1)$ and $(c, r) \notin Exists(S_1)$. Thus by definition, $m \notin S_1.E$.

We need to show that $(c, r) \in Deletes(S_2)$. Suppose that is not the case. Since $(c, r) \in Seen(S_1)$ and since $Seen(S_1) \subseteq Seen(S_2)$, we know that $(c, r) \in Seen(S_2)$. Since we are assuming that $(c, r) \notin Deletes(S_2)$, it implies that $(c, r) \in Exists(S_2)$. Thus, by definition of $Exists()$, $m \in S_2.E$.

Thus, $m \notin S_1.E.$ and $m \in S_2.E$. Hence $m \in S_2.E \setminus S_1.E$. Since we are considering the case where $S_1 \leq_{compare} S_2$, from the code of COMPARE,

$$m \in S_2.E \setminus S_1.E \implies c \notin S_1.V[r]$$

which is a contradicts the fact that $(c, r) \in Seen(S_1)$. Hence our assumption that $(c, r) \notin Deletes(S_2)$ is incorrect. Thus, if $(c, r) \in Deletes(S_1)$ then $(c, r) \in Deletes(S_2)$. Hence $Deletes(S_1) \subseteq Deletes(S_2)$

($\Longleftarrow$) Conversely, suppose $Seen(S_1) \subseteq Seen(S_2)$ and $Deletes(S_1) \subseteq Deletes(S_2)$.

By definition of $Seen()$, for all r', $S_1.V[r'] \subseteq S_2.V[r']$.

Let m be a triple such that $m \in S_2.E \setminus S_1.E$. Thus $(c, r) \in Exists(S_2)$ and $(c, r) \notin Exists(S_1)$. Since for any state S, $Exists(S) = Seen(S) \setminus Deletes(S)$, this implies that $(c, r) \in Seen(S_2)$ and $(c, r) \notin Deletes(S_2)$. Furthermore, $(c, r) \notin Seen(S_1)$ or $(c, r) \in Deletes(S_1)$. Since $Deletes(S_1) \subseteq Deletes(S_2)$, it is clear that if $(c, r) \notin Deletes(S_2)$, then $(c, r) \notin Deletes(S_1)$. Thus, we have $(c, r) \in Seen(_)S_2$ and $(c, r) \notin Seen(S_1)$ which implies that $c \notin S_1.V[r]$. Since $c = ts(m)$ and $r = rep(m)$, we have

$$m \in S_2.E \setminus S_1.E \implies ts(m) \notin S_1.V[rep(m)]$$

From this and the fact that

$$\forall r' : S_1.V[r'] \subseteq S_2.V[r']$$

, we can conclude that $S_1 \leq_{compare} S_2$.

For any S, $S \leq_{compare} S$. For S_1 and S_2, $S_1 \leq_{compare} S_2$ and $S_2 \leq_{compare} S_1$ implies that $Seen(S_1) = Seen(S_2)$ and $Deletes(S_1) = Deletes(S_2)$. This implies that $S_1 \equiv S_2$.

Finally $S_1 \leq_{compare} S_2$ and $S_2 \leq_{compare} S_3$ implies $Seen(S_1) \subseteq Seen(S_2) \subseteq Seen(S_3)$ and $Deletes(S_1) \subseteq Deletes(S_2) \subseteq Deletes(S_3)$. Hence from the previous part $S_1 \leq_{compare} S_3$. Thus $\leq_{compare}$ is a partial order on the set of states $\mathcal{S}$. □

We now show that the state computed merge operation is the upper bound of the two states in the partial order defined by $\leq_{compare}$.

Proposition 46. *For states S_1, S_2 and S_3 we have $S_3 = S_1 \circ \mathsf{merge}(S_2)$ iff $Seen(S_3) = Seen(S_1) \cup Seen(S_2)$ and $Deletes(S_3) = Deletes(S_1) \cup Deletes(S_2)$.*

Proof. ($\Longrightarrow$) : Suppose $S_3 = S_1 \circ \mathsf{merge}(S_2)$.

From the code, for any r', $S_3.V[r'] = S_1.V[r'] \cup S_2.V[r']$. Hence, by definition, $Seen(S_3) = Seen(S_1) \cup Seen(S_2)$.

Let m be a triple with $ts(m) = c$ and $rep(m) = r$.

By definition of $Deletes()$,

$$\begin{aligned}
(c,r) \in Deletes(S_3) &\iff (c,r) \in Seen(S_3) \wedge (c,r) \notin Exists(S_3) \\
&\iff (c,r) \in Seen(S_3) \wedge m \notin S_3.E \\
&\iff (c,r) \in Seen(S_3) \wedge m \notin S_3.E \\
&\iff (c,r) \in Seen(S_3) \wedge [m \notin (S_1.E \cup S_2.E) \vee \\
&\qquad\qquad (m \notin (S_1.E \cap S_2.E) \wedge c \in (S_1.V[r] \cap S_2.V[r]))] \\
&\qquad\qquad \text{(From the code of MERGE)}
\end{aligned}$$

We will consider the $(m \notin (S_1.E \cup S_2.E))$ case and $(m \notin (S_1.E \cap S_2.E) \wedge c \in (S_1.V[r] \cap S_2.V[r]))$ case sepearately.

Case : $m \notin S_1.E \cup S_2.E$: Then from the above it is clear that

$$\begin{aligned}
(c,r) \in Deletes(S_3) &\iff (c,r) \in Seen(S_3) \wedge m \notin S_1.E \wedge (c,r) m S_2.E \\
&\iff (c,r) \in Seen(S_3) \wedge (c,r) \notin Exists(S_1) \wedge (c,r) \notin Exists(S_2) \\
&\iff (c,r) \in Seen(S_1) \vee (c,r) \in Seen(S_2) \\
&\qquad\qquad \wedge (c,r) \notin Exists(S_1) \wedge (c,r) \notin Exists(S_2) \\
&\implies (c,r) \in Seen(S_1) \setminus Exists(S_1) \vee (c,r) \in Seen(S_2) \setminus Exists(S_2) \\
&\implies (c,r) \in Deletes(S_1) \vee (c,r) \in Deletes(S_2) \\
&\implies (c,r) \in Deletes(S_1) \cup Deletes(S_2)
\end{aligned}$$

Case : $m \in S_1.E \cup S_2.E$: Without loss of generality, let us assume that $m \in S_1.E$. Thus, $(c,r) \in Exists(S_1)$ and by definition, $(c,r) \in Seen(S_1)$.

From the equivalence before the case analysis, it is clear that

$$\begin{aligned}
(c,r) \in \mathit{Deletes}(S_3) &\iff (c,r) \in \mathit{Seen}(S_3) \wedge (m \notin S_1.E \wedge S_2.E) \wedge \\
&\qquad\qquad c \in (S_1.V[r] \cap S_2.V[r])) \\
&\iff (c,r) \in \mathit{Seen}(S_3) \wedge (m \notin S_2.E) \wedge c \in (S_1.V[r] \cap S_2.V[r]) \\
&\quad \text{(Since } m \in S_1.E\text{)} \\
&\implies m \notin S_2.E \wedge c \in S_2.V[r] \\
&\implies (c,r) \notin \mathit{Exists}(S_2) \wedge (c,r) \in \mathit{Seen}(S_2) \\
&\implies (c,r) \in \mathit{Deletes}(S_2) \\
&\implies (c,r) \in \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)
\end{aligned}$$

Thus in both these cases, $\mathit{Deletes}(S_3) \subseteq \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$.

Conversely, suppose $(c,r) \in \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$. Without loss of generality let us assume $(c,r) \in \mathit{Deletes}(S_1)$. Thus $(c,r) \in \mathit{Seen}(S_1)$ and $(c,r) \notin \mathit{Exists}(S_1)$. We need to show that $(c,r) \in \mathit{Deletes}(S_3)$. Suppose not. Then, either $(c,r) \notin \mathit{Seen}(S_3)$ or $(c,r) \in \mathit{Exists}(S_3)$. It cannot be the case that $(c,r) \notin S_3$ since $(c,r) \in \mathit{Seen}(S_1)$ and $\mathit{Seen}(S_1) \subseteq \mathit{Seen}(S_3)$. Thus $(c,r) \in \mathit{Exists}(S_3)$. Thus in this case, we can say that

$$\begin{aligned}
(c,r) \notin \mathit{Deletes}(S_3) &\iff (c,r) \in \mathit{Exists}(S_3) \\
&\iff m \in S_3.E \\
&\iff m \in S_1.E \cup S_2.E \wedge (m \in S_1.E \cap S_2.E \vee c \notin S_1.V[r] \cap S_2.V[r]) \\
&\qquad \text{(From the code of MERGE)} \\
&\iff m \in S_2.E \wedge (c \notin S_1.V[r] \cap S_2.V[r]) \\
&\qquad \text{(Since } (c,r) \notin \mathit{Exists}(S_1) \implies m \notin S_1.E\text{)} \\
&\iff m \in S_2.E \wedge (c \notin S_2.V[r]) \\
&\qquad \text{(Since } (c,r) \in \mathit{Seen}(S_1) \implies c \in S_1.V[r]\text{)} \\
&\iff \text{A contradiction since } m \in S_2.E \implies c \in S_2.V[r]
\end{aligned}$$

Hence $(c,r) \in \mathit{Deletes}(S_3)$. Thus, $(c,r) \in \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2) \implies (c,r) \in \mathit{Deletes}(S_3)$. Thus, $\mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2) \subseteq \mathit{Deletes}(S_3)$.

Thus, from this and the earlier case analysis, $\mathit{Deletes}(S_3) = \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$

($\Longleftarrow$) Conversely suppose $\mathit{Seen}(S_3) = \mathit{Seen}(S_1) \cup \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_3) = \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$. Let $S_3' = S_1.\mathsf{merge}(S_2)$. From the previous part, $\mathit{Seen}(S_3') = \mathit{Seen}(S_1) \cup \mathit{Seen}(S_2)$ and $\mathit{Deletes}(S_3') = \mathit{Deletes}(S_1) \cup \mathit{Deletes}(S_2)$. Since the code of the merge effects only the E and the V elements and doesn't update the counter c, $(\mathit{Seen}(S), \mathit{Deletes}(S))$ pair uniquely identifies a state. Thus $S_3 = S_3'$. □

Using this result we show that merge operation indeed computes the least-upper-bound.

Proposition 47. *If $S_3 = S_1 \circ \mathsf{merge}(S_2)$ then S_3 is the least upper bound of S_1 and S_2 in the partial order defined by $\leq_{compare}$.*

Proof. From propositions 46 and 45 it is clear that $S_1 \leq_{compare} S_3$ and $S_2 \leq_{compare} S_3$. Hence S_3 is an upper bound of S_1 and S_2. If S_4 is any other upper bound of S_1 and S_2 then by lemma 45, $Seen(S_1) \subseteq Seen(S_4)$ and $Seen(S_2) \subseteq Seen(S_4)$. This implies that $Seen(S_1) \cup Seen(S_2) \subseteq Seen(S_4)$. Similarly, $Deletes(S_1) \cup Deletes(S_2) \subseteq Deletes(S_4)$. Since from lemma 46, we have $Seen(S_1) \cup Seen(S_2) = Seen(S_3)$ and $Deletes(S_1) \cup Deletes(S_2) = Deletes(S_3)$ we can infer that $Seen(S_3) \subseteq Seen(S_4)$ and $Deletes(S_3) \subseteq Deletes(S_4)$. From lemma 45 this implies that $S_3 \leq_{compare} S_4$. Hence S_3 is the least upper bound of S_1 and S_2 in the partial order defined by $\leq_{compare}$. □

Lemma 48. *For states S_1,S_2 and S_3 ,*

1. *$S_1 \leq_{compare} S_2$ iff $Seen(S_1) \subseteq Seen(S_2)$ and $Deletes(S_1) \subseteq Deletes(S_2)$. Therefore $\leq_{compare}$ defines a partial order on $\mathcal{S}$.*

2. *$S_3 = S_1 \circ merge(S_2)$ iff $Seen(S_3) = Seen(S_1) \cup Seen(S_2)$ and $Deletes(S_3) = Deletes(S_1) \cup Deletes(S_2)$ iff S_3 is the least upper bound of S_1 and S_2 in the partial order defined by $\leq_{compare}$.*

Proof. Follows from Propositions 45, 46 and 47. □

From Lemma 36 and Lemma 48, we can show that the optimized OR-Set implementation is a CRDT. In particular it is both CmRDT as well as CvRDT.

Theorem 49. *The optimized* OR-Set *implementation is a CmRDT and a CvRDT.*

Proof. From Lemma 36, we know that all the update-receive operations of OR-Set commute. Furthermore, the OR-Set specification does not have any delivery preconditions. Thus, the implementation is a CmRDT since it satisfies the sufficient condition for a replicated datatype to be a CmRDT.

From Lemma 45, we know that $\leq_{compare}$ induces a partial order on the set of states $\mathcal{S}$ such that $(\mathcal{S}, \leq_{compare})$ is a join-semilattice. From propositions 33 and 46, we know that the states are monotonically non-decreasing across update operations. Finally, from proposition 47 we know that the merge of any pair of states yields the least upper bound of the two states in the join semi-lattice $(\mathcal{S}, \leq_{compare})$. Hence the implementation is a CvRDT. □

3.6 Equivalence of the original and the generic optimised implementations

In this section, we want to show the functional equivalence between the *original implementation* (Algorithm 1) and our *generic optimized implementation* (Algorithm 3). This way, the proof of correctness that we have provided for the generic optimized implementation with respect to the distributed specification will be applicable for the original implementation as well. We achieve this by establishing a strong correspondence between the two implementations in terms of *bisimulation*.

A original state describes the local state of one replica during the execution of Algorithm 1.

Definition 50. *A original state is a tuple* $(E, T, count)$ *where* $E, T \subseteq \mathcal{M}$ *such that* $E \cup T$ *is valid,* $E \cap T = \emptyset$ *and* $count \in \mathbb{N}$. $\mathcal{S}^{original}$ *denotes the set of original states. The initial original state is the state* $(\emptyset, \emptyset, 0)$. *For a valid set* M, *a original state* $(E, T, count)$ *is called* M*-compatible if* $E \cup T \subseteq M$.

An opt state describes the local state of one replica during the execution of Algorithm 3.

Definition 51. *An opt state is a tuple* $(E, V, count)$ *where* $E \subseteq \mathcal{M}$ *is a valid set,* $V : \mathcal{R} \to \mathcal{I}$ *is a vector of interval sequences and* $count \in \mathbb{N}$ *such that* $\forall m \in E : ts(m) \in V[rep(m)]$. $\mathcal{S}^{opt}$ *denotes the set of opt states. In the initial opt state,* $E = \emptyset$, $V[i] = \emptyset$ *for each* $i \in \mathcal{R}$, $count = 0$. *For a valid set* M, *an opt state* $(E, V, count)$ *is called* M*-compatible if* $E \subseteq M$ *and for all* $r \in \mathcal{R}$ *and* $c \in V[r]$, *there is some* x *such that* $(x, c, r) \in M$.

Let $\mathcal{S} = \mathcal{S}^{original} \cup \mathcal{S}^{opt}$ be the set of all states. For a state $S \in \mathcal{S}$, we use the notation $S.E$, $S.T$, $S.V$, and $S.count$, to refer to the appropriate components of S.

To show the correspondence between the behaviour of the two implemenations, we have to define when an original state is equivalent to an opt state. The intuition is that $(E, T, count)$ is equivalent to $(E, V, count)$, if the elements seen so far in $E \cup T$ correspond to the timestamps represented in V—this exploits the fact that for each element $(x, c, r) \in E \cup T$ the pair (c, r) is unique.

Definition 52. *Let* $S = (E, T, count)$ *be an original state,* $S' = (E', V', count')$ *be an opt state, and* M *be a valid set of labelled elements. We say that* S *is* M*-equivalent to* S' *(denoted* $S \equiv_M S'$*) if* $E = E'$, $count = count'$, $E \cup T \subseteq M$ *and for all* $m \in M$, $m \in E \cup T$ *iff* $ts(m) \in V'[rep(m)]$.

We now characterize an original state in terms of *Seen*() and *Deletes*() as we did in the previous section for an optimized state.

Definition 53. *Let $S = (E, T, count)$ be an original state. Then, we define the following*

$$Exists_{orig}(S) = \{(ts(m), rep(m)) \mid m \in E\}$$

$$Deletes_{orig}(S) = \{(ts(m), rep(m)) \mid m \in T\}$$

$$Seen_{orig}(S) = Exists_{orig}(S) \cup Deletes_{orig}(S)$$

Since for any original state S, we have $S.E \cap S.T = \emptyset$ from the definitions above, we can observe that

$$Exists_{orig}(S) = Seen_{orig}(S) \setminus Deletes_{orig}(S)$$

and

$$Deletes_{orig}(S) = Seen_{orig}(S) \setminus Exists_{orig}(S)$$

In the following, we use $S_1 \leq_{orig} S_2$ to denote that $\text{COMPARE}_{orig}(S_1, S_2)$ returns TRUE, for two original states S_1 and S_2. We shall denote by $S \circ \mathsf{merge}_{orig}(S)$ the state obtained by merging the state S_2 into S_1 using the MERGE function of the original implementation. We now present the following three results which will characterize the $\leq_{orig}$ and $\mathsf{merge}_{orig}()$ in terms of $Seen_{orig}()$ and $Deletes_{orig}()$

Proposition 54. *For a pair of original states S_1 and S_2 $S_1 \leq_{orig} S_2$ iff $Seen_{orig}(S_1) \subseteq Seen_{orig}(S_2)$ and $Deletes_{orig}(S_1) \subseteq Deletes_{orig}(S_2)$*

Proof. From the code,

$$\begin{aligned} S_1 \leq_{orig} S_2 &\iff (S_1.E \cup S_1.T \subseteq S_2.E \cup S_2.T) \wedge (S_1.T \subseteq S_2.T) \\ &\iff (Seen_{orig}(S_1) \subseteq Seen_{orig}(S_2)) \wedge (Deletes_{orig}(S_1) \subseteq Deletes_{orig}(S_2)) \\ &\qquad \text{(By the definition of } Seen_{orig}() \text{ and } Deletes_{orig}()) \end{aligned}$$

□

Proposition 55. *For three original states S_1, S_2 and S_3, we have $S_3 = S_1 \circ \mathsf{merge}_{orig}(S_2)$ iff*

- $Seen_{orig}(S_3) = Seen_{orig}(S_1) \cup Seen_{orig}(S_2)$ *and*
- $Deletes_{orig}(S_3) = Deletes_{orig}(S_1) \cup Deletes_{orig}(S_2)$ *and*
- $S_3.count = S_1.count$.

Proof. From the code of the original implementation,

$$S_3 = S_1 \circ \mathsf{merge}_{orig}(S_2) \iff (S_3.T = S_1.T \cup S_2.T) \wedge$$
$$(S_3.E = S_1.E \setminus S_2.T \cup S_2.E \setminus S_1.T) \wedge S_3.count = S_1.count$$

Now,

$$S_3.T = S_1.T \cup S_2.T \iff Deletes(S_3) = Deletes(S_1) \cup Deletes(S_2)$$

and

$$\begin{aligned}
&S_3.E = S_1.E \setminus S_2.T \cup S_2.E \setminus S_1.T \\
&\iff S_3.E = (S_1.E \cup S_1.T) \setminus (S_1.T \cup S_2.T) \cup (S_2.E \cup S_2.T) \setminus (S_1.T \cup S_2.T) \\
&\quad \text{(Since } S_i.E \cap S_i.T = \emptyset \text{ for } i \in \{1,2\}) \\
&\iff S_3.E = ((S_1.E \cup S_1.T) \setminus S_3.T) \cup ((S_2.E \cup S_2.T) \setminus S_3.T) \\
&\iff S_3.E \cup S_3.T = (S_1.E \cup S_1.T) \cup (S_2.E \cup S_2.T) \\
&\iff Seen_{orig}(S_3) = Seen_{orig}(S_1) \cup Seen_{orig}(S_2)
\end{aligned}$$

This proves the proposition. □

We now show that the state produced by the MERGE$_{orig}$ of a pair of states is the least upper bound of the two states.

Proposition 56. *For three original states S_1, S_2 and S_3 $S_3 = S_1 \circ \mathsf{merge}_{orig}(S_2)$ iff S_3 is the least upper bound of S_1 and S_2 with respect to $\leq_{orig}$.*

Proof. Let S' be an upper bound of S_1 and S_2 with respect to $\leq_{orig}$.

Thus we have $S_1 \leq_{orig} S'$ and $S_2 \leq_{orig} S'$. By Proposition 54, we have for $i \in \{1,2\}$, $Seen_{orig}(S_i) \subseteq Seen_{orig}(S')$ and $Deletes_{orig}(S_i) \subseteq Deletes(S')$. Thus,

$$Seen_{orig}(S_1) \cup Seen_{orig}(S_2) \subseteq Seen_{orig}(S')$$

and

$$Deletes_{orig}(S_1) \cup Deletes_{orig}(S_2) \subseteq Deletes_{orig}(S')$$

. Since $S_3 = S_1 \circ \mathsf{merge}_{orig}(S_2)$, by Proposition 55, we have

$$Seen_{orig}(S_3) \subseteq Seen_{orig}(S')$$

and

$$Deletes_{orig}(S_3) \subseteq Deletes_{orig}(S')$$

.

Once again, by Proposition 54, we have

$$S_3 \leq_{orig} S'$$

Thus, S_3 is not just an upper bound of S_1 and S_2 with respect to $\leq_{orig}$, but also a least upper bound.

□

We shall denote the *Seen*(), *Deletes*() and *Exists*() sets defined in the previous section for opt states as $Seen_{opt}()$, $Deletes_{opt}()$ and $Exists_{opt}()$ to contrast them with the corresponding sets defined for the original state. We now provide a characterization of $M-$equivalent original and opt states in terms of their respective *Seen*() and *Deletes*() sets.

Proposition 57. *Let S and S' respectively be original and opt states such that both S and S' are $M-$compatible for some $M \subseteq \mathcal{M}$. Then,*

$$S \equiv_M S' \iff Seen_{orig}(S) = Seen_{opt}(S') \wedge Deletes_{orig}(S) = Deletes_{opt}(S') \wedge S.count = S'.count$$

Proof. From the definition of $M-$equivalence,

$$\begin{aligned}
S \equiv_M S' \iff & (\forall m \in M : m \in S.E \cup S.T \iff ts(m) \in S'.V[rep(m)]) \wedge \\
& (S.E = S'.E) \wedge (S.count = S'.count) \\
\iff & Seen_{orig}(S) = Seen_{opt}(S') \wedge \\
& Exists_{orig}(S) = Exists_{opt}(S') \wedge \\
& (S.count = S'.count) \\
\iff & Seen_{orig}(S) = Seen_{opt}(S') \wedge \\
& Seen_{orig}(S) \setminus Exists_{orig}(S) = Seen_{opt}(S') \setminus Exists_{opt}(S') \wedge \\
& (S.count = S'.count) \\
\iff & Seen_{orig}(S) = Seen_{opt}(S') \wedge \\
& Deletes_{orig}(S) = Deletes_{opt}(S') \wedge \\
& (S.count = S'.count)
\end{aligned}$$

□

Thus the following observation is immediate for any valid set M of labelled elements.

Observation 58. *1. For every M-compatible original state $S = (E, T, count)$, there is exactly one opt state S' such that $S \equiv_M S'$.*

2. *For every M-compatible opt state $S' = (E', V', count')$, there is exactly one original state S with $S \equiv_M S'$.*

The global configurations of the original and optimized implementations consists of tuples of their respective states.

Definition 59. *A original (respectively, opt) configuration is a tuple $(S_0, \ldots, S_{N-1})$ of original (respectivelly, opt) states, one for each replica. The collection of original and opt configurations are denoted, respectively, by $\mathcal{C}^{original}$ and $\mathcal{C}^{opt}$. A original (respectively, opt) configuration $C = (S_0, \ldots, S_{N-1})$ is an initial configuration if each S_i is an initial original (repectively, opt) state.*

Given a valid set of labelled elements M, an original configuration $C = (S_0, \ldots, S_{N-1})$ is M-equivalent to an opt configuration $C' = (S'_0, \ldots, S'_{N-1})$ iff $S_i \equiv_M S'_i$ for all $i \in \mathcal{R}$.

Since the auxiliary information generated by the original and our generic optimized algorithms are different we need a way to associate the corresponding update-receive operations of the two implementations. Towards this, we define the following abstract operations on the states of our formal model.

Definition 60. *The set of operations is given by*

$$\{r.\mathsf{contains}(x), r.\mathsf{add}(x), r.\mathsf{addrcv}(m), r.\mathsf{delete}(x), r.\mathsf{delrcv}(M), r.\mathsf{delrcv}(V'), r.\mathsf{merge}(S)\}$$

where $r \in \mathcal{R}, x \in \mathsf{Univ}, m \in \mathcal{M}$, $M, M' \subseteq \mathcal{M}$, $V' : \mathcal{R} \to \mathcal{I}$ and $S \in \mathcal{S}$.

We say that an operation is original (respectively, opt) operation if the state S on which it is applied satisfies $S \in \mathcal{S}^{original}$ (respectively, $S \in \mathcal{S}^{opt}$). The sets of original and opt operations are denoted $Ops^{original}$ and Ops^{opt}, respectively.

For an operation o, $Site(o) = r$ where $o = r.\mathsf{contains}(x), r.\mathsf{add}(x), r.\mathsf{merge}(S)$, etc. This is the replica at which the operation is applied.

Given a valid sets of labelled elements M, a subset $M' \subseteq M$ and an interval version vector V' we say that M' is M-equivalent to V' (written as $M' \equiv_M V'$) iff $\forall m \in M, m \in M' \iff ts(m) \in V'[rep(m)]$. If $M = M'$ then we drop the subscript M and write $M' \equiv V'$.

Given a valid set of labelled elements M, a original operation o is M-equivalent to an opt operation o' iff either $o = o'$ or ($o = r.\mathsf{delrcv}(M')$, $o' = r.\mathsf{delrcv}(V')$, and $M' \equiv_M V'$) or ($o = r.\mathsf{merge}(S)$, $o' = r.\mathsf{merge}(S')$, and $S \equiv_M S'$).

The operations $r.\mathsf{add}(x)$ and $r.\mathsf{delete}(x)$ take an element as argument and prepare the corresponding set of labelled elements to be added or deleted. This information is propagated through the network. The actual addition or deletion is handled by $r.\mathsf{addrcv}(m)$ and $r.\mathsf{delrcv}(M)$ (or $r.\mathsf{delrcv}(V')$ in case of the opt-implementation), respectively, which add or delete the labelled elements provided as an argument. Formally, the effect of these operations on original and opt configurations is captured through the transition relations described below.

Definition 61. *Given two original configurations $C = (S_0, \ldots, S_{N-1})$ and $C' = (S'_0, \ldots, S'_{N-1})$ from $\mathcal{C}^{original}$, and a original operation $o \in Ops^{original}$ with $Site(o) = r$, we say that $C \xrightarrow{o}_{\text{orig}} C'$ iff:*

- *For $i \neq r$, $S_i = S'_i$.*
- *If $o = r.\mathsf{contains}(x)$ or then $E' = E$, $T = T'$, and $count = count'$ and $Ret(o) = S_i.\mathsf{contains}(x)$.*
- *if $o = r.\mathsf{add}(x)$ with $m := S_r.\textsc{add.prepare}(x)$ or $o = r.\mathsf{addrcv}(m)$ then $E' = (E \cup \{m\}) \setminus T$, $T' = T$, and $count' = \begin{cases} ts(m) + 1 & \text{if} \quad rep(m) = r \\ count & otherwise \end{cases}$*
- *If $o = r.\mathsf{delete}(x)$ with $M := S_r.\textsc{delete.prepare}(x)$ or $o = r.\mathsf{delrcv}(M)$ then $E' = E \setminus M$, $T' = T \cup M$, and $count' = count$.*
- *If $o = r.\mathsf{merge}(S)$ and $S = (E_1, T_1, count_1)$ then $E' = (E \setminus T_1) \cup (E_1 \setminus T)$, $T' = T \cup T_1$ and $count' = count$.*

Definition 62. *Given two opt configurations $C = (S_0, \ldots, S_{N-1})$ and $C' = (S'_0, \ldots, S'_{N-1})$ from $\mathcal{C}^{opt}$, and an opt operation $o \in Ops^{opt}$ with $Site(o) = r$, we say that $C \xrightarrow{o}_{\text{opt}} C'$ iff:*

- *for $i \neq r$, $S_i = S'_i$*
- *with $S_r = (E, V, count)$ and $S'_r = (E', V', count')$, the following conditions are satisfied:*
  - *if $o = r.\mathsf{contains}(x)$ then $E' = E$ and $V' = V$ and $Ret(o) = S_i.\mathsf{contains}(x)$.*
  - *if $o = r.\mathsf{add}(x)$ with $m = S_r.\textsc{add.prepare}(x)$ or $o = r.\mathsf{addrcv}(m)$ then*
    - *$E' = E \cup \{m\}$,*
    - *$V'[rep(m)] = \boldsymbol{add}(V[rep(m)], \{ts(m)\})$ and $V'[i] = V[i]$ for $i \neq rep(m)$, and*
    - *$count' = \begin{cases} ts(m) + 1 & \text{if} \quad rep(m) = r \\ count & otherwise \end{cases}$*
  - *if $o = r.\mathsf{delete}(x)$ with $S_r.\textsc{delete.prepare}(x) = V''$ or $o = r.\mathsf{delrcv}(V'')$ then*
    - *$E' = E \setminus M$ where $M = \{m \in E \mid ts(m) \in V''[rep(m)]\}$.*
    - *for all $i \in \mathcal{R}$: $V'[i] = V[i] \cup V''[i]$, and*
    - *$count' = count$.*
  - *if $o = r.\mathsf{merge}(S)$ and $S = (E_1, V_1, count_1)$ then*
    - *$E' = \{m \in E \cup E_1 \mid m \in E \cap E_1 \vee ts(m) \notin V[rep(m)] \cap V_1[rep(m)]\}$.*
    - *for all $i \in \mathcal{R}$: $V'[i] = V[i] \cup V_1[i]$.*

$*$ $count' = count$.

In this section, when we say *run* of a system, we mean a sequence of configurations starting from the initial configuration that respects the transition relation. In addition, we have to ensure that each update-receive operation has a corresponding update operation that occurs before the update-receive operation it in the sequence.

Definition 63. *A original run (respectively, opt run) of an OR-set implementation is a sequence $C_0 o_1 C_1 \cdots C_{n-1} o_n C_n$ where (letting each $C_i = (S_0^i, \ldots, S_{N-1}^i)$):*

- *each C_i is a original (respectively, opt) configuration*
- *C_0 is an initial original (respectively, opt) configuration*
- *each o_i is a original (respectively, opt) operation*
- *for all $i < n$, $C_i \xrightarrow{o_{i+1}}_{\text{orig}} C_{i+1}$ (respectively, $C_i \xrightarrow{o_{i+1}}_{\text{opt}} C_{i+1}$)*
- *if o_i is an $r.\mathsf{add}(x)$ operation, then the corresponding*

  $$S_r^{i-1}.\text{ADD.PREPARE}(x) = (x, S_r^{i-1}.count, r) = m$$

  which will be broadcast to all the other replicas. Further, $S_r^{i-1} \xrightarrow{r.\mathsf{addrcv}(m)}_{\text{orig}} S_r^i$ (resp. $S_r^{i-1} \xrightarrow{r.\mathsf{addrcv}(m)}_{\text{opt}} S_r^i$)
- *if o_i is an $r.\mathsf{delete}(x)$ operation, then $S_r^{i-1}.\text{DELETE.PREPARE}(x) = M$ in case of original-implementation (resp. $S_r^{i-1}.\text{DELETE.PREPARE}(x) = V'$ in case of opt-implementation) such that $M = \{m \in S_r^{i-1}.E \mid data(m) = x\}$ (respectively, $V' : \mathcal{R} \to \mathcal{I}$ such that for all $m \in S_r^{i-1}.E$ with $data(m) = x$, $ts(m) \in V'[rep(m)]$). and $S_r^{i-1} \xrightarrow{r.\mathsf{delrcv}(M)}_{\text{orig}} S_r^i$ (respectively $S_r^{i-1} \xrightarrow{r.\mathsf{delrcv}(V')}_{\text{opt}} S_r^i$)*
- *if o_i is an $r'.\mathsf{addrcv}((x, c, r))$ operation, then there exists $j < i$ such that o_j is an $r.\mathsf{add}(x)$ operation and $c = S_r^{j-1}.count$.*
- *if o_i is a $r'.\mathsf{delrcv}(M)$ operation (respectively, $r'.\mathsf{delrcv}(V')$ operation), then there exists $j < i, r \in \mathcal{R}$ such that o_j is a $r.\mathsf{delete}(x)$ operation and $M = \{m \in S_r^{j-1}.E \mid data(m) = x\}$ (respectively, $V' : \mathcal{R} \to \mathcal{I}$ such that for all $m \in S_r^{j-1}.E$ with $data(m) = x$, $ts(m) \in V'[rep(m)]$).*
- *if o_i is a $r.\mathsf{merge}(S)$ operation, then S is a state in some earlier configuration C_j.*

By a run (without any qualifiers) we mean either a original run or an opt run. For a run $\alpha = C_0 o_1 C_1 \cdots C_{n-1} o_n C_n$, define $C(\alpha)$ to be C_n.

We can now define what it means for a original run to be equivalent to an opt run. We begin with the following preliminary definition of the set of labelled elements generated during a run.

Definition 64. *If $\alpha = C_0 o_1 C_1 \ldots C_{n-1} o_n C_n$ is a run, then $\mathcal{M}(\alpha) = \{m \mid \exists i \leq n, r \in \mathcal{R}$ such that o_i is an $r.\mathsf{addrcv}(m)$ operation$\}$.*

Proposition 65. *For any run α, $\mathcal{M}(\alpha)$ is a valid set of labelled elements.*

Proof. We prove by induction on the length of α the following property (letting $C(\alpha) = (S_0, \ldots, S_{N-1})$ and $\mathcal{M}(\alpha) = M$):

$$[\forall m \in M.ts(m) < S_{rep(m)}.count] \wedge M \text{ is valid.}$$

The base case is a run with zero operations. In this case $M = \emptyset$ and all *count* values are zero. So the proposition is true.

Suppose now that $\alpha = \alpha' \cdot oC$ for some operation o and configuration C. Let $M' = \mathcal{M}(\alpha')$ and let $C(\alpha') = (S'_0, \ldots, S'_{N-1})$. We observe that the $M = M'$ and $S'_i.count = S_i.count$ for all $i \in \mathcal{R}$, except when $o = r.\mathsf{add}(x)$ with $S'_r.\textsc{add.prepare}(x) = m$ with $rep(m) = r, ts(m) = S'_r.count$ In this cases, $M = M' \cup \{m\}$ and $ts(m) = S'_r.count$ and therefore $S_r.count = S'_r.count + 1$.

By induction hypothesis, for every $m' \in M'$, $ts(m') < S'_{rep(m')}.count$. And $ts(m) < S_r.count$. And hence the first part of the statement is true. Also, since every $m' \in M'$ with $rep(m) = r$, $ts(m') < S'_r.count = ts(m)$. Thus m is distinct from all other labelled elements in M. And the statement of the proposition is proved. □

Definition 66. *For an original run*

$$\alpha = C_0 o_1 C_1 \ldots C_{n-1} o_n C_n,$$

and an opt run

$$\alpha' = C'_0 o'_1 C'_1 \ldots C'_{n-1} o'_n C'_n,$$

we say that α is equivalent to α' (denoted $\alpha \equiv \alpha'$) iff for every $i \leq n$, $C_i \equiv_{\mathcal{M}(\alpha)} C'_i$ and $o_i \equiv_{\mathcal{M}(\alpha)} o'_i$.

Remark Though it is not obvious from the definition above, the relation $\alpha \equiv \alpha'$ is symmetric. We will show that if α is equivalent to α', then $\mathcal{M}(\alpha) = \mathcal{M}(\alpha')$.

We now show that the optimized OR-set implementation behaves the same as the original implementation. The strategy is to set up a correspondence between the configurations of the original and the optimized implementations through a bisimulation.

For a pair of opt states S and S', we say $S \leq_{opt} S'$ iff $\text{COMPARE}_{opt}(S, S')$ returns True.

Our first claim is that $\equiv_M$ guarantees query equivalence. We use the notation $S.\mathsf{merge}_{orig}(S')$ to denote the resulting state of a replica r on invoking $r.\text{MERGE}_{orig}(S')$ with r in state S. $S.\mathsf{merge}_{opt}(S')$ has a similar meaning.

Lemma 67. *Suppose M is a valid set of labelled elements. Suppose S_1, S_2 are M-compatible original states, and S_1', S_2' are M-compatible opt states such that $S_1 \equiv_M S_1'$ and $S_2 \equiv_M S_2'$. Then*

1. *For any $x \in \mathsf{Univ}$, $S_1.\mathsf{contains}_{\text{orig}}(x)$ returns True iff $S_1'.\mathsf{contains}_{\text{opt}}(x)$ returns True.*
2. *$S_1 \leq_{\text{orig}} S_2$ iff $S_1' \leq_{\text{opt}} S_2'$.*
3. *$S_1.\mathsf{merge}_{\text{orig}}(S_2) \equiv_M S_1'.\mathsf{merge}_{\text{opt}}(S_2')$*
4. *If $S_1.\mathsf{merge}_{\text{orig}}(S_2)$ is the least upper bound of S_1 and S_2, then $S_1'.\mathsf{merge}_{\text{opt}}(S_2')$ is the least upper bound of S_1' and S_2'.*

Proof. Let $S_i = (E_i, T_i, count_i)$ and $S_i' = (E_i', V_i', count_i')$, for $i \in \{1, 2\}$. Given the assumptions, the following statements hold for $i \in \{1, 2\}$:

- $E_i = E_i'$,
- for $m \in M$, $m \in E_i \cup T_i$ iff $ts(m) \in V_i[rep(m)]$.
- for all $r \in \mathcal{R}$, $V_i'[r] = \{ts(m) \mid m \in E_i \cup T_i, rep(m) = r\}$.
- $T_i = \{m \in M \setminus E_i' \mid ts(m) \in V_i'[rep(m)]\}$.
- $Seen_{orig}(S_i) = Seen_{opt}(S_i')$
- $Deletes_{orig}(S_i) = Deletes_{opt}(S_i')$

1. $S_1.\mathsf{contains}_{orig}(x)$ returns True iff there is $m \in E_1$ such that $data(m) = x$ iff there is $m \in E_1'$ such that $data(m) = x$ iff $S_1'.\mathsf{contains}_{opt}(x)$ returns True.
2. $$\begin{aligned} S_1 \leq_{orig} S_2 &\iff Seen_{orig}(S_1) \subseteq Seen_{orig}(S_2) \wedge Deletes_{orig}(S_1) \subseteq Deletes_{opt}(S_2) \\ &\quad \text{(By Proposition 54)} \\ &\iff Seen_{opt}(S_1) \subseteq Seen_{opt}(S_2) \wedge Deletes_{orig}(S_1) \subseteq Deletes_{opt}(S_2) \\ &\quad \text{(By Proposition 57)} \\ &\iff S_1' \leq_{opt} S_2' \text{(By Lemma 48)} \end{aligned}$$

3.

$$
\begin{aligned}
S_3 = S_1.\mathsf{merge}_{orig}(S_2) \iff\ & Seen_{orig}(S_3) = Seen_{orig}(S_1) \cup Seen_{orig}(S_2) \wedge \\
& Deletes_{orig}(S_3) = Deletes_{orig}(S_1) \cup Deletes_{orig}(S_2) \wedge \\
& S_3.count = S_1.count \\
& \text{(By Proposition 55)} \\
\iff\ & Seen_{opt}(S'_3) = Seen_{opt}(S'_1) \cup Seen_{opt}(S'_2) \wedge \\
& Deletes_{opt}(S'_3) = Deletes_{opt}(S'_1) \cup Deletes_{opt}(S'_2) \wedge \\
& S'_3.count = S'_1.count \\
& \text{(By Proposition 57)} \\
\iff\ & S'_3 = S'_1.\mathsf{merge}_{opt}(S'_2) \text{(By Lemma 48)}
\end{aligned}
$$

4. This result follows from Lemma 48 and Proposition 56.

□

Our second claim is that equivalent runs reach configurations that are strongly equivalent to each other. We use the following useful observation which simplifies the proof. A bit of notation first: for a original (respectively, opt) state S, we define $\mathsf{incr}(S)$ to be the original (respectively, opt) state S' which is the same as S except that $S'.count = S.count + 1$.

Observation 68. *Let* $o_1 = r.\mathsf{add}(x)$ *with* $r.\text{ADD.PREPARE}(x) := m$.

Let $o_1^{rcv} = r.\mathsf{addrcv}(m)$.

Let $o_2 = r.\mathsf{delete}(x)$ *with* $r.\text{DELETE.PREPARE}(x) = M$.

Let $o_2^{rcv} = r.\mathsf{delrcv}(M)$.

Let $o'_2 = r.\mathsf{delrcv}(x)$ *with* $r.\text{DELETE.PREPARE}(x) = V$.

Let $o_2^{'rcv} = r.\mathsf{delrcv}(V)$.

Let $S_{add}^{original} = (\{m\}, \emptyset, 0), \quad S_{del}^{original} = (\emptyset, M, 0),$

$S_{add}^{opt} = (\{m\}, V'_1, 0)$, *and* $\quad S_{del}^{opt} = (\emptyset, V'_2, 0)$
where

- $V_1'[rep(m)] = \{ts(m)\}$, *and* $V_1'[i] = \emptyset$ *for* $i \neq rep(m)$.
- $V_2'[i] = V[i]$ *for all* $i \in \mathcal{R}$.

Let $C = (S_0, \ldots, S_r, \ldots, S_{N-1})$ *and* $C' = (S_0, \ldots, S_r', \ldots, S_{N-1})$ *be two configurations. Then*

1. *If* $C \xrightarrow{o_1}_{\text{orig}} C'$ *or* $C \xrightarrow{o_1^{rcv}}_{\text{orig}} C'$ *then*

$$S_r' = \begin{cases} \mathit{incr}(S_r.\mathsf{merge}_{\text{orig}}(S_{add}^{original})) & \text{if} \quad rep(m) = r \\ S_r.\mathsf{merge}_{\text{orig}}(S_{add}^{original}) & \text{otherwise} \end{cases}$$

2. *If* $C \xrightarrow{o_1}_{\text{opt}} C'$ *or* $C \xrightarrow{o_1^{rcv}}_{\text{opt}} C'$ *then*

$$S_r' = \begin{cases} \mathit{incr}(S_r.\mathsf{merge}_{\text{opt}}(S_{add}^{opt})) & \text{if} \quad rep(m) = r \\ S_r.\mathsf{merge}_{\text{opt}}(S_{add}^{opt}) & \text{otherwise} \end{cases}$$

3. *If* $C \xrightarrow{o_2}_{\text{orig}} C'$ *or* $C \xrightarrow{o_2^{down}}_{\text{orig}} C'$ *then* $S_r' = S_r.\mathsf{merge}_{\text{orig}}(S_{del}^{original})$.
4. *If* $C \xrightarrow{o_2'}_{\text{opt}} C'$ *then* $S_r' = S_r.\mathsf{merge}_{\text{opt}}(S_{del}^{opt})$.

Lemma 69. *Suppose* α *is a original run and* α' *is an opt run such that* $\alpha \equiv \alpha'$. *Then*

1. $\mathcal{M}(\alpha) = \mathcal{M}(\alpha')$.
2. *For any original operation* o *and original configuration* C *such that* $\alpha \cdot oC$ *is a original run, there exists an opt operation* o' *and an opt configuration* C' *such that* $\alpha' \cdot o'C'$ *is an opt run and* $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.
3. *For any opt operation* o' *and opt configuration* C' *such that* $\alpha' \cdot o'C'$ *is an opt run, there exists a original operation* o *and a original configuration* C *such that* $\alpha \cdot oC$ *is a original run and* $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

Proof. We prove the lemma by induction on the number of operations in α (and hence in α'). Assume that the result holds for all equivalent runs α and α', each having strictly fewer than n operations. Now let $\alpha = C_0 o_1 C_1 \cdots C_{n-1} o_n C_n$ be a original run and $\alpha' = C_0' o_1' C_1' \cdots C_{n-1}' o_n' C_n'$ be an opt run such that $\alpha \equiv \alpha'$. Let $M_n = \mathcal{M}(\alpha)$ and $M_n' = \mathcal{M}(\alpha')$. When $n > 0$, let $M_{n-1} = \mathcal{M}(C_0 o_1 C_1 \cdots C_{n-1})$ and $M_{n-1}' = \mathcal{M}(C_0' o_1' C_1' \cdots C_{n-1}')$.

1. If $n = 0$ then $M_n = M_n' = \emptyset$. Hence the statement holds. Assume that $n > 0$. Then by induction hypothesis $M_{n-1} = M_{n-1}'$. The following cases need to be considered.

$o_n = r.\mathsf{contains}(x)$ **or** $o_n = r.\mathsf{add}(x)$ **or** $o_n = r.\mathsf{delete}(x)$**:** In this case $o_n = o'_n$, $M_n = M_{n-1}$, and $M'_n = M'_{n-1}$. Therefore $M_n = M'_n$.

$o_n = r.\mathsf{add}(x)$**:** Suppose $r.\text{ADD.PREPARE}(x) = m$ In this case also $o_n = o'_n$. Since $rep(m) = r$, then $M_n = M_{n-1} \cup \{m\}$ and $M'_n = M'_{n-1} \cup \{m\}$. Hence $M_n = M'_n$.

$o_n = r.\mathsf{addrcv}(m)$**:** In this case also $o_n = o'_n$. Since $rep(m) \neq r$, then $M_n = M_{n-1}$ and $M'_n = M'_{n-1}$. Hence $M_n = M'_n$.

$o_n = r.\mathsf{delete}(x)$**:** Suppose $r.\text{DELETE.PREPARE}^{original}(x) = M$. Then, $o'_n = o_n$ with $r.\text{DELETE.PREPARE}^{opt}(x) = V$ such that $M \equiv_{M_n} V$. Further, $M \subseteq M_{n-1} = M'_{n-1}$. Therefore $M_n = M_{n-1}$. and importantly, $M'_n = M'_{n-1}$. Hence $M_n = M'_n$.

$o_n = r.\mathsf{delrcv}(M)$**:** like in the earlier case we set $o'_n = r.\mathsf{delrcv}(V)$ such that $M \equiv_{M_n} V$. As before, $M \subseteq M_{n-1} = M'_{n-1}$. Therefore $M_n = M_{n-1}$. and importantly, $M'_n = M'_{n-1}$. Hence $M_n = M'_n$.

$o_n = r.\mathsf{merge}(S)$**:** In this case $o'_n = r.\mathsf{merge}(S')$ such that S and S' are states from configurations C_i and C'_i respectively, $i < n$. Thus both the states are M_{n-1}-compatible. But then $M_n = M_{n-1}$ and $M'_n = M'_{n-1}$. Hence $M_n = M'_n$.

2. Let o and C be a original operation and configuration, respectively, such that $\alpha \cdot oC$ is a run. Let $M = \mathcal{M}(\alpha \cdot oC)$. Note that since $C_n \equiv_{M_n} C'_n$, it is also the case $C_n \equiv_M C'_n$. We aim to show that there is an opt operation o' and an opt configuration C' such that $\alpha' \cdot o'C'$ is a run and $\alpha \cdot oC \equiv \alpha' \cdot o'C'$. Let $Site(o) = r$. Then the only state change happens in replica r. We denote the state of r in C_n and C'_n by S_1 and S'_1 respectively and the state of r in C and C' by S_2 and S'_2 respectively.

   The following cases need to be considered.

   $o = r.\mathsf{contains}(x)$ **or** $o = r.\mathsf{add}(x)$ **or** $o = r.\mathsf{delete}(x)$**:** In this case $C = C_n$. We choose o' to be o and $C' = C'_n$. From the fact that $\alpha \cdot oC$ is a original run, it easily follows that $\alpha' \cdot o'C'$ is an opt run. Furthermore, $o \equiv_M o'$ and $C = C_n \equiv_M C'_n = C'$. Therefore $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

   $o = r.\mathsf{add}(x)$**:** Suppose $r.\text{ADD.PREPARE}(x) = m$. In this case, $M = M_n \cup \{m\}$. We choose o' to be o and C' such that $C'_n \xrightarrow{o'}_{opt} C'$. Since $\alpha \cdot oC$ is a original run, $\alpha' \cdot o'C'$ is an opt run. By the observation preceding this lemma, there is a original state S and an opt state S' such that $S \equiv_M S'$, $S_2 = S_1.\mathsf{merge}_{orig}(S)$, and $S'_2 = S'_1.\mathsf{merge}_{opt}(S')$. Thus it follows that $S_2 \equiv_M S'_2$ (from part 3 of Lemma 67). Therefore $C_n \equiv_M C'_n$ and hence $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

   $o = r.\mathsf{addrcv}(m)$**:** In this case, $M = M_n$. We choose o' to be o and C' such that $C'_n \xrightarrow{o'}_{opt} C'$. Since $\alpha \cdot oC$ is a original run, $\alpha' \cdot o'C'$ is an opt run. By the observation preceding this lemma, there is a original state S and an opt state S' such that

$S \equiv_M S'$, $S_2 = S_1.\mathsf{merge}_{orig}(S)$, and $S_2' = S_1'.\mathsf{merge}_{opt}(S')$. Thus it follows that $S_2 \equiv_M S_2'$ (from part 3 of Lemma 67). Therefore $C_n \equiv_M C_n'$ and hence $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

$o_n = r.\mathsf{delete}(x)$**:** Suppose $r.\text{DELETE.PREPARE}_{original}(x) = M'$. In this case, $M = M_n$. Choose $o' = o$ with $r.\text{DELETE.PREPARE}_{opt}(x) = V'$ such that $M' \equiv_M V'$ and C' such that $C_n' \xrightarrow{o'}_{opt} C'$. Since $\alpha \cdot oC$ is a original run, $\alpha' \cdot o'C'$ is an opt run. By the observation preceding this lemma, there is a original state S and an opt state S' such that $S \equiv_M S'$, $S_2 = S_1.\mathsf{merge}_{orig}(S)$, and $S_2' = S_1'.\mathsf{merge}_{opt}(S')$. Thus it follows that $S_2 \equiv_M S_2'$ (from part 3 of Lemma 67). Therefore $C_n \equiv_M C_n'$ and hence $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

$o_n = r.\mathsf{delrcv}(M')$**:** In this case, $M = M_n$. Choose o' to be $r.\mathsf{delrcv}(V')$ with $M' \equiv_M V'$ and C' such that $C_n' \xrightarrow{o'}_{opt} C'$. Since $\alpha \cdot oC$ is a original run, $\alpha' \cdot o'C'$ is an opt run. By the observation preceding this lemma, there is a original state S and an opt state S' such that $S \equiv_M S'$, $S_2 = S_1.\mathsf{merge}_{orig}(S)$, and $S_2' = S_1'.\mathsf{merge}_{opt}(S')$. Thus it follows that $S_2 \equiv_M S_2'$ (from part 3 of Lemma 67). Therefore $C_n \equiv_M C_n'$ and hence $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

$o_n = r.\mathsf{merge}(S)$**:** In this case too, $M = M_n$. Since $\alpha \cdot oC$ is a original run, there exists a replica r' and an index $i \leq n$ such that S is the local state of r' in C_i. Choose $o' = r.\mathsf{merge}(S')$ where S' is the local state of r' in C_i', and choose C' such that $C_n' \xrightarrow{o'}_{opt} C'$. It is easily seen that $\alpha' \cdot o'C'$ is an opt run. Since $\alpha \equiv \alpha'$, $S \equiv_M S'$ and hence $o \equiv_M o'$. Since $S \equiv_M S'$, $S_2 = S_1.\mathsf{merge}_{orig}(S)$, and $S_2' = S_1'.\mathsf{merge}_{opt}(S')$, it follows that $S_2 \equiv_M S_2'$ (from part 3 of Lemma 67). Therefore $C \equiv_M C'$ and hence $\alpha \cdot oC \equiv \alpha' \cdot o'C'$.

3. Similar to the proof of previous item by swapping the roles of o and o', α and α', and C and C'.

□

We can now describe the correspondence we seek between original runs and opt runs. We match a original configuration C with an opt configuration C' if they can be reached by equivalent runs

Definition 70. *Let $\mathcal{B}$ be a binary relation on $\mathcal{C}^{original} \times \mathcal{C}^{opt}$ defined by*

$\mathcal{B} \stackrel{\text{def}}{=} \{(C, C') \mid \exists$ *a original run α and an opt run α' such that $C = C(\alpha), C' = C(\alpha'), \alpha \equiv \alpha'$*$\}$.

Lemma 71. *$\mathcal{B}$ is a nontrivial bisimulation.*

Proof. Follows from Lemmas 67, 69, and the fact that $(C_0, C_0') \in \mathcal{B}$, where C_0 and C_0' are the initial original and initial opt configurations, respectively. □

Having established a bisimulation between the two systems, we can assert that our optimized implementation of OR-Sets inherits all the properties that have already been established for the original implementation in [Shapiro et al., 2011a].

In the next section we evaluate the space complexity for the original implementation, the implementation requiring causal-delivery, and the generic optimized implementation.

3.7 Space Complexity and Payload Size

Let $|\mathcal{R}| = N$. Let S_r denote the state of a replica $r \in \mathcal{R}$ at the end of some run α. Let n_r be the number of add operations at source replica r in the run α. Let $n_t = \sum_{r \in \mathcal{R}} n_r$. Let $n_m = max\{n_r \mid r \in \mathcal{R}\}$. Let $n_r^p(x)$ denote the number all $\mathsf{add}(x)$ operations $u \in \mathsf{Past}(S_r)$ such that $\mathsf{CoveringDel}(u) \cap \mathsf{Past}(S_r) = \emptyset$. Let $n_r^p = \Sigma_{x \in \mathsf{Univ}} n_r^p(x)$ denote the total number of add operations in the $\mathsf{Past}(S_r)$ that are not covered by any delete. Let $n_p = max\{n_r^p \mid r \in \mathcal{R}\}$. Clearly $n_p \leq n_t$.

The space required to store S_r in the original implementation (Algorithm 1) is bounded by $O(n_t \log(n_t))$, since every update seen by a replica is preserved either in the $S_r.E$ set or the $S_r.T$ tombstone set. If we refer to the data sent via the $\mathsf{add}(x)$-send and $\mathsf{delete}(x)$-send operations as the *payload* of those operations, then the payload-size of the $\mathsf{add}(x)$-send operation is $O(log(n_t)+N)$ since we only send the triple m. The payload-size of $\mathsf{delete}(x)$-send operation is $O(n_r^p(x) log(n_t))$ since we send all the triples of the form $(y, c, r'') \in S_r.E \setminus S_r.T$ such that $y = x$. Note that $n_r^p(x) \leq n_t$. The worst case occurs when all the add operations correspond to $\mathsf{add}(x)$ with no deletes covering any of them. Then the first delete ends up covering all those add operations, thus $n_r^p(x) = n_t$.

Let n_{int} denote the maximum number of intervals across any index r' in $V_r[r']$ for all the replicas r.

In our optimized implementation(Algorithm 3), the space required to store $S_r.V$ is bounded by $N n_{int} \log(n_t)$ and the space required to store $S_r.E$ is bounded by $n_p \log(n_t)$. The space required to store S_r is thus bounded by $O((n_p + N n_{int}) \log(n_t))$. In the worst case, n_{int} is bounded by $n_m/2$, which happens when r sees only the alternate elements generated by some replica. Thus the worst case complexity is $O((n_p + N n_m) \log(n_t))$. The payload-size of $\mathsf{add}(x)$-send operation is $O(log(n_t) + N)$. In case of $\mathsf{delete}(x)$-send operation, our generic optimized implementation sends the interval version vector associated with $\{(c', r') \mid (x, c', r') \in S_r.E\}$. If n_{int}^d is the total number of intervals in this interval version vector, then, the payload-size of the $\mathsf{delete}(x)$-send is $O(n_{int}^d log(n_t)$. Note that in the worst case $n_{int}^d = n_r^p(x)$ where each triple (x, c', r') to be deleted corresponds to an interval of single element. The best case occurs when there is exactly one interval in each of the N interval sequences and thus $n_{int}^d = N$. Thus the worst case payload-size of $\mathsf{delete}(x)$-send operation is $O(n_r^p(x) log(n_t))$ while the best-case payload-size of $\mathsf{delete}(x)$-send operation is $O(N log(n_t))$.

Note that the factor that is responsible for increasing the space complexity is the number of intervals in the interval sequences of the Interval Version Vector. We propose a reasonable way of bounding this value below.

Definition 72 (k-Causal-Delivery of an updates)**.** *Let $\alpha = (\rho, \varphi)$ be a run with $\rho = Io_1 \cdots o_n$.*

We say that updates are k-causally-delivered *in this run, iff for any update $\rho_r[i]$ at replica r, for any $j : 0 < j < |\rho|$, if $\rho_r[i] \in \mathsf{Past}_\alpha(\rho[j])$ then $\mathsf{Past}_\alpha(\rho_r[i-k]) \subseteq \mathsf{Past}_{(\rho,\varphi)}(\rho[j])$.*

When $k = 1$, this is the same as causal delivery.

Intuitively it means that when a replica r' sees the i^{th} add operation originating at replica r, it should have already seen all the operations that causally-precede the operations at r with index smaller than $i - k$. Thus *k-causal-delivery* ensures that the out of order delivery of updates is restricted to a bounded suffix of the sequence of operations submitted to the replicated datatype.

In particular, if the latest add operation u received by a replica r from a replica r' corresponds to the c^{th} add operation at r', then *k-causal-delivery* ensures that r would have received all the addrcv operations from r' whose index is less than or equal to $(c-k)$. Thus, $S_r.V[r']$ consists of one interval corresponding to the first $(c-k)$ add-receive operations from r' and at most $k/2$ intervals for the remaining k add-receive operations from r'. Since this is true for every r', we can conclude that n_{int} is bounded by $O(k)$. Hence, the space-complexity of the state S_r of an optimized-OR-Set in the presence of *k-causal-delivery* is $O((n_p + Nk)\log(n_t))$. If we assume causal delivery of updates ($k = 1$), the space complexity is bounded by $O((n_p+N)\log(n_t))$, which is the space complexity of the algorithm in [Bieniusa et al., 2012].

Coalescing adds: An Optimization

[Bieniusa et al., 2012] also provides an optimization of their algorithm which allows to coalesce add-operations of the same element by the same replica. This is achievable even in the case of *k-causal-delivery* which allows us to enforce a bound on the size of the E set at every replica by coalescing the adds of the same element originating from the same replica. The algorithm 4 captures this optimization. Whenever a replica r receives an $\mathsf{addrcv}(x, c, r')$ from a replica r' , it could evict all the triples (x, c', r') from its E set for in which $c' \leq c - k$ (lines 13–14 in Algorithm 4). Every replica can uniformly do this add-coalescing on receiving (x, c, r') since k-causal-delivery ensures that any replica which sees (x, c, r') would have seen all the triples (y, c', r') with $c' \leq c - k$. Similarly, whenever a replica sees a triple (x, c, r') as a part of the delete-receive, it knows that the source-replica of that delete operation would have seen all the triples (y, c', r') with $c' \leq c - k$, and would have performed add-coalescing of all such triples with $y = x$. So this replica on receiving the delete-receive can also evict all

triples (x, c', r') with $c' \leq c - k$ (lines 34-35) even though c' may not feature in the interval version vector $V'[r']$ sent as an argument of the delete-receive operation.

These optimisations ensures that every replica r stores at most k triples corresponding to a visible element $x \in \mathsf{Univ}$ added by the source replica r'. Thus if the number of visible elements is denoted by n_a, the size of the E set is bounded by $O(n_a k N \log(n_t))$. Thus the total space complexity with add-coalescing is $O((n_a + 1)kN \log(n_t))$. The payload-size of the $\mathsf{delete}(x)$-send operation is bounded by $O(Nk \log(n_t))$ since due to add coalition we would have only retained at most a single x-triple from the first interval, owing to k-causal delivery, there can be at most k intervals in the interval-sequence corresponding to each of the N replicas. If we assume causal delivery of updates ($k = 1$), the space complexity and the payload-size of the optimized solution with coalescing adds matches he space complexity and the payload-size of the solution in [Bieniusa et al., 2012].

It should also be noted that if the messages are delivered in FIFO order, we can coalesce the adds in a manner similar to the case when $k = 1$ in Algorithm 4. Thus, the space complexity of the E set at every replica would be $O(n_a N \log(n_t))$. However, FIFO does not guarantee any bound on the the number of intervals in each component of the interval-version-vector as illustrated in example 73.

Example 73. *Consider an OR-Set with three replicas r, r' and r''. Suppose r gets unboundedly many add requests of which every alternate add request corresponds to that of element x and every other add request correspond to that of distinct elements of the universe. Thus the E set at r would be of the form*

$$\{(x, 0, r), (y, 1, r), (x, 2, r), (z, 3, r), (x, 4, r), \ldots\}$$

Suppose further that all the add operations from r get propagated to replica r', but none of those operations are yet propagated to replica r''. This is allowed in FIFO ordering. Now, replica r' gets delete requests corresponding to all the non-x elements that have been added by replica r. Suppose further that the receive operations of these deletes are delivered at r'' in FIFO order. Thus at r'', $V[r] = \{[1, 1], [3, 3], [5, 5], \ldots\}$. Hence the number of intervals can be unbounded.

In the case where message delivery is constrained by FIFO ordering, the worst case space complexity in would be $O((n_a + n_{int})N \log(n_t))$ which is comparable to space complexity of the generalized algorithm without ordering constraints since $O(n_{int})$ can be as large as $O(n_t)$ as highlighted in the example above.

3.8 Summary

In this chapter, we have presented an optimized OR-Set implementation that does not depend on the order in which updates are delivered. The worst-case space complexity is comparable

Algorithm 4 An optimized OR-Set implementation with k-causal-delivery

Optimized OR-set implementation for the replica r with k-causal delivery constraint

```
1  E ⊆ M, V : R → I, c ∈ ℕ: initially ∅, [∅, ..., ∅], 0
2
3  Boolean CONTAINS(x ∈ Univ):
4      return (∃m : m ∈ E ∧ data(m) = x)
5
6  ADD(x ∈ Univ):
7      Let m := ADD.PREPARE(x)
8      Call ADD.APPLY(m)
9      Broadcast ADD.send(m)
10 ADD.PREPARE(x ∈ Univ):
11     return (x, c, r)
12 ADD.APPLY(m ∈ M):
13     M = {m' | data(m') = data(m) ∧
                 rep(m') = rep(m) ∧
                 ts(m') ≤ ts(m) − k}
14     E = E \ M
15     if (ts(m) ∉ V[rep(m)])
16         E := E ∪ {m}
17         V[rep(m)] :=
               add(V[rep(m)], {ts(m)})
18     if (rep(m) = r)
19         c = ts(m) + 1
20 Receive ADD.receive(m ∈ M):
21     Call ADD.APPLY(m)
22
23 DELETE(x ∈ Univ):
24     Let V' := DELETE.PREPARE(x)
25     Call DELETE.APPLY(x, V')
26     Broadcast DELETE.send(x, V')
27 DELETE.PREPARE(x ∈ Univ):
28     Let V' : R → I = [0, ..., 0]
29     for m ∈ E with data(m) = x
30         add(V'[rep(m)], {ts(m)})
31     return V'
32 DELETE.APPLY(x ∈ Univ,
33                V' : R → I):
34     Let M = {m ∈ E |
                 ts(m) ∈ V'[rep(m)]∨
                 (data(m) = x ∧
                 ∃c ∈ V'[rep(m)] ts(m) ≤ c − k)}
35     E := E \ M
36     ∀i ∈ R.(V[i] := V[i] ∪ V'[i])
37 Receive DELETE.receive(x ∈ Univ, V' : R → I):
38     Call DELETE.APPLY(x, V')
39
40
41 Boolean COMPARE(S', S'' ∈ S):
42     Assume that S' = (E', V')
43     Assume that S'' = (E'', V'')
44     b_seen := ∀i(V'[i] ⊆ V''[i])
45     b_deletes := ∀m ∈ E'' \ E'
                (ts(m) ∉ V'[rep(m)])
46         // If m is deleted from E' then
47         // it is also deleted in E''.
48         // So anything in E'' \ E'
49         // is not even visible in S'.
50     return b_seen ∧ b_deletes
51
52 MERGE(S' ∈ S):
53     Assume that S' = (E', V')
54     E := {m ∈ E ∪ E' |
              m ∈ E ∩ E'∨
              ts(m) ∉ V[rep(m)] ∩ V'[rep(m)]}
55 // You retain m if it is either
56 // in the intersection, or if it is fresh
57 // (so one of the states has not seen it).
58 ∀i.(V[i] := V[i] ∪ V'[i])
```

to the original implementation [Shapiro et al., 2011a] and the best-case complexity is the same as that of the solution proposed in [Bieniusa et al., 2012].

The solution in [Bieniusa et al., 2012] requires causal ordering over all updates. As we have argued, this is an unreasonably strong requirement. On the other hand, there seems to be no simple relaxation of causal ordering that retains the structure required by the simpler algorithm of [Bieniusa et al., 2012]. Our new generalized algorithm can accommodate any specific ordering constraint that is guaranteed by the delivery subsystem. Moreover, our solution has led us to identify *k-causal ordering* as a natural generalization of causal ordering, where the parameter k directly captures the impact of out-of-order delivery on the space requirement for bookkeeping.

Our optimized algorithm uses interval version vectors to keep track of the elements that have already been seen. It is known that regular version vectors have a bounded representation when the replicas communicate using pairwise synchronization [Almeida et al., 2004]. An alternative proof of this in [Mukund et al., 2020] is based on the solution to the *gossip problem* for synchronous communication [Mukund and Sohoni, 1997], which has also been generalized to message-passing systems [Mukund et al., 2003]. It would be interesting to see if these ideas can be used to maintain interval version vectors using a bounded representation. This is not obvious because intervals rely on the linear order between timestamps and reusing timestamps typically disrupts this linear order.

Another direction that can be explored is to characterize the class of datatypes with non-commutative operations for which a CRDT implementation can be obtained using interval version vectors.

We end this chapter by noting that the OR-Set is also supported by real-world data stores such as RiakDB and Redis. RiakDB supports a variant of OR-Set known as ORSWOT where the delete() operation can be associated with a context [Owen, 2015]. The Redis datastore has support for the classical OR-Set which guaratees the *add-wins* semantics [Redis, 2020].

4

Declarative Specification for CRDTs

4.1 Introduction

It is well understood that in a serialized setting, where operations are performed in a sequential order, the properties of data types can be described independent of their implementations. For instance, in a serialized setting, in the case of the stack, the expectation is that the *pop()* method should return the value that was the argument to the most recent *push()*. Similarly, in the case of a Queue, the result of a *dequeue()* operation should correspond to the value enqueued by the earliest unmatched *enqueue()* operation. Thus, every implementation of a stack or a queue would have these properties. This behavioural description of the data type independent of their implementation is known as its *abstract specification*. In the previous chapter, we saw that for the OR-Set CRDT, there were three different implementations which all adhered to the same specification. Such a specification plays a crucial role in formal verification of the correctness of any implementation of the data type. However, in their initial days, most of the early work on replicated data types described these data types through implementations [Shapiro et al., 2011a,b; Bieniusa et al., 2012; Mukund et al., 2014].

In the serialized setting, the specification of a data type can be described in terms of its observable behavior, which is a sequence of operations. For instance in the case of the stack, where the specification says that the return value of a *pop()* should be the same as the *latest* value that was pushed, the notion of *latest push* is unambiguous since the sequence of operations is a total order. However such a serialized specification cannot be applied in a distributed setting, where there are concurrent operations and the data types guarantee weaker consistency criteria such as *eventual consistency*. Hence there was a need for a formal framework in which declarative specification of a wide variety of distributed data-types could be described. Such a framework was first proposed in [Burkhardt et al., 2014]. This work also provided a methodology to prove the correctness of an implementation via replication-aware

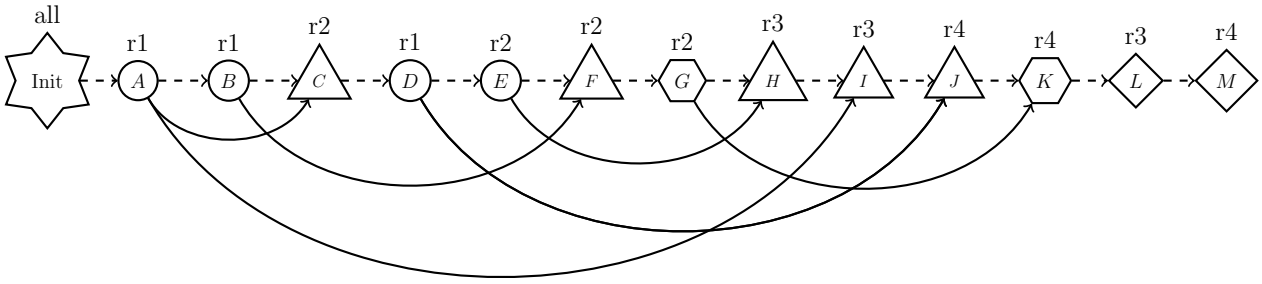


Figure 4.1: An example Run. Here, we denote the *update* operations *A, B, D, E* as circles, the corresponding *update-receive* operations *C, F, H, I, J* as triangles, the *merge-send* and *merge-receive* operations as hexagons and the *query* operations *L, M* as diamonds. The source replicas of the operations are indicated above the shape representing the operation. Furthermore, the solid arrows associate every *update-receive* operation with the corresponding *update* operation and also associates every *merge-receive* operation with the corresponding *merge-send* operation.

simulations.

In this chapter we shall present a simplified version of the framework from [Burkhardt et al., 2014]. Our simplified framework is based on our work [Mukund et al., 2015a] that describes the behaviours of replicated data types using the concept of standard *labelled partial orders* which were introduced in [Mazurkiewicz, 1987; Pratt, 1986] to describe the properties of asynchronous communicating automata. This simplified framework is sufficient for providing the abstract specification of the existing replicated data types. Our motivation for using a simpler framework to synthesize a reference implementation of the replicated data type from its abstract specification. This reference implementation will serve as a foundation in the next chapter where we will adapt some of the results, originally proved in the context of asynchronous automata, to replicated data types in order to obtain a bounded reference implementation of a replicated data type from its specification. This bounded reference implementation can be used towards the effective verification of a given implementation of the replicated data type.

4.2 Definitions for Declarative Specifications

In this section we introduce a framework for providing declarative-specification for replicated data-types. The notation followed here is an amalgamation of the notation used in [Burckhardt, 2014] and [Mukund et al., 2015a].

Consider the example of a run in Figure 4.1.For an external observer, this is the sequence of operations that was performed by the implementation of the replicated data type. However since the implementation contains several independent communicating replicas, the observable behaviour of the data type depends on the sequence operations observed by the individual replicas. This is because when a user issues a query request, the source replica of that query provide a response based on its current local state. Thus, in order to reason about the response provided by the replica to a query, we only need to consider the operations that have impacted that replica. In the example in Figure 4.1, in order to answer the query L at replica $r3$, we need to reason about the impact of the operations $\{A, E, H, I\}$ on $r3$. While in order to answer the query M at replica $r4$, we need to reason about the operations that have impacted the replica $r4$. This set includes $\{A, B, C, D, E, F, G, J, K\}$.

The set of operations that impact a replica at any given point in the run can be modeled using the notion of event structures [Mazurkiewicz, 1987]. Here, where we model every operation as an event and record the relationships between these events.

We first associate an event with each operation of a run.

Definition 74 (Events)**.** *Let $\alpha = (\rho, \varphi)$ be a run of a replicated data type. We define $\mathcal{E}_\rho$ to be the set of* events *associated with the operations in ρ.*

$$\mathcal{E}_\rho = \{e_i \mid 0 \leq i < |\rho|\}$$

. Let e_i be the event associated with $\rho[i]$ then, for $F() \in \{Rep(), Op(), Ret(), Args()\}$ we let $F(e_i) = F(\rho[i])$.

For an event $e_i \in \mathcal{E}_\rho$ we say that

- *e_i is a query event if $Op(e_i) \in$* Queries
- *e_i is an update event if $Op(e_i) \in$* Updates
- *e_i is an update-send event if $Op(e_i) =$* **usend**
- *e_i is an update-receive event if $Op(e_i) =$* **ureceive**
- *e_i is an merge-send event if $Op(e_i) =$* **msend**
- *e_i is an merge-receive event if $Op(e_i) =$* **mreceive**

*If e_i is an event with $Op(e_i) \in \{$***ureceive**, **mreceive***$\}$ associated with the operation $\rho[i]$ then we define the matching update event(if e_i is an* **ureceive** *event) or the matching-*__msend__ *event (if e_i is an* **mreceive** *event) denoted by $\varphi(e_i)$ to be*

$$\varphi(e_i) = e_j \iff \text{in the run } \alpha = (\rho, \varphi), \text{ we have } \varphi(i) = j.$$

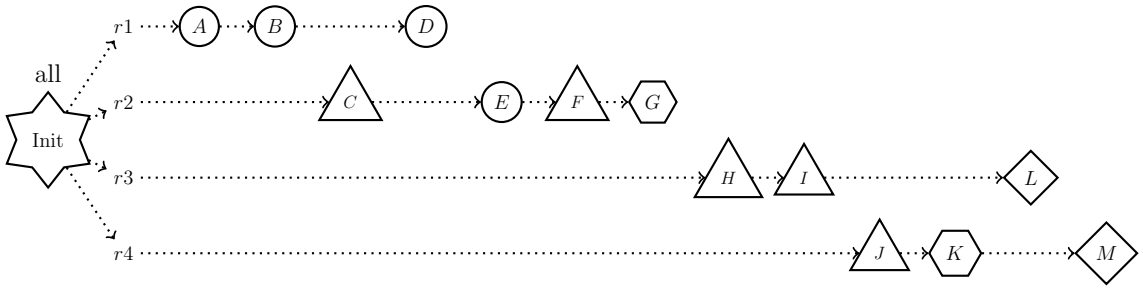


Figure 4.2: Replica Order : Sequence of operations seen by each replica

If e_i is an update event associated with $\rho[i]$ then the corresponding sent of update-receive events is denoted by $\varphi^{-1}(e_i)$ and defined to be the set

$$\varphi^{-1}(e_i) = \{e_j \mid \varphi(e_j) = e_i\}$$

If e_i is an update event at a replica which is distinct from another replica r, then $\varphi_r^{-1}(e_i)$ denotes the **ureceive** *event e_j at replica r whose matching update is e_i. Formally,*

$$\varphi_r^{-1}(e_i) = e_j \iff Rep(e_j) = r \wedge \varphi(e_j) = e_i$$

If e_i, e_j are a pair of update events, we say e_i happened-before *e_j, written as $e_i \xrightarrow{\mathsf{hb}} e_j$ iff $\rho[i] \xrightarrow{\mathsf{hb}} \rho[j]$.*

Each replica in the system sees operations applied at them in a particular order. From the example in Figure 4.1, the events at replica $r1$ is $\{A, B, D\}$, while at replica $r2$ is $\{C, E, F, G\}$ respectively, in that order. This is shown in Figure 4.2. We model the total order of operations performed at each replica as the *replica order*.

We capture this notion via the following definition.

Definition 75 (Replica Order)**.** *Let $\mathcal{E}_\rho$ be the events associated with a run $\alpha = (\rho, \varphi)$ of a replicated data type.*

Let $\mathcal{E}_\rho^r = \{e_i \in \mathcal{E}_\rho \mid Rep(e) = r\}$.

We define the replica order for r*, denoted by $\xrightarrow{r}$, to be the order in which the events occur at r. Formally,*

$$\xrightarrow{r} = \{(e_i, e_j) \mid e_i, e_j \in \mathcal{E}_\rho^r \wedge i \leq j\}$$

Note that $\xrightarrow{r}$ is a total order.

Then, we define the replica-order *of a run, denoted by $\xrightarrow{\mathcal{R}}$, to be the union of the replica orders of all the replicas.*

$$\xrightarrow{\mathcal{R}} = \bigcup_{r \in \mathcal{R}} \xrightarrow{r}$$

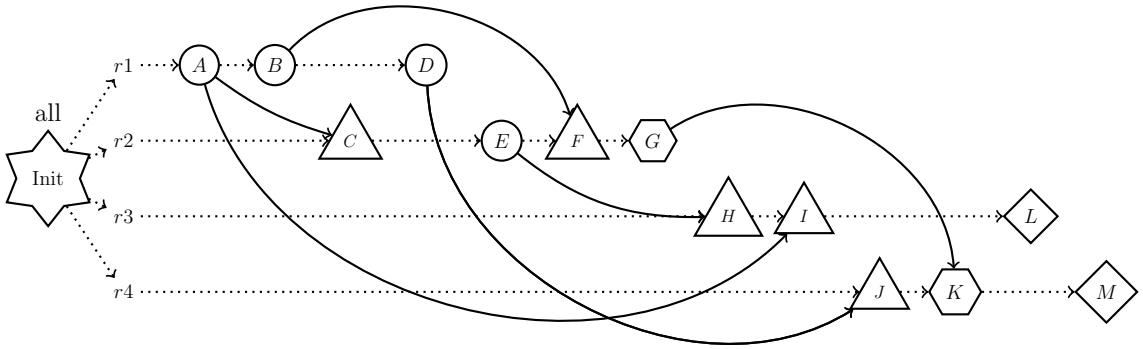


Figure 4.3: Broadcast and Merge Orders : Between communicating replicas.

The replicas communicate with each other via broadcast messages after every update operation (in the case of operation-based replicated data types) and merge operations (in the case of state-based replicated data types). In the run from Figure 4.1, the update A at replica $r1$ is communicated to replicas $r2$ and $r3$ via update receive methods C and I respectively. Similarly replica $r2$ sends its state to replica $r4$ via *merge-send* operation G which is received by $r4$ as the *merge-receive* operation K.

We can model these communication in our event structure framework as broadcast-orders and merge-orders over the events of a run (Figure 4.3). We formally define these below.

Definition 76 (Broadcast Order)**.** *Let $\mathcal{E}_\rho$ be the events associated with some run $\alpha = (\rho, \varphi)$.*

Then we define the broadcast-order*, denoted by $\xrightarrow{\text{broadcast}}$, to be the the set associating an* update *operation with its corresponding* update-receive *operations. Formally,*

$$\xrightarrow{\text{broadcast}} = \{(\varphi(e_i), e_i) \mid e_i \in \mathcal{E}_\rho \wedge Op(e) = \textbf{ureceive}\}$$

Definition 77 (Merge Order)**.** *Let $\mathcal{E}$ be the events associated with some run $\alpha = (\rho, \varphi)$. Then we define the* merge-order*, denoted by $\xrightarrow{\text{merge}}$, to be the set associating a merge-send with the corresponding merge receive.*

$$\xrightarrow{\text{merge}} = \{(\varphi(e_i), e_i) \mid e_i \in \mathcal{E}_\rho \wedge Op(e) = \textbf{mreceive}\}$$

The state of a replica at any point in the run is determined by the set of operations that are *known* to the replica at that point in the run. Note at any point in the run, a replica knows of

- The operations whose source replica was itself.

- Remote update operations whose broadcast was received by the replica.
- Remote operations of the run whose effect was merged into the replica through a merge receive operation.

In the example run from Figure 4.1, the replica $r4$ at the end of the query operation M knows of

- The locally performed *update-receive* operation J and the *merge-receive* operation K along with the query operation M itself.
- The remote update D from replica $r1$ whose information was received through the *update-receive* operation J.
- The *merge-send* operation G from replica $r2$ and the operations $\{A, B, C, E, F, G\}$ which were known to replica $r2$ at G. These information about these operations reaches r via the *merge-receive* operation K.

To reason about the state of replica $r4$ at G, it is sufficient to consider the set of operations known to it and the relationships between these operations. It is immaterial if $r4$ receives the knowledge of these operations either because they were locally performed or via an *update-receive* or a *merge-receive*. Thus, we capture this flow of knowledge of operations in our event structure as a *visibility relation* over the events of the run.

Definition 78 (Visibility Relation)**.** *Let $\alpha = (\rho, \varphi)$ be a run and let $\mathcal{E}_\rho$ denote the set of events corresponding to the operations in ρ.*

We define the visibility relation *over $\mathcal{E}_\rho$ for the run α, denoted by* vis*, to be the smallest acyclic relation over $\mathcal{E}$ satisfying the following:*

1. $\xrightarrow{\mathcal{R}} \cup \xrightarrow{\mathsf{broadcast}} \cup \xrightarrow{\mathsf{merge}} \subseteq \mathsf{vis}$
2. $\mathsf{vis}; \xrightarrow{\mathcal{R}} \subseteq \mathsf{vis}$
3. $\mathsf{vis}; \xrightarrow{\mathsf{merge}} \subseteq \mathsf{vis}$.

where $X; Y$ denotes the composition of a pair of binary relations X and Y

$$X; Y = \{(a, c) \mid (a, b) \in X \wedge (b, c) \in Y\}$$

The visibility relation captures the set of events of the run that any particular replica is aware of during any point in the run. Thus, if an event e_i is visible to e_j, denoted by $e_i \xrightarrow{\mathsf{vis}} e_j$, it implies that the source replica of the event e_j is aware of the event e_i at that point in time. In the definition above,

Condition 1 insists that the visibility relation should contain the replica order, the broadcast order and the merge order. Thus the replica continues to be aware of any operation in the run that it had previously locally performed. The replica becomes aware of a remote update when it receives the corresponding broadcast. The replica becomes aware of a merge request sent by a remote replica when it receives that merge request.

Condition 2 insists that once some remote update is visible to a replica, it continues to remain visible to all the subsequent operations at the replica. This is equivalent of the Monotonic-Reads constraint defined in [Burckhardt, 2014].

Condition 3 ensures that once a replica receives a merge request from any other replica, all the operations that are visible to the sending replica at the time of the merge-send are visible to the recepient of the merge.

Thus, given a run, the event-set along with the visibility relation over the events in that set is sufficient to reason about the state of the replicas at different points in that run. This event-structure consisting of the event set along with the visibility relation is defined to be the *trace* of a run. Using the abstraction of traces, we can declarative specifications of replicated data-types [Burckhardt, 2014].

Definition 79 (Trace of a Run)**.** *Let $\alpha = (\rho, \varphi)$ be a valid run of a replicated data-type. Let $\mathcal{E}_\rho$ be the set of events associated with ρ and vis be the visiblity relation of α.*

Then we define the trace of the run *α to be the tuple $\mathcal{T}(\alpha) = (\mathcal{E}_\rho, \mathsf{vis})$.*

The set of all the traces of all the runs of a replicated data type $\mathcal{D}$, denoted by $\mathcal{T}(\mathcal{D})$ is defined to be the set $\bigcup_{\alpha \in Runs(\mathcal{D})} \mathcal{T}(\alpha)$.

Figure 4.4 shows the trace of a run from Figure 4.1. In this we have a single relationship between the events of the operations of the run, namely the visibility relation.

Note: In the trace of a run, when an event (say B) becomes visible to another event (say K) it remains visible to all the future events of the same replica (here L). In Figure 4.4 we have omitted those edges for the sake of clarity.

The trace of a run consists of an event set where there is a representative event for every operation in the run. In addition there is a visibility relation which describes the relationship between these events. We will now show that the visibility relation over the events a run is uniquely computable, thereby showing that every run has a unique computable trace.

Proposition 80. *Let $\alpha = (\rho, \varphi)$ be a run and let $\mathcal{T}(\alpha) = (\mathcal{E}_\rho, \mathsf{vis})$ be the trace of α. Let $\xrightarrow{\mathcal{R}}$, $\xrightarrow{\mathsf{broadcast}}$ and $\xrightarrow{\mathsf{merge}}$ be relations over $\mathcal{E}_\rho$ as defined earlier.*

Let $\mathsf{vis}_0 = \xrightarrow{\mathcal{R}} \cup \xrightarrow{\mathsf{broadcast}} \cup \xrightarrow{\mathsf{merge}}$.

For $i > 0$ Let $\mathsf{vis}_i = \mathsf{vis}_{i-1} \cup (\mathsf{vis}_{i-1}; \xrightarrow{\mathcal{R}}) \cup (\mathsf{vis}_{i-1}; \xrightarrow{\mathsf{merge}})$

Then, the visiblity relation over $\mathcal{E}_\rho$ for α is $\mathsf{vis} = \mathsf{vis}_k$ for the smallest $k \in \mathbb{N}$ such that for all $k' > k$, $\mathsf{vis}_{k'} = \mathsf{vis}_k$.

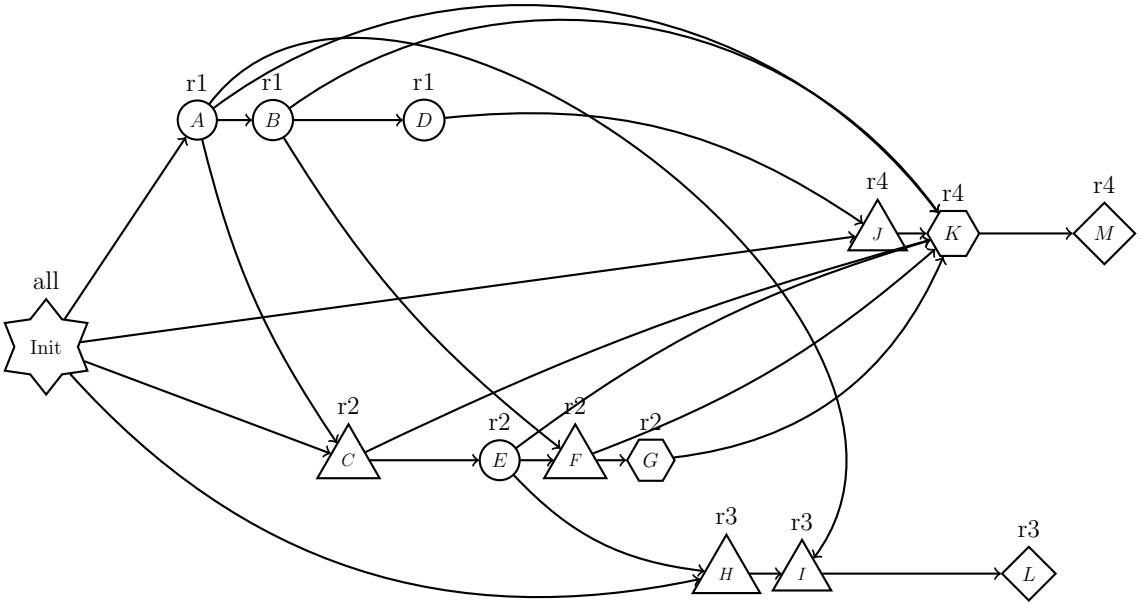


Figure 4.4: Trace of a Run from Figure 4.1

Proof. Note that $\mathsf{vis}_0 \subseteq \mathsf{vis}_1 \subseteq \cdots \subseteq (\mathcal{E}_\rho \times \mathcal{E}_\rho)$.

Since α is a finite run, $\mathcal{E}_\rho$ is a finite set of events and thus, $(\mathcal{E}_\rho \times \mathcal{E}_\rho)$ is finite.

From this, it follows that there will be a k such that for all $k' > k$, $\mathsf{vis}_{k'} = \mathsf{vis}_k$.

We will show that $\mathsf{vis} \subseteq \mathsf{vis}_k$. Towards that, we will first prove that vis_k satisfies the constraints that need to be satisfied by the visibility relation.

Note that since $\mathsf{vis}_0 = \xrightarrow{\mathcal{R}} \cup \xrightarrow{\mathsf{broadcast}} \cup \xrightarrow{\mathsf{merge}} \subseteq \mathsf{vis}_k$, the first constraint is satisfied.

Now, by definition, $\mathsf{vis}_{k+1} = \mathsf{vis}_k \cup (\mathsf{vis}_k; \xrightarrow{\mathcal{R}}) \cup (\mathsf{vis}_k; \xrightarrow{\mathsf{merge}})$. Hoewever since $\mathsf{vis}_{k'} = \mathsf{vis}_k$ for all $k' > k$, it follows that, $\mathsf{vis}_{k+1} = \mathsf{vis}_k \cup (\mathsf{vis}_k; \xrightarrow{\mathcal{R}}) \cup (\mathsf{vis}_k; \xrightarrow{\mathsf{merge}}) = \mathsf{vis}_k$.

From this we can see that $(\mathsf{vis}_k; \xrightarrow{\mathcal{R}}) \subseteq \mathsf{vis}_k$ and $(\mathsf{vis}_k; \xrightarrow{\mathsf{merge}}) \subseteq \mathsf{vis}_k$.

Thus, the second and the third constraints are satisfied.

We now need to show if vis is the visibility relation, then, $\mathsf{vis}_k = \mathsf{vis}$.

Note that since by definition the visibility relation vis is the minimal relation satisfying the three constraints, we have $\mathsf{vis} \subseteq \mathsf{vis}_k$.

If we show that $\mathsf{vis}_k \subseteq \mathsf{vis}$, it follows that $\mathsf{vis} = \mathsf{vis}_k$.

We shall show that each $i \geq 0$, $\mathsf{vis}_i \subseteq \mathsf{vis}$.

For $i = 0, \mathsf{vis}_0 = \xrightarrow{\mathcal{R}} \cup \xrightarrow{\mathsf{broadcast}} \cup \xrightarrow{\mathsf{merge}} \subseteq \mathsf{vis}$ Hence, the result holds for $i = 0$. Let us assume that the result holds for all $i < n$.

By definition, $\mathsf{vis}_n = \mathsf{vis}_{n-1} \cup (\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}}) \cup (\mathsf{vis}_{n-1}; \xrightarrow{\mathsf{merge}})$

- By induction hypothesis, $\mathsf{vis}_{n-1} \subseteq \mathsf{vis}$.
- Since $\mathsf{vis}_{n-1} \subseteq \mathsf{vis}$, it can be seen that $(\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}}) \subseteq (\mathsf{vis}; \xrightarrow{\mathcal{R}})$. Since vis satisfies the second constraint, we have $(\mathsf{vis}; \xrightarrow{\mathcal{R}}) \subseteq \mathsf{vis}$. Thus $(\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}}) \subseteq \mathsf{vis}$
- Since $\mathsf{vis}_{n-1} \subseteq \mathsf{vis}$, It can be seen that $(\mathsf{vis}_{n-1}; \xrightarrow{\mathsf{merge}}) \subseteq (\mathsf{vis}; \xrightarrow{\mathsf{merge}})$. Since vis satisfies the third constraint, we have $(\mathsf{vis}; \xrightarrow{\mathsf{merge}}) \subseteq \mathsf{vis}$. Thus $(\mathsf{vis}_{n-1}; \xrightarrow{\mathsf{merge}}) \subseteq \mathsf{vis}$

From this, it can be seen that $\mathsf{vis}_n \subseteq \mathsf{vis}$.

Thus, by the principle of mathematical induction, for all $i \geq 0$, $\mathsf{vis}_i \subseteq \mathsf{vis}$. In particular, $\mathsf{vis}_k \subseteq \mathsf{vis}$.

Thus, since $\mathsf{vis} \subseteq \mathsf{vis}_k$ and $\mathsf{vis}_k \subseteq \mathsf{vis}$ it follows that $\mathsf{vis} = \mathsf{vis}_k$.

Thus, the visibility relation for a given run is uniquely computable. □

We can intuit that when a run of a replicated data type is extended with newer operations, the events corresponding to these newer operations in the trace of the extended run do not become visible to the events of the trace of the original run. Nor do the events which were visible to some event in the trace of the original run cease to become visible to that event in the trace of the extended run. Thus for a given event in a trace of a run, the set of events that are visible to it remains fixed in the trace of any extension of the run. We formally prove this through this next proposition.

Proposition 81. *Let $\alpha = (\rho, \varphi)$ be a run. Let $\alpha' = (\rho', \varphi')$ be a prefix of α with $\rho' = \rho[1, \ldots, i]$ and $\varphi' = \varphi|_{\rho'}$ for some $i < |\rho|$. Let $(\mathcal{E}_\rho, \mathsf{vis})$ be the trace of α and let $(\mathcal{E}_{\rho'}, \mathsf{vis}')$ be the trace over α'.*

Then, $\mathsf{vis}' = \mathsf{vis}|_{\mathcal{E}_{\rho'}}$

Proof. Let $\xrightarrow{\mathcal{R}}{}'$ and $\xrightarrow{\mathcal{R}}$ denote the replica order over α' and α respectively.

Let $\xrightarrow{\mathsf{broadcast}}{}'$ and $\xrightarrow{\mathsf{broadcast}}$ respectively denote the broadcast order for α' and α respectively.

Let $\xrightarrow{\mathsf{merge}}{}'$ and $\xrightarrow{\mathsf{merge}}$ denote the merge order for α' and α respectively.

It can be seen that $\xrightarrow{\mathcal{R}}{}' = \xrightarrow{\mathcal{R}}|_{\mathcal{E}_{\rho'}}$, $\xrightarrow{\mathsf{broadcast}}{}' = \xrightarrow{\mathsf{broadcast}}|_{\mathcal{E}_{\rho'}}$ and $\xrightarrow{\mathsf{merge}}{}' = \xrightarrow{\mathsf{merge}}|_{\mathcal{E}_{\rho'}}$.

From Proposition 80, vis' and vis can be uniquely computed through an iterative procedure.

Let vis'_j and vis_j denote the subsets of vis' and vis respectively after the j^{th} iterations.

We shall show that for each $j \geq 0$, $\mathsf{vis}'_j = \mathsf{vis}_j|_{\mathcal{E}_{\rho'}}$.

$\mathsf{vis}'_0 = \xrightarrow{\mathcal{R}}{}' \cup \xrightarrow{\mathsf{broadcast}}{}' \cup \xrightarrow{\mathsf{merge}}{}' = (\xrightarrow{\mathcal{R}}|_{\mathcal{E}_{\rho'}}) \cup (\xrightarrow{\mathsf{broadcast}}|_{\mathcal{E}_{\rho'}}) \cup (\xrightarrow{\mathsf{merge}}|_{\mathcal{E}_{\rho'}}) = (\xrightarrow{\mathcal{R}} \cup \xrightarrow{\mathsf{broadcast}} \cup \xrightarrow{\mathsf{merge}})|_{\mathcal{E}_{\rho'}} = \mathsf{vis}_0|_{\mathcal{E}_{\rho'}}$. Thus, the result holds for $j = 0$.

Assume that the result holds for all $j < n$.

By definition, $\mathsf{vis}'_n = \mathsf{vis}'_{n-1} \cup (\mathsf{vis}'_{n-1}; \xrightarrow{\mathcal{R}}{}') \cup (\mathsf{vis}'_{n-1}; \xrightarrow{\mathsf{merge}}{}')$. By inductive hypothesis and the definitions of $\xrightarrow{\mathcal{R}}{}'$ and $\xrightarrow{\mathsf{merge}}{}'$, we can rewrite this as

$\mathsf{vis}'_n = (\mathsf{vis}_{n-1}|_{\mathcal{E}_{\rho'}}) \cup ((\mathsf{vis}_{n-1}|_{\mathcal{E}_{\rho'}}); (\xrightarrow{\mathcal{R}}|_{\mathcal{E}_{\rho'}})) \cup ((\mathsf{vis}_{n-1}|_{\mathcal{E}_{\rho'}}); (\xrightarrow{\mathsf{merge}}|_{\mathcal{E}_{\rho'}}))$.

Now, it can be seen that $((\mathsf{vis}_{n-1}|_{\mathcal{E}_{\rho'}}); (\xrightarrow{\mathcal{R}}|_{\mathcal{E}_{\rho'}})) \subseteq (\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}})|_{\mathcal{E}_{\rho'}}$.

Suppose $(e_k, e_{k'}) \in (\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}})|_{\mathcal{E}_{\rho'}}$. Then, $e_k, e_{k'} \in \mathcal{E}_{\rho'}$. Since $\rho' = \rho[1, \ldots, i]$, it follows that $k, k' \leq i$. Again, since $(e_k, e_{k'}) \in (\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}})$, here exists a k'' such that $(e_k, e_{k''}) \in \mathsf{vis}_{n-1}$ and $(e_{k''}, e_{k'}) \in \xrightarrow{\mathcal{R}}$. Since $(e_{k''}, e_{k'}) \in \xrightarrow{\mathcal{R}}$, it follows that $k'' < k'$. Thus, $k'' < i$ which implies that $e_{k''} \in \mathcal{E}_{\rho'}$.

Thus, $(e_k, e_{k'}) \in (\mathsf{vis}_{n-1}|_{\mathcal{E}_{\rho'}}); (\xrightarrow{\mathcal{R}}|_{\mathcal{E}_{\rho'}})$.

Since e_k and $e_{k'}$ were arbitrary events such that, $(e_k, e_{k'}) \in (\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}})|_{\mathcal{E}_{\rho'}}$, it follows that $(\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}})|_{\mathcal{E}_{\rho'}} \subseteq (\mathsf{vis}_{n-1}|_{\mathcal{E}_{\rho'}}); (\xrightarrow{\mathcal{R}}|_{\mathcal{E}_{\rho'}})$.

Thus, we have $(\mathsf{vis}_{n-1}|_{\mathcal{E}_{\rho'}}); (\xrightarrow{\mathcal{R}}|_{\mathcal{E}_{\rho'}}) = (\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}})|_{\mathcal{E}_{\rho'}}$.

Using similar line of reasoning, replacing $\xrightarrow{\mathcal{R}}$ with $\xrightarrow{\mathsf{merge}}$, we can show that

$(\mathsf{vis}_{n-1}|_{\mathcal{E}_{\rho'}}); (\xrightarrow{\mathsf{merge}}|_{\mathcal{E}_{\rho'}}) = (\mathsf{vis}_{n-1}; \xrightarrow{\mathsf{merge}})|_{\mathcal{E}_{\rho'}}$.

From this, we can see that $\mathsf{vis}'_n = (\mathsf{vis}_{n-1}|_{\mathcal{E}_{\rho'}}) \cup (\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}})|_{\mathcal{E}_{\rho'}} \cup (\mathsf{vis}_{n-1}; \xrightarrow{\mathsf{merge}})|_{\mathcal{E}_{\rho'}}$.

From this, $\mathsf{vis}'_n = (\mathsf{vis}_{n-1} \cup (\mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}}) \cup (\mathsf{vis}_{n-1}; \xrightarrow{\mathsf{merge}}))|_{\mathcal{E}_{\rho'}} = \mathsf{vis}_n|_{\mathcal{E}_{\rho'}}$.

Thus it follows that, for all $j \geq 0$, $\mathsf{vis}'_j = \mathsf{vis}_j|_{\mathcal{E}_{\rho'}}$ which implies that $\mathsf{vis}' = \mathsf{vis}|_{\mathcal{E}_{\rho'}}$. □

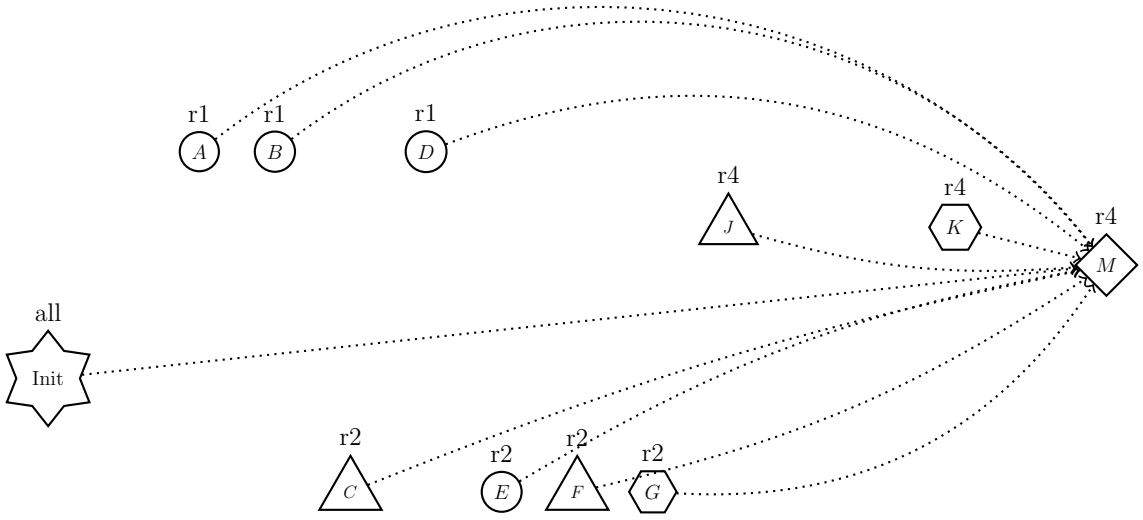


Figure 4.5: Visible event set of event M

The set of events visible to any replcia at any point in the round would involve identifying the latest event of the replica at that point in the run and determining the set of events that are visible to that latest event. We formally define this set of events as the *visible events set* of that event. For example, the Figure 4.5 shows the visible event set of the replica $r4$ at the end of the operation M. This includes the operations $\{Init, A, B, C, D, E, F, G, J, K, M\}$.

Definition 82 (Visible Event Set)**.** *Given the trace* $(\mathcal{E}_\rho, \mathsf{vis})$ *of a run* $\alpha = (\rho, \varphi)$*, for any event* $e_i \in \mathcal{E}_\rho$*, we define the visible event set of* e_i*, denoted by* $\mathsf{vis}^{-1}(e_i)$*, to be the set of all the events visible to* e_i*, i.e* $\mathsf{vis}^{-1}(e_i) = \{e_j \in \mathcal{E}_\rho \mid e_j \xrightarrow{\mathsf{vis}} e_i\}$.

We note that when a run is extended with a new operation, the visible event set of the event corresponding to the new operation includes the visible event set of the predecessors of that event. We now show how the visible event set of an event can be computed from the visible event sets of its immediate predecessors in the visibility relation. (**Note**: An event can have more than one immediate predecessors as per the visibility relation. In the case of an *update-receive* event, its immediate predecessors would be the latest event of that same replica prior to the *update-receive* event, and the remote *update* event corresponding to the *update-receive* event. Same is the case with a *merge-receive* event. All other types of events will have a unique predecessor, being the latest event on the source replica prior to them).

Proposition 83. *Let* $\alpha = (\rho, \varphi)$ *be a run. Let* i *and* j *be integers such that* $j < i \leq |\rho|$ *and* $Rep(\rho[i]) = Rep(\rho[j])$ *and for all* $k : j < k < i$*,* $Rep(\rho[k]) \neq Rep(\rho[i])$*. Then,*

$$\mathsf{vis}^{-1}(e_i) = \begin{cases} \mathsf{vis}^{-1}(e_j) \cup \{e_i\} & \textit{if } Op(e_i) \in \mathsf{Queries} \cup \mathsf{Updates} \cup \{\mathbf{usend}, \mathbf{msend}\} \\ \mathsf{vis}^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\} & \textit{if } Op(e_i) = \mathbf{ureceive} \\ \mathsf{vis}^{-1}(e_j) \cup \mathsf{vis}^{-1}(\varphi(e_i)) \cup \{e_i\} & \textit{if } Op(e_i) = \mathbf{mreceive} \end{cases}$$

Proof. We shall prove the following direction of containment first

$$\mathsf{vis}^{-1}(e_i) \supseteq \begin{cases} \mathsf{vis}^{-1}(e_j) \cup \{e_i\} & \text{if } Op(e_i) \in \mathsf{Queries} \cup \mathsf{Updates} \cup \{\mathbf{usend}, \mathbf{msend}\} \\ \mathsf{vis}^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\} & \text{if } Op(e_i) = \mathbf{ureceive} \\ \mathsf{vis}^{-1}(e_j) \cup \mathsf{vis}^{-1}(\varphi(e_i)) \cup \{e_i\} & \text{if } Op(e_i) = \mathbf{mreceive} \end{cases}$$

We consider the following cases

$Op(\rho[i]) \in \mathsf{Queries} \cup \mathsf{Updates} \cup \{\mathbf{usend}, \mathbf{msend}\}$**:** Note that if some $e_k \in \mathsf{vis}^{-1}(e_j)$ then since $e_j \xrightarrow{Rep(e_i)} e_i$ and $\mathsf{vis}; \xrightarrow{\mathcal{R}} \subseteq \mathsf{vis}$, we have $e_k \in \mathsf{vis}^{-1}(e_i)$. By defintion, $e_i \in \mathsf{vis}^{-1}(e_i)$. Thus, $\mathsf{vis}^{-1}(e_j) \cup \{e_i\} \subseteq \mathsf{vis}^{-1}(e_i)$.

$Op(\rho[i]) = \mathbf{ureceive}$**:** If $e_k \in \mathsf{vis}^{-1}(e_j)$ then since $e_j \xrightarrow{Rep(e_i)} e_i$ and $\mathsf{vis}; \xrightarrow{\mathcal{R}} \subseteq \mathsf{vis}$, we have $e_k \in \mathsf{vis}^{-1}(e_i)$ Thus $\mathsf{vis}^{-1}(e_j) \subseteq \mathsf{vis}^{-1}(e_i)$. Since $\varphi(e_i) \xrightarrow{\mathsf{broadcast}} e_i$ and $\xrightarrow{\mathsf{broadcast}} \subseteq \mathsf{vis}$, we have $\varphi(e_i) \in \mathsf{vis}^{-1}(e_i)$. By definition, $e_i \in \mathsf{vis}^{-1}(e_i)$. Thus, $\mathsf{vis}^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\} \subseteq \mathsf{vis}^{-1}(e_i)$.

$Op(\rho[i]) = \mathbf{mreceive}$**:** If $e_k \in \mathsf{vis}^{-1}(e_j)$ then since $e_j \xrightarrow{Rep(e_i)} e_i$ and $\mathsf{vis}; \xrightarrow{\mathcal{R}} \subseteq \mathsf{vis}$, we have $e_k \in \mathsf{vis}^{-1}(e_i)$ Thus $\mathsf{vis}^{-1}(e_j) \subseteq \mathsf{vis}^{-1}(e_i)$. Since $\varphi(e_i) \xrightarrow{\mathsf{merge}} e_i$ and $\mathsf{vis}; \xrightarrow{\mathsf{merge}} \subseteq \mathsf{vis}$, for any $e_k \in \mathsf{vis}^{-1}(\varphi(e_i))$, we have $e_k \in \mathsf{vis}^{-1}(e_i)$. Thus $\mathsf{vis}^{-1}(\varphi(e_i)) \subseteq \mathsf{vis}^{-1}(e_i)$. By defintion, $e_i \in \mathsf{vis}^{-1}(e_i)$. Thus, $\mathsf{vis}^{-1}(e_j) \cup \mathsf{vis}^{-1}(\varphi(e_i)) \cup \{e_i\} \subseteq \mathsf{vis}^{-1}(e_i)$.

Thus, having proved one direction of the equality, to prove the result, it sufficies to prove that

$$\mathsf{vis}^{-1}(e_i) \subseteq \begin{cases} \mathsf{vis}^{-1}(e_j) \cup \{e_i\} & \text{if } Op(e_i) \in \mathsf{Queries} \cup \mathsf{Updates} \cup \{\mathbf{usend}, \mathbf{msend}\} \\ \mathsf{vis}^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\} & \text{if } Op(e_i) = \mathbf{ureceive} \\ \mathsf{vis}^{-1}(e_j) \cup \mathsf{vis}^{-1}(\varphi(e_i)) \cup \{e_i\} & \text{if } Op(e_i) = \mathbf{mreceive} \end{cases}$$

Let $\mathsf{vis}_n^{-1}(e_i) = \{e_k \mid e_k \xrightarrow{\mathsf{vis}_n} e_j\}$. We shall show by induction that for each $n \geq 0$,

$$\mathsf{vis}_n^{-1}(e_i) \subseteq \begin{cases} \mathsf{vis}_n^{-1}(e_j) \cup \{e_i\} & \text{if } Op(e_i) \in \mathsf{Queries} \cup \mathsf{Updates} \cup \{\mathbf{usend}, \mathbf{msend}\} \\ \mathsf{vis}_n^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\} & \text{if } Op(e_i) = \mathbf{ureceive} \\ \mathsf{vis}_n^{-1}(e_j) \cup \mathsf{vis}_n^{-1}(\varphi(e_i)) \cup \{e_i\} & \text{if } Op(e_i) = \mathbf{mreceive} \end{cases}$$

For $n = 0$, recall that $\mathsf{vis}_0 = \xrightarrow{\mathcal{R}} \cup \xrightarrow{\mathsf{broadcast}} \cup \xrightarrow{\mathsf{merge}}$. We consider the following cases

$Op(\rho[i]) \in \mathsf{Queries} \cup \mathsf{Updates} \cup \{\mathbf{usend}, \mathbf{msend}\}$**:** Note that in this case there is no event e_k such that $e_k \xrightarrow{\mathsf{broadcast}} e_i$ or $e_k \xrightarrow{\mathsf{merge}} e_i$.

Thus, $e_k \in \mathsf{vis}_0^{-1}(e_i) \implies e_k \xrightarrow{Rep(e_i)} e_i$ or $e_k = e_i$.

If $e_k \xrightarrow{Rep(e_i)} e_i$, then either $e_k = e_j$ or $e_k \xrightarrow{Rep(e_i)} e_j$. In both these cases $e_k \in \mathsf{vis}_0^{-1}(e_j)$.

Thus, $e_k \in \mathsf{vis}_0^{-1}(e_i) \implies e_k \in \mathsf{vis}_0^{-1}(e_j) \cup \{e_i\}$. Thus $\mathsf{vis}_0^{-1}(e_i) \subseteq \mathsf{vis}_0^{-1}(e_j) \cup \{e_i\}$

$Op(\rho[i]) = \mathbf{ureceive}$**:** In this case there is no event e_k such that $e_k \xrightarrow{\mathsf{merge}} e_i$. Thus, if $e_k \in \mathsf{vis}_0^{-1}(e_i)$ then either $e_k = e_i$ or $e_k \xrightarrow{Rep(e_i)} e_i$ or $e_k \xrightarrow{\mathsf{broadcast}} e_i$.

If $e_k \xrightarrow{Rep(e_i)} e_i$ then using the reasoning from the previous case we have $e_k \in \mathsf{vis}_0^{-1}(e_j)$.
If $e_k \xrightarrow{\mathsf{broadcast}} e_i$ then $e_k = \varphi(e_i)$.

Thus, $e_k \in \mathsf{vis}_0^{-1}(e_i) \implies e_k \in \mathsf{vis}_0^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\}$.

Thus $\mathsf{vis}_0^{-1}(e_i) \subseteq \mathsf{vis}_0^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\}$

$Op(\rho[i]) = \mathbf{mreceive}$**:** In this case there is no event e_k such that $e_k \xrightarrow{\mathsf{broadcast}} e_i$. Thus, if $e_k \in \mathsf{vis}_0^{-1}(e_i)$ then either $e_k \xrightarrow{Rep(e_i)} e_i$ or $e_k \xrightarrow{\mathsf{merge}} e_i$ or $e_k = e_i$.

If $e_k \xrightarrow{Rep(e_i)} e_i$ then using the reasoning from the first case we have $e_k \in \mathsf{vis}_0^{-1}(e_j)$. If $e_k \xrightarrow{\mathsf{merge}} e_i$ then $e_k = \varphi(e_i)$ and thus $e_k \in \mathsf{vis}_0^{-1}(\varphi(e_i))$.

Thus, $e_k \in \mathsf{vis}_0^{-1}(e_i) \implies e_k \in \mathsf{vis}_0^{-1}(e_j) \cup \mathsf{vis}_0^{-1}(\varphi(e_i)) \cup \{e_i\}$.

Hence in this case we have $\mathsf{vis}_0^{-1}(e_i) \subseteq \mathsf{vis}_0^{-1}(e_j) \cup \mathsf{vis}_0^{-1}(\varphi(e_i)) \cup \{e_i\}$

Thus, the result holds for $n = 0$. Suppose the result holds for all integers smaller than n. We shall now show that the result holds for n as well. Recall that

$$\mathsf{vis}_n = \mathsf{vis}_{n-1} \cup \mathsf{vis}_{n-1}; \xrightarrow{\mathcal{R}} \cup \mathsf{vis}_{n-1}; \xrightarrow{\mathsf{merge}}$$

$Op(\rho[i]) \in \mathsf{Queries} \cup \mathsf{Updates} \cup \{\mathbf{usend}, \mathbf{msend}\}$**:** Note that in this case for no e_k is it the case that $e_k \xrightarrow{\mathsf{broadcast}} e_i$ or $e_k \xrightarrow{\mathsf{merge}} e_i$.

Thus if $e_k \in \mathsf{vis}_n^{-1}(e_i)$ then either $e_k \in \mathsf{vis}_{n-1}^{-1}(e_i)$ or there exists $e_{k'}$ such that $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k'} \xrightarrow{Rep(e_i)} e_i$.

Now if $e_k \in \mathsf{vis}_{n-1}^{-1}(e_i)$ by induction hypothesis, $e_k \in \mathsf{vis}_{n-1}^{-1}(e_j) \cup \{e_i\}$. Since $\mathsf{vis}_{n-1}^{-1}(e_j) \subseteq \mathsf{vis}_n^{-1}(e_j)$, this implies that in this case $e_k \in \mathsf{vis}_n^{-1}(e_j) \cup \{e_i\}$.

Otherwise if there exists $e_{k'}$ such that $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k'} \xrightarrow{Rep(e_i)} e_i$, then either $e_{k'} = e_j$ or $e_{k'} \xrightarrow{Rep(e_i)} e_j$. If $e_{k'} = e_j$, then $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k'} \implies e_k \xrightarrow{\mathsf{vis}_{n-1}} e_j \implies e_k \in \mathsf{vis}_{n-1}^{-1}(e_j) \implies e_k \in \mathsf{vis}_n^{-1}(e_j)$.

If $e_{k'} \xrightarrow{Rep(e_i)} e_j$, then $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k'} \xrightarrow{Rep(e_i)} e_j \implies e_k \in \mathsf{vis}_n^{-1}(e_j)$.

Thus, in both these sub-cases, we have $e_k \in \mathsf{vis}_n^{-1}(e_j)$.

From this we can conclude that for $Op(e_i) \in \mathsf{Queries} \cup \mathsf{Updates} \cup \{\mathbf{usend}, \mathbf{msend}\}$ $e_k \in \mathsf{vis}_n^{-1}(e_i) \implies e_k \in \mathsf{vis}_n^{-1}(e_j) \cup \{e_i\}$. Thus $\mathsf{vis}_n^{-1}(e_i) \subseteq \mathsf{vis}_n^{-1}(e_i) \cup \{e_i\}$.

$Op(\rho[i]) = \mathbf{ureceive}$**:** In this case for no e_k we have $e_k \xrightarrow{\mathsf{merge}} e_i$. Thus, if $e_k \in \mathsf{vis}_n^{-1}(e_i)$ then either $e_k \in \mathsf{vis}_{n-1}^{-1}(e_i)$ or there exists $e_{k'}$ such that $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k'} \xrightarrow{Rep(e_i)} e_i$.

If $e_k \in \mathsf{vis}_{n-1}^{-1}(e_i)$, by induction hypothesis, $e_k \in \mathsf{vis}_{n-1}^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\}$ since $\mathsf{vis}_{n-1}^{-1}(e_j) \subseteq \mathsf{vis}_n^{-1}(e_j)$, this implies that $e_k \in \mathsf{vis}_n^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\}$

Otherwise if there exists $e_{k'}$ such that $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k''} \xrightarrow{Rep(e_i)} e_i$, then using the reasoning similar to the previous case we can conclude that $e_k \in \mathsf{vis}_n^{-1}(e_j)$.

Thus, $e_k \in \mathsf{vis}_n^{-1}(e_i) \implies e_k \in \mathsf{vis}_n^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\}$.

Thus in this case $\mathsf{vis}_n^{-1}(e_i) \subseteq \mathsf{vis}_n^{-1}(e_i) \cup \{\varphi(e_i)\} \cup \{e_i\}$.

$Op(\rho[i]) = \mathbf{mreceive}$**:** In this case, if $e_k \in \mathsf{vis}_n^{-1}(e_i)$ then either $e_k \in \mathsf{vis}_{n-1}^{-1}(e_i)$ or there exists $e_{k'}$ such that $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k'} \xrightarrow{Rep(e_i)} e_i$ or there exists a $e_{k''}$ such that $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k''} \xrightarrow{\mathsf{merge}} e_i$.

If $e_k \in \mathsf{vis}_{n-1}^{-1}(e_i)$, by induction hypothesis, $e_k \in \mathsf{vis}_{n-1}^{-1}(e_j) \cup \mathsf{vis}_{n-1}^{-1}(\varphi(e_i)) \cup \{e_i\}$. Since $\mathsf{vis}_{n-1}^{-1}(e_j) \subseteq \mathsf{vis}_n^{-1}(e_j)$ and $\mathsf{vis}_{n-1}^{-1}(\varphi(e_i)) \subseteq \mathsf{vis}_n^{-1}(\varphi(e_i))$, this implies that $e_k \in \mathsf{vis}_n^{-1}(e_j) \cup \mathsf{vis}_n^{-1}(\varphi(e_i)) \cup \{e_i\}$.

Otherwise if there exists $e_{k'}$ such that $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k'} \xrightarrow{Rep(e_i)} e_i$, then using the reasoning similar to the first case we can conclude that $e_k \in \mathsf{vis}_n^{-1}(e_j)$.

Finally if there exists $e_{k''}$ such that $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k''} \xrightarrow{\mathsf{merge}} e_i$, then, $e_{k''} = \varphi(e_i)$. Thus, $e_k \xrightarrow{\mathsf{vis}_{n-1}} e_{k''} \implies e_k \xrightarrow{\mathsf{vis}_{n-1}} \varphi(e_i) \implies e_k \xrightarrow{\mathsf{vis}_n} \varphi(e_i) \implies e_k \in \mathsf{vis}_n^{-1}(\varphi(e_i))$

Thus, $e_k \in \mathsf{vis}_n^{-1}(e_i) \implies e_k \in \mathsf{vis}_n^{-1}(e_j) \cup \mathsf{vis}_n^{-1}(\varphi(e_i)) \cup \{e_i\}$.

Thus in this case $\mathsf{vis}_n^{-1}(e_i) \subseteq \mathsf{vis}_n^{-1}(e_i) \cup \mathsf{vis}_n^{-1}(\varphi(e_i)) \cup \{e_i\}$.

Thus, by the principle of mathematical induction we have shown that for each $n \geq 0$,

$$\mathsf{vis}_n^{-1}(e_i) \subseteq \begin{cases} \mathsf{vis}_n^{-1}(e_j) \cup \{e_i\} & \text{if } Op(e_i) \in \mathsf{Queries} \cup \mathsf{Updates} \cup \{\mathbf{usend}, \mathbf{msend}\} \\ \mathsf{vis}_n^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\} & \text{if } Op(e_i) = \mathbf{ureceive} \\ \mathsf{vis}_n^{-1}(e_j) \cup \mathsf{vis}_n^{-1}(\varphi(e_i)) \cup \{e_i\} & \text{if } Op(e_i) = \mathbf{mreceive} \end{cases}$$

With this we have shown that

$$\mathsf{vis}^{-1}(e_i) = \begin{cases} \mathsf{vis}^{-1}(e_j) \cup \{e_i\} & \text{if } Op(e_i) \in \mathsf{Queries} \cup \mathsf{Updates} \cup \{\mathbf{usend}, \mathbf{msend}\} \\ \mathsf{vis}^{-1}(e_j) \cup \{\varphi(e_i)\} \cup \{e_i\} & \text{if } Op(e_i) = \mathbf{ureceive} \\ \mathsf{vis}^{-1}(e_j) \cup \mathsf{vis}^{-1}(\varphi(e_i)) \cup \{e_i\} & \text{if } Op(e_i) = \mathbf{mreceive} \end{cases}$$

□

The abstract specification of the replicated data type should explain the observable behaviour of that replicated data types. The observable behaviour would be the response of a query operation submitted to some replica in a run. The abstract specification should provide this response as a function of the update operations that were visible to the replica at that point in the run. This is because only the update methods result in change of the state of the data type. The *update-receive*, *merge-send* and *merge-receive* are operations which are defined by the implementation, but they are not methods defined by the abstract replicated data type. In the trace of a run, the *visible event set* of an event may consist of all kinds of events, including the ones corresponding to query operations, and communication operations apart from the update operations. Thus, for the purpose of defining the abstract specification, we are interested only in the subset of *update events* from the *visible event set* of an event. This subset is known as an *ideal* of that event. Before we formally define an *ideal* we set up the following notation.

Suppose $\alpha = (\rho, \varphi)$ is a run and $(\mathcal{E}_\rho, \mathsf{vis})$ is its trace. Let $E \subseteq \mathcal{E}_\rho$ be some subset of events. Then,

- $E|_{\mathsf{Updates}} = \{e_i \in E \mid Op(e_i) \in \mathsf{Updates}\}$
- $E|_{\mathsf{Queries}} = \{e_i \in E \mid Op(e_i) \in \mathsf{Queries}\}$
- $E|_{(\mathsf{Queries} \cup \mathsf{Updates})} = E|_{\mathsf{Queries}} \cup E|_{\mathsf{Updates}}$

Definition 84 (Ideal of an event)**.** *Let $T = (\mathcal{E}_\rho, \mathsf{vis})$ be a trace of some run. Let $e \in \mathcal{E}_\rho$. Then the* ideal *of e in the trace T, denoted by $Ideal_T(e)$ is the set of update events visible to e.*

$$Ideal_T(e) = \mathsf{vis}^{-1}(e)|_{\mathsf{Updates}}$$

For example, the Figure 4.6 shows the Ideal of the event M contains the update operations $\{Init, A, B, D, E\}$.

Here, we recall that the causal past of an operation is the set of update operations in the run which were delivered at the source replica of that operation at that point in the run. The causal past plays an important role in determining if the implementation of a replicated data type satisfies strong eventual consistency. We now show that in the trace of a run, the causal past of an operation in a run corresponds to the *ideal* of the corresponding event in the trace.

Proposition 85. *Let $\alpha = (\rho, \varphi)$ be a run. Let $\mathcal{E}_\rho$ be the set of events corresponding to the operations in ρ. Then for $j \leq i \leq |\rho|$,*

$$\rho[j] \in \mathsf{Past}_\alpha(\rho[i]) \iff e_j \in Ideal_T(e_i)$$

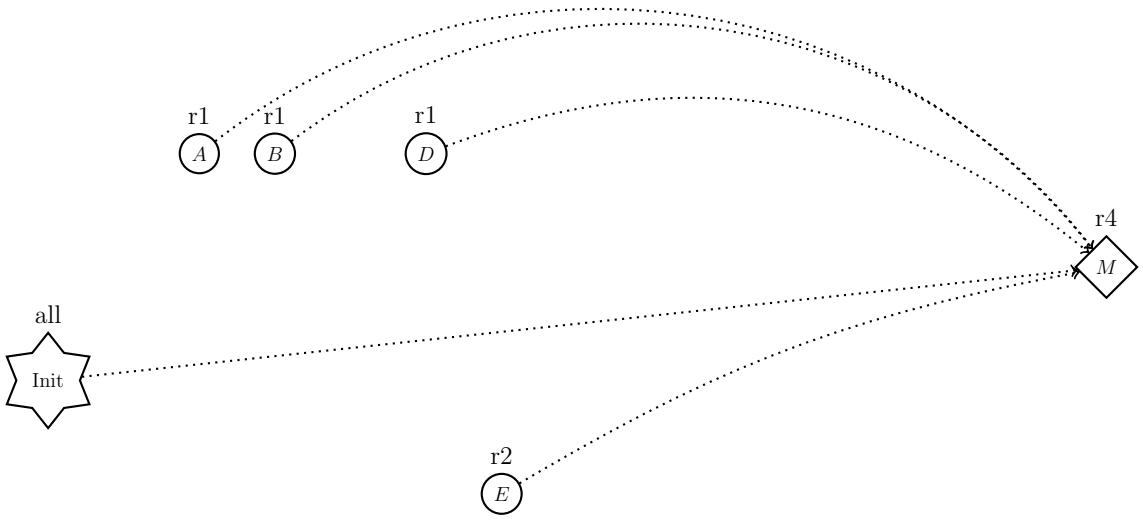


Figure 4.6: Ideal of the event M

Proof. We shall prove this by induction over the length of ρ. For a $n = 0$ with just the initialization operation, the result trivially holds.

Let us assume that the result is true for all runs of length at most $n - 1$.

Consider a run $\alpha = (\rho, \varphi)$ with $|\rho| = n$. Let $T = (\mathcal{E}, \mathsf{vis})$ be its trace.

Suppose there exists $j \leq i < n$, such that $\rho[j] \in \mathsf{Past}_\alpha(\rho[i])$. If we consider the prefix run $\alpha' = (\rho', \varphi')$ with $\rho' = \rho[1, \ldots, i]$ and $\varphi' = \varphi|_{\rho'}$. Then, we have $\rho[j] = \rho'[j] \in \mathsf{Past}_{\alpha'}(\rho'[i])$.

Let $T' = (\mathcal{E}', \mathsf{vis}')$ be the trace of α'.

For $j \leq i < n$, we have,

$$\begin{aligned}
\rho[j] \in \mathsf{Past}_\alpha(\rho[i]) &\iff \rho'[j] \in \mathsf{Past}_{\alpha'}(\rho'[i]) \text{ [Since } \rho' \text{ is a prefix of } \rho \text{ and } |\rho'| = i] \\
&\iff e_j \in \mathit{Ideal}_{T'}(e_i) \text{ [By Induction Hypothesis]} \\
&\iff e_j \in \mathsf{vis}'^{-1}(e_i)|_{\mathsf{Updates}} \text{ [By Definition of an Ideal]} \\
&\iff e_j \in \mathsf{vis}^{-1}(e_i)|_{\mathsf{Updates}} \text{ [Since } \mathsf{vis}' = \mathsf{vis}|_{\mathcal{E}'} \text{ and } e_i, e_j \in \mathcal{E}'] \\
&\iff e_j \in \mathit{Ideal}_T(e_i)
\end{aligned}$$

Thus for $j \leq i < n$, the result is proved.

We now consider the case when $i = n$. It suffices to prove that for all $j \leq n$,

$$\rho[j] \in \mathsf{Past}_\alpha(\rho[n]) \iff e_j \in \mathit{Ideal}_T(e_n).$$

Let $k < n$ be the largest integer such that $\mathit{Rep}(\rho[k]) = \mathit{Rep}(\rho[n])$.

We consider the following cases.

$Op(e_n) \in \mathsf{Queries} \cup \{\mathbf{msend}, \mathbf{usend}\}$**:** In this case $\mathsf{Past}_\alpha(\rho[n]) = \mathsf{Past}_\alpha(\rho[k])$.

$$\begin{aligned}
\rho[j] \in \mathsf{Past}_\alpha(\rho[n]) &\iff \rho[j] \in \mathsf{Past}_\alpha(\rho[k]) \text{ [By definition]} \\
&\iff e_j \in \mathit{Ideal}_T(e_k) \text{ [By induction hypothesis]} \\
&\iff e_j \in \mathsf{vis}^{-1}(e_k)|_{\mathsf{Updates}} \text{ [By definition of ideal]} \\
&\iff e_j \in (\mathsf{vis}^{-1}(e_k) \cup \{e_n\})|_{\mathsf{Updates}} \\
&\qquad\qquad \text{[Since } e_n \text{ is not an update event]} \\
&\iff e_j \in \mathsf{vis}^{-1}(e_n)|_{\mathsf{Updates}} \text{ [proposition 83]} \\
&\iff e_j \in \mathit{Ideal}_T(e_i) \text{ [By definition]}
\end{aligned}$$

$Op(e_n) \in \mathsf{Updates}$**:** In this case,

$$\mathsf{Past}_\alpha(\rho[n]) = \mathsf{Past}_\alpha(\rho[k]) \cup \{\rho[n]\}$$

. And $\rho[j] = \rho[n]$ if and only if $e_j = e_n$.

$$\begin{aligned}
\rho[j] \in \mathsf{Past}_\alpha(\rho[n]) &\iff \rho[j] \in \mathsf{Past}_\alpha(\rho[k]) \cup \{\rho[n]\} \text{ [By definition]} \\
&\iff e_j \in (\mathit{Ideal}_T(e_k) \cup \{e_n\}) \text{ [By induction hypothesis]} \\
&\iff e_j \in (\mathsf{vis}^{-1}(e_k)|_{\mathsf{Updates}} \cup \{e_n\}) \text{ [By definition of an Ideal]} \\
&\iff e_j \in (\mathsf{vis}^{-1}(e_k) \cup \{e_n\})|_{\mathsf{Updates}} \text{ [Since } e_n \text{ is an update event]} \\
&\iff e_j \in \mathsf{vis}^{-1}(e_n)|_{\mathsf{Updates}} \text{ [proposition 83]} \\
&\iff e_j \in \mathit{Ideal}_T(e_i) \text{ [By definition]}
\end{aligned}$$

$Op(e_n) = \mathbf{ureceive}$: Let $\varphi(\rho[n]) = \rho[k']$. In this case,

$$\mathsf{Past}_\alpha(\rho[n]) = \mathsf{Past}_\alpha(\rho[k]) \cup \{\rho[k']\}$$

. And $\rho[j] = \rho[k']$ if and only if $e_j = e_{k'}$.

$$\begin{aligned}
\rho[j] \in \mathsf{Past}_\alpha(\rho[n]) &\iff \rho[j] \in \mathsf{Past}_\alpha(\rho[k]) \cup \{\varphi(\rho[n])\} \text{ [By definition]} \\
&\iff \rho[j] \in \mathsf{Past}_\alpha(\rho[k]) \cup \{\rho[k']\} \text{ [Since } \varphi(\rho[n]) = \rho[k']] \\
&\iff e_j \in (\mathit{Ideal}_T(e_k) \cup \{e_{k'}\}) \text{ [By induction hypothesis]} \\
&\iff e_j \in (\mathsf{vis}^{-1}(e_k)|_{\mathsf{Updates}} \cup \{e_{k'}\}) \text{ [By definition of the Ideal]} \\
&\iff e_j \in (\mathsf{vis}^{-1}(e_k) \cup \{e_{k'}\})|_{\mathsf{Updates}} \\
&\qquad\qquad \text{[Since } e_{k'} \text{ is an update event]} \\
&\iff e_j \in (\mathsf{vis}^{-1}(e_k) \cup \{e_{k'}\} \cup \{e_n\})|_{\mathsf{Updates}} \\
&\qquad\qquad \text{[Since } e_n \text{ is not an update event]} \\
&\iff e_j \in \mathsf{vis}^{-1}(e_n)|_{\mathsf{Updates}} \text{ [Proposition 83]} \\
&\iff e_j \in \mathit{Ideal}_T(e_i) \text{ [By definition]}
\end{aligned}$$

$Op(e_n) = \mathbf{mreceive}$: Let $\varphi(\rho[n]) = \rho[k']$.

$\mathsf{Past}_\alpha(\rho[n]) = \mathsf{Past}_\alpha(\rho[k]) \cup \mathsf{Past}_\alpha(\rho[k'])$.

$$\begin{aligned}
\rho[j] \in \mathsf{Past}_\alpha(\rho[n]) &\iff \rho[j] \in \mathsf{Past}_\alpha(\rho[k]) \cup \mathsf{Past}_\alpha(\varphi(\rho[n])) \text{ [By definition]} \\
&\iff \rho[j] \in \mathsf{Past}_\alpha(\rho[k]) \cup \mathsf{Past}_\alpha(\rho[k']) \text{ [By definition]} \\
&\iff e_j \in \mathit{Ideal}_T(e_k) \cup \mathit{Ideal}_T(e_{k'}) \text{ [By induction hypothesis]} \\
&\iff e_j \in \mathsf{vis}^{-1}(e_k)|_{\mathsf{Updates}} \cup \mathsf{vis}^{-1}(e_{k'})|_{\mathsf{Updates}} \text{ [By definition]} \\
&\iff e_j \in (\mathsf{vis}^{-1}(e_k) \cup \mathsf{vis}^{-1}(e_{k'}))|_{\mathsf{Updates}} \\
&\qquad\qquad \text{[By property of restriction } |_{\mathsf{Updates}}] \\
&\iff e_j \in (\mathsf{vis}^{-1}(e_k) \cup \mathsf{vis}^{-1}(e_{k'}) \cup \{e_n\})|_{\mathsf{Updates}} \\
&\qquad\qquad \text{[Since } e_n \text{ is not an update event]} \\
&\iff e_j \in \mathsf{vis}^{-1}(e_n)|_{\mathsf{Updates}} \text{ [Proposition 83]} \\
&\iff e_j \in \mathit{Ideal}_T(e_i) \text{ [By definition]}
\end{aligned}$$

Thus, in each of these cases we have shown that for all $j \leq n$, $\rho[j] \in \mathsf{Past}_\alpha(\rho[n]) \iff e_j \in \mathit{Ideal}_T(e_n)$. Thus, the result is true when $|\rho| = n$.

By principle of mathematical induction, the for all runs $\alpha = (\rho, \varphi)$, for all $j \leq i \leq |\rho|$,

$$\rho[j] \in \mathsf{Past}_\alpha(\rho[i]) \iff e_j \in \mathit{Ideal}_T(e_i)$$

□

Thus, the *ideal* of an event captures the causal past of the corresponding operation. In the following corollary we show that for a pair of update events of a run, the visibility relation captures *happened-before* relation between the corresponding update operations.

Corollary 86. *Let $\alpha = (\rho, \varphi)$ be a run. Let $\mathcal{E}_\rho$ be the set of events corresponding to the operations in ρ. Then, for any pair of update operations $\rho[i], \rho[j]$, $\rho[i] \xrightarrow{\mathsf{hb}} \rho[j]$ iff $e_i \xrightarrow{\mathsf{vis}} e_j$.*

Proof. Let $Rep(\rho[j]) = r$.

By definition $\rho[i] \xrightarrow{\mathsf{hb}} \rho[j]$ iff $\rho[i] \in \mathsf{Past}_\alpha(\rho[j])$. From Proposition 85 this is true iff $e_i \xrightarrow{\mathsf{vis}} e_j$. □

Note that in a run $\alpha = (\rho, \varphi)$ where the updates are causally delivered, for any three update operations $\rho[i]$, $\rho[j]$ and $\rho[k]$, if $\rho[i]$ *happened-before* $\rho[j]$ and $\rho[j]$ *happened-before* $\rho[k]$, then, by causal delivery, $\rho[i]$ would have *happened-before* $\rho[k]$. Thus the causal past of any operation would be downward closed under the *happened-before* relation. Since in the trace representation of a run, the *ideal* of an event corresponds to the causal past of the corresponding operation of an event, we can define the notion of a *downward closed ideal* with respect to the visibility relation.

Definition 87 (Downward closed Ideal)**.** *Let $T = (\mathcal{E}, \mathsf{vis})$ be a trace of some run. Let $e \in \mathcal{E}$ be some event. Then, $Ideal_T(e)$ is said to be downward closed with respect to* vis *iff for any pair of update events e'', e', such that $e'' \xrightarrow{\mathsf{vis}} e'$ and $e' \in Ideal_T(e)$, we have $e'' \in Ideal_T(e)$.*

We now show that in the trace of any causally-delivered run, all the ideals are downward closed with respect to the visibility relation.

Proposition 88. *Let $T = (\mathcal{E}, \mathsf{vis})$ be a trace of some causally-delivered run. Then, for every event $e \in \mathcal{E}$, $Ideal_T(e)$ is downward closed with respect to* vis.

Proof. Let $\alpha = (\rho, \varphi)$ be the run of T. Then, $\mathcal{E} = \mathcal{E}_\rho$.

Let e_k, e_j be update events such that $e_k \xrightarrow{\mathsf{vis}} e_j$. Let $e_j \in Ideal_T(e_i)$ for some $e_i \in \mathcal{E}$. We need to show that $e_k \in Ideal_T(e_i)$. If $k = j$, there is nothing to prove. Consider the case where $k \neq j$.

Let $Rep(e_i) = r$. Let $Rep(e_j) = r'$.

Since $e_j \in Ideal_T(e_i)$ and by definition $Ideal_T(e_i) = \mathsf{vis}^{-1}(e_i)|_{\mathsf{Updates}}$, by proposition 85, $\rho[j] \in \mathsf{Past}_\alpha(\rho[i])$.

Since $e_k \xrightarrow{\mathsf{vis}} e_j$, from proposition 85, $\rho[k] \in \mathsf{Past}_\alpha(\rho[j])$.

Since α is a causally delivered run, $\rho[j] \in \mathsf{Past}_\alpha(\rho[i]) \implies \mathsf{Past}_\alpha(\rho[j]) \subseteq \mathsf{Past}_\alpha(\rho[i])$.

Thus, $\rho[k] \in \mathsf{Past}_\alpha(\rho[i])$. Again from proposition 85, $e_k \in \mathsf{vis}^{-1}(e_i)|_{\mathsf{Updates}} = Ideal_T(e_i)$.

Since e_i, e_j, e_k were arbitrary events chosen, for all the events in T, the ideal is downward closed with respect to vis. □

As a corollary to this proposition we can show the following interesting property for traces of causally-delivered runs where the ideal of an update receive event can be computed from the ideal of its predecessors. We will use this later when we define reference implementations for replicated-data types.

Corollary 89. *If $T = (\mathcal{E}, \mathsf{vis})$ be a trace of a causally delivered run.*

Let e be an **ureceive** *event with $Rep(e) = r$. Let e' be the immediate predecessor of e in $\xrightarrow{r}$. Then,*

$$Ideal_T(e) = Ideal_T(e') \cup Ideal_T(\varphi(e))$$

Proof. From proposition 83, Now $\mathsf{vis}^{-1}(e) = \mathsf{vis}^{-1}(e') \cup \{\varphi(e)\} \cup \{e\}$. Thus, $Ideal_T(e) = \mathsf{vis}^{-1}(e)|_{\mathsf{Updates}} = \mathsf{vis}^{-1}(e')|_{\mathsf{Updates}} \cup \{\varphi(e)\} = Ideal_T(e') \cup \{\varphi(e)\}$. Since $\varphi(e) \in Ideal_T(\varphi(e))$, $Ideal_T(e) \subseteq Ideal_T(e') \cup Ideal_T(\varphi(e))$.

Now, since, $Ideal_T(e) = Ideal_T(e') \cup \{\varphi(e)\}$, we have $Ideal_T(e') \subseteq Ideal_T(e)$.

Suppose $e'' \in Ideal_T(\varphi(e))$. Then, e'' is an update event such that $e'' \xrightarrow{\mathsf{vis}} \varphi(e)$. Since $\varphi(e) \in Ideal_T(e)$, and T is a trace of a causally delivered run, $Ideal_T(e)$ is downward closed with respect to vis. Thus $e'' \in Ideal_T(e)$.

Since e'' is an arbitrary member of $Ideal_T(\varphi(e))$ it implies that $Ideal_T(\varphi(e)) \subseteq Ideal_T(e)$.

Thus, $Ideal_T(e') \cup Ideal_T(\varphi(e)) \subseteq Ideal_T(e)$.

From this, we can conclude that $Ideal_T(e) = Ideal_T(e') \cup Ideal_T(\varphi(e))$. □

Many a times, we are only interested in the relationship between a subset of events in the trace of a run. For instance, we would like to know how the update events in the ideal of an event are related to each other with respect to the visibility relation. This can be obtained by restricting the trace over these interesting subset of events. In general, a trace restricted to a subset of its events is known as *subtrace* induced by that subtrace of events.

Definition 90 (Subtrace of a Trace)**.** *Let $T = (\mathcal{E}, \mathsf{vis})$ be a trace of some run. Let $E \subseteq \mathcal{E}$ be some subset of events. Let $\mathsf{vis}|_E$ denote the visibility relation restricted to the events in E. Then, the subtrace of the trace T induced by the subset E, denoted by T_E is $(E, \mathsf{vis}|_E)$*

The subtrace induced by the *ideal* of an event is of particular interest to us for defining the abstract specification of the replicated data type. We call this subtrace the *view* of that event. For example, the Figure 4.7 shows the view of the event M induced by its ideal $\{Init, A, B, D, E\}$.

Definition 91 (View of an Event)**.** *Let $T = (\mathcal{E}_\rho, \mathsf{vis})$ be a trace of some run. Let $e \in \mathcal{E}_\rho$. The* view *of e in T is the subtrace generated by the ideal $Ideal_T(e)$. We denote the view of e in T by $\partial_T(e)$.*

$$\partial_T(e) = (Ideal_T(e), \mathsf{vis}|_{Ideal_T(e)})$$

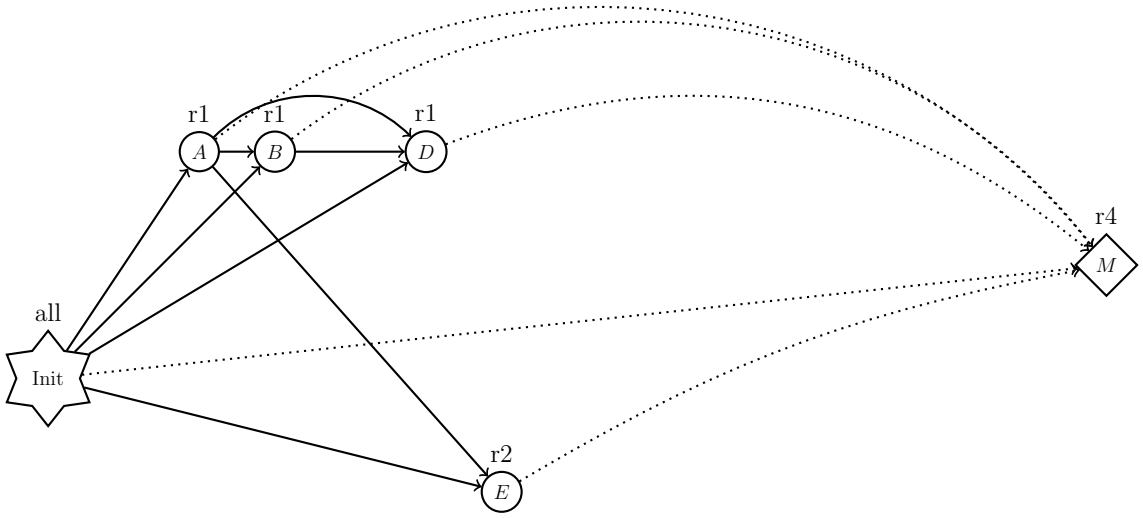


Figure 4.7: View of the event M

The set of all the views of a RDT $\mathcal{D}$ is denoted by $\mathcal{V}(\mathcal{D})$ and is defined to be

$$\mathcal{V}(\mathcal{D}) = \bigcup_{T \in \mathcal{T}(\mathcal{D})} \bigcup_{e \in Events(T)} \partial_T(e)$$

For some replicated data types, such as the OR-Sets, where the updates are delivered in a causal order, or MV-Registers the observable behaviour of the data type is defined by the *latest* update events in the view which do not have a successor in that view with respect to the visibility relation. In the view of the event M in Figure 4.7, the events D and E do not have any successors as per the visibility relation inside the view. We denote such events to be the *maximal events* in the view. We shall define the maximal events in a sub-trace first. Since a *view* is also a subtrace, the definition can be extended to a *view* as well.

Definition 92 (Maximal events in a sub-trace)**.** *Let $T = (\mathcal{E}_\rho, \mathsf{vis})$ be a trace of some run. Let T_E be a subtrace of T induced by the events $E \subseteq \mathcal{E}_\rho$. Then set of maximal events in T_E, denoted by $max(T_E)$ is defined to be the subset of events in E that do not have a successor with respect to the visibility relation* vis *in E. Thus*

$$max(T_E) = \{e \in E : \quad | \quad \forall e' \in E : e' \xrightarrow{\mathsf{vis}} e \vee e' || e\}$$

Let e be an event in T, then the set of maximal update events in the view of e in T is denoted by $max\partial_T(e)$.

A trace presents us with a picture of which operations were visible to other operations. However , it does not shed any light on the manner in which the replicas arbitrate among concurrent conflicting update events. This arbitration information might be relevant to certain kind of replicated data types, such as the Key-Value stores, where multiple Write operations to the same key might be visible to a Read operation, but the order in which they were arbitrated determines what is returned by the replica as a response to the Read() query. In practice, the replicas may choose any strategy to arbitrate between the concurrent update operations. Popular strategies include *last-writer-wins* (where every update is tagged with a timestamp and the replicas arbitrate among on the maximal update events to pick the event with the greatest timestamp) or picking the arbitrating the update based on the greatest replica-id. We now formally define *arbitration strategy* which provides an total order known as *arbitration relation* over the events of a trace of a given run.

Definition 93 (Arbitration Strategy and Arbitration relation)**.** *Let $T = (\mathcal{E}, \mathsf{vis})$ be a trace of a run $\alpha = (\rho, \varphi)$. Then, we define an arbitration strategy to be a function $F_{\mathsf{arb}}(\alpha, T)$ which generates a total order* arb *over the update events in $\mathcal{E}$ such that $\mathsf{vis}|_{\mathsf{Updates}} \subseteq \mathsf{arb}$. This total order* arb *is said to be an arbitration relation over α.*

We say that an arbitration strategy $F_{\mathsf{arb}}()$ is consistent *if for any prefix α' of α whose trace is $T' = (\mathcal{E}', \mathsf{vis}')$, if $\mathsf{arb}' = F_{\mathsf{arb}}(\alpha', T')$, then, it is the case that $\mathsf{arb}' = \mathsf{arb}|_{\mathcal{E}'}$.*

Given a subset of update events $E \subseteq \mathcal{E}|_{\mathsf{Updates}}$ and an arbitration relation arb *over the update events in $\mathcal{E}$, we can determine the unique* maximum *event in E with respect to the arbitration relation. We denote this* maximum *event as $max^{\mathsf{arb}}(E)$ which can be defined as*

$$max^{\mathsf{arb}}(E) = e \in E \iff \forall e' \in E : e' \xrightarrow{\mathsf{arb}} e$$

A run has a unique trace associated with it, since the visibility relation is obtained from the replica-order, the broadcast-order and the merge-order which are derived from the run. However, an arbitration relation is dependent on the arbitration-strategy which is implementation dependent. Hence, in order to reason about the behaviours of the implementations of such replicated data types that rely upon an arbitration strategy, we need to include the arbitration relation in our model along with the trace of a run. Hence we define an *execution* of a run which contains both the trace of a run and the arbitration relation over the events of the trace.

Definition 94 (Execution of a Run)**.** *Let $\alpha = (\rho, \varphi)$ be a valid run of a replicated data-type. Let arbitration strategy function $F_{\mathsf{arb}}()$. Let $T = (\mathcal{E}, \mathsf{vis})$ be the trace of α. Let $\mathsf{arb} = F_{\mathsf{arb}}(\alpha, T)$ be the arbitration relation over T defined by F_{arb}*

Then an execution of this run is the tuple $\mathcal{A}(\alpha, \mathsf{arb}) = (\mathcal{E}_\rho, \mathsf{vis}, \mathsf{arb})$, where $(\mathcal{E}_\rho, \mathsf{vis})$ is the trace of α and arb *is an arbitration order over the trace $(\mathcal{E}_\rho, \mathsf{vis})$*

The set of all the executions of all the runs of a replicated data type $\mathcal{D}$, denoted by $\mathcal{A}(\mathcal{D})$ is defined to be the set $\bigcup_{\alpha \in Runs(\mathcal{D})} \bigcup_{\mathsf{arb}\ over\ \mathcal{T}(\alpha)} \mathcal{A}(\alpha, \mathsf{arb})$.

Note that the observable behavior of a replica at any point in the run is based on the view of the latest event of the replica in the trace of the run. In case of implementations that depend on arbitrating among the update events, it would also depend on the arbitration relation over the update events in that view. In order to model this, we define the *context* of an event in an *execution*.

Definition 95 (Context)**.** *Let $A = (\mathcal{E}_\rho, \mathsf{vis}, \mathsf{arb})$ be an execution of $\mathcal{D}$. Let the corresponding trace be $T = (\mathcal{E}_\rho, \mathsf{vis})$. Let $e \in \mathcal{E}_\rho$ be some event.*

Then the Context of event e, written as $Ctxt_A(e)$ is the the sub-execution of A restricted to $Ideal_T(e)$. Thus

$$Ctxt_A(e) = (\mathcal{E}_\rho|_{Ideal_T(e)}, \mathsf{vis}|_{Ideal_T(e)}, \mathsf{arb}|_{Ideal_T(e)})$$

Set of all contexts of $\mathcal{D}$, denoted by $Ctxt(\mathcal{D})$, is defined to be $\bigcup_{A \in \mathcal{A}(\mathcal{D})} \bigcup_{e \in Events(A)} Ctxt_A(e)$.

We say that a pair of context $U = (\mathcal{E}, \mathsf{vis}, \mathsf{arb})$ and $U' = (\mathcal{E}', \mathsf{vis}', \mathsf{arb}')$ where $U, U' \in Ctxt(\mathcal{D})$ are isomorphic *if there exists a bijective map $g : \mathcal{E} \to \mathcal{E}'$ such that for all $e \in \mathcal{E}$*

$$Op(e) = Op(g(e)) \wedge Args(e) = Args(g(e))$$

and for all $e_1, e_2 \in \mathcal{E}$,we have

$$e_1 \xrightarrow{\mathsf{vis}} e_2 \iff g(e_1) \xrightarrow{\mathsf{vis}'} g(e_2) \wedge e_1 \xrightarrow{\mathsf{arb}} e_2 \iff g(e_1) \xrightarrow{\mathsf{arb}'} g(e_2)$$

Thus, a pair of isomorphic contexts are identical, except that the corresponding update events in the two context may have different source replicas.

Note that any run of a replicated data type can be extended by a query operation performed at of the replicas in the system. The abstract specification of the data type should tell us what should the response of this query be if it is performed at the replica at that point in the run. The abstract specification should be able to inform this based on the update operations that are visible to the replica at that point, the visibility relation between the update operations themselves. In case the implementation requires an arbitration strategy to answer the query, the abstract specification should also take that arbitration strategy into consideration. For a given arbitration strategy, the *context* of an event in an execution of a run contains all this relevant information. Thus, we now define the *declarative specification* for a replicated data-type in terms of the return value of the query when that query is applied to a particular context.

Definition 96 (Specification of an RDT [Burckhardt, 2014])**.** *A specification of a replicated datatype $\mathcal{D}$ is a function*

$$Spec_{\mathcal{D}} : Ctxt(\mathcal{D}) \times \mathsf{Queries} \times \mathsf{Univ}^* \to \mathsf{Rets}$$

which given a context and a query and some valid argument to the query provides the value that the query should return when applied at this context such that for any pair of isomorphic contexts U, U' and any query q and any valid arguments to the query arg

$$Spec_{\mathcal{D}}(U, q, \text{arg}) = Spec_{\mathcal{D}}(U', q, \text{arg})$$

Thus a specification defines the observable behaviour of a replicated data-type in an implementation-independent manner. We will define a particular kind of specification called the *arbitration agnostic specification*, which we shall be using in this and the subsequent chapters. These specification do not depend on the arbitration relation of a context to determine what should be the return value of a query when applied to a context.

Definition 97 (Arbitration Agnostic Specification)**.** *A specification $Spec_{\mathcal{D}}$ is said to be arbitration agnostic if for every query $q \in \mathsf{Queries}$ and every valid argument* arg $\in \mathsf{Univ}^*$*, for any pair of contexts $U = (\mathcal{E}, \mathsf{vis}, \mathsf{arb})$ and $U' = (\mathcal{E}', \mathsf{vis}', \mathsf{arb}')$ such that $\mathcal{E} = \mathcal{E}'$ and $\mathsf{vis} = \mathsf{vis}$, we have $Spec_{\mathcal{D}}(U, q, \text{arg}) = Spec_{\mathcal{D}}(U', q, \text{arg})$*

Thus an arbitration-agnostic specification is the function $Spec_{\mathcal{D}} : \mathcal{V}(\mathcal{D}) \times \mathsf{Queries} \times \mathsf{Univ}^* \to \mathsf{Rets}$.

4.2.1 Specifications of popular replicated data types

We shall now provide declarative specifications of some of the popular CRDTs.

PN counters

A PN counter is a replicated counter which allows clients to increment the value of a counter by 1 via the method $\mathsf{Inc}()$ and decrement the value of the counter by one via the method $\mathsf{Dec}()$. It also allows the clients to know the current value of the counter via the method $\mathsf{Fetch}()$. In an *operation-based* implementation of the *PN Counter*, the replicas on receiving an $\mathsf{Inc}()$ (respectively $\mathsf{Dec}()$) request from the clients will update their local state to reflect the state change following the increment (respectively decrement). They will then propagate some auxiliary information which will help the remote replicas modify their respective local states to incorporate this latest increment (respectively decrement). In a *state-based* implementation of the *PN Counter*, when the replica receives an $\mathsf{Inc}()$ (respectively $\mathsf{Dec}()$) request, it updates the local state to incorporate the increment (respectively decrement) into the state. From time to time, the replicas share their entire state with other replicas and the recepient replicas will update their state so that the value of the counter corresponds to the total number of unique increments and decrements requests seen by replica and the sender of the merge. When a replica receives a $\mathsf{Fetch}()$ requests, it returns the value of the counter that it is currently aware of. Below, we provide an arbitration agnostic specification for the *PN-Counter*.

- $\mathsf{Updates} = \{\mathsf{Inc}, \mathsf{Dec}\}$
- $\mathsf{Queries} = \{\mathsf{Fetch}\}$ and $arity(\mathsf{Fetch}) = 0$
- **Specification:** Let V be a view in the trace T of a well-defined run of PN-Counters. Let $I(V) = \{e \in T \mid Op(e) = \mathsf{Inc}\}$ and $D(V) = \{e \in V \mid Op(e) = \mathsf{Dec}\}$.

$$Spec_{Counter}(V, \mathsf{Fetch}, \bot) = |I(V)| - |D(V)|$$

LWW registers

An Last Writer Wins (LWW) register is a replicated read-write register. It provides a method named Write which allows the clients to modify the state of the register and a method named as Read to query the current state of the register. Since there could be concurrent writes to the register, in order for the state to eventually converge, all the replicas should agree on the order in which the concurrent write operations are applied. Thus, the implementations will follow an arbitration strategy, in this case the *Last Writer Wins* strategy where the replicas will typically tag every write operation with a time-stamp which is consistent with the happened-before relation. Thus, on a Read request from a client, the replicas return the value corresponding to the Write operation with the greatest timestamp.

- $\mathsf{Updates} = \{\mathsf{Write}\}$
- $\mathsf{Queries} = \{\mathsf{Read}\}$ and $arity(\mathsf{Read}) = 0$
- **Specification:** Let $U = (\mathcal{E}, \mathsf{vis}, \mathsf{arb})$ be a context of in the abstract execution of a well-defined run of LWW-registers. Let $W(U) = \{e \in \mathcal{E}(U) \mid Op(e) = \mathsf{Write}\}$. Then,

$$Spec_{LWWReg}(U, Read, \bot) = Args(max^{\mathsf{arb}}(max(U|_{(W(U))})))$$

MV Registers

A *Multi-Valued Register (MV Register)* is a replicated *read-write* register which like the *LWW Register* provides a Write method to update the state of the register and a Read method to query the state of the register. However, unlike the *LWW Register*, the *MV Register* does not arbitrate between the concurrent writes. Instead, when a replica of the *MV Register* receive a *Read*() request from the client, it will return a set of values corresponding to the maximal concurrent Write requests that it is aware of. Thus, the response may contain more than one value and it is up to the clients to perform the reconciliation by issuing a fresh Write. The specification of the *MV registers* provided below is arbitration agnostic.

- $\mathsf{Updates} = \{\mathsf{Write}\}$

- $\mathsf{Queries} = \{\mathsf{Read}\}$ and $arity(\mathsf{Read}) = 0$
- **Specification:** Let V be a view in the trace of a well-defined run of MV-registers. Let $W(V) = \{e \in V \mid Op(e) = \mathsf{Write}\}$. Then,

$$Spec_{MVReg}(V, Read, \bot) = \bigcup Args(max(V|_{(W(V))}))$$

OR sets

An OR set is a distributed set that follows the "add-wins" semantics for concurrent adds and deletes of the same element. The specification of OR-Sets is arbitration agnostic.

- $\mathsf{Updates} = \{\mathsf{add}, \mathsf{delete}\}$
- $\mathsf{Queries} = \{\mathsf{contains}\}$ and $arity(\mathsf{contains}) = 1$
- **Specification:** For an element $x \in \mathsf{Univ}$, and a view $V = (\mathcal{E}, \mathsf{vis})$ in the trace of any well-defined run of OR-set, we define the set $\mathcal{E}_x = \{e \in V \mid x \in Args(e)\}$. Then,

$$Spec_{ORSet}(V, \mathsf{contains}, x) = True \iff \exists e \in \mathcal{E}_x : Op(e) = \mathsf{add} \wedge \forall e' \in \mathcal{E}_x : e \xrightarrow{\mathsf{vis}} e' \implies Op(e') \neq \mathsf{delete}$$

Thus, this specification is equivalent to the specification defined in Chapter 3 (Definition 26) where at any event in the trace, a $\mathsf{contains}(x)$ query returns True iff there exists an $\mathsf{add}(x)$ event in the view which doesn't have a *covering delete*.

If the updates are causally delivered, this specification reduces to checking the existence of maximal $\mathsf{add}(x)$ event in $V|_{\mathcal{E}_x}$. Thus, in the case where the updates are causally delivered,

$$Spec_{ORSet}(V, Contains, x) = \mathsf{True} \iff \exists e \in max(V|_{\mathcal{E}_x}) : Op(e) = \mathsf{add}$$

In the next subsection, we discuss the correctness of a run and an implementation of a replicated data type with respect to its declarative specification.

4.2.2 Correctness of implementations with respect to a Specification

A run of a replicated data type is said to be correct with respect to an abstract specification if the return value of every query operation in the run matches the return value provided by the abstract specification when applied to the context of the event corresponding to that query operation in the execution of that run. We formally define it below.

Definition 98 (Valid Runs defined by a Specification)**.** *Let $\alpha = (\rho, \varphi)$ be a run of a replicated data type $\mathcal{D}$. Let $T = (\mathcal{E}_\rho, vis)$ be its trace. Let $F_{\mathsf{arb}}()$ be an arbitration strategy with $\mathsf{arb} = F_{\mathsf{arb}}(\alpha, T)$. Thus the context of α defined by F_{arb} is $A = (\mathcal{E}_\rho, \mathsf{vis}, \mathsf{arb})$.*

We say that α is correct with respect to a specification $Spec_{\mathcal{D}}$ and the arbitration strategy $F_{\mathsf{arb}}()$ iff for all events $e \in \mathcal{E}_\rho$ with $Op(e) \in \mathsf{Queries}$ we have have

$$Ret(e) = Spec_{\mathcal{D}}(Ctxt_A(e), Op(e), Args(e))$$

The set of all valid runs of the replicated data type $\mathcal{D}$ with respect to the specification $Spec_{\mathcal{D}}$ and an arbitration strategy $F_{\mathsf{arb}}()$ is denoted by $Runs(\mathcal{D}, Spec_{\mathcal{D}}, F_{\mathsf{arb}})$.

Further, if $Spec_{\mathcal{D}}$ is arbitration-agnostic, then α is valid with respect to this specification iff with $T = (\mathcal{E}_\rho, \mathsf{vis})$ being the trace of α, for every event query event $e \in \mathcal{E}_\rho$ we have,

$$Ret(e) = Spec_{\mathcal{D}}(\partial_T(e), Op(e), Args(e))$$

The set of all valid runs of $\mathcal{D}$ as per the arbitration-agnostic specification $Spec_{\mathcal{D}}$ is denoted by $Runs(\mathcal{D}, Spec_{\mathcal{D}})$

We can now formally define the notion of correctness of an implementation is correct with respect to a given specification. Note that the behaviour of an implementation of a replicated data type is the set of all its runs. Thus, for an implementation to be correct, all its runs should be correct.

Definition 99 (Correctness of an implementation)**.** *Let $\mathcal{D}_I$ be an implementation of $\mathcal{D}$. $\mathcal{D}_I$ is said to be correct with respect to a specification $Spec_{\mathcal{D}}$ iff*

- *$Spec_{\mathcal{D}}$ is an arbitration agnostic specification and $Runs(\mathcal{D}_I) \subseteq Runs(\mathcal{D}, Spec_{\mathcal{D}})$.*

  OR

- *There exists an arbitration strategy $F_{\mathsf{arb}}()$ such that $Runs(\mathcal{D}_I) \subseteq Runs(\mathcal{D}, Spec_{\mathcal{D}}, F_{\mathsf{arb}})$.*

Note that at the end of some run of an implementation of a replicated data type, if a pair of replicas have the same *causal past*, then, the *contexts* of the respective latest events of the two replicas in any *execution* of the run will be identical. Thus, if that implementation is correct with respect to a declarative specification, then any query operations performed at these two replicas at this point will return the same result since the return value of a query in any correct run is defined by the declarative specification of that replicated data type and the declarative specification only concerns with the context. Thus, we can prove the following key result that shows that any implementation of a replicated data type that is correct with respect to the declarative specification satisfies strong eventual consistency.

Theorem 100. *If an implementation of a replicated data is correct with respect to specification, then it satisfies strong eventual consistency.*

Proof. Suppose $\mathcal{D}_I$ is an implementation of a replicated data-type $\mathcal{D}$ and it correct with respect to a update-based specification $Spec_{\mathcal{D}}$. Then by definition, there exists an arbitration strategyh $F_{\mathsf{arb}}()$ such that $Runs(\mathcal{D}_I) \subseteq Runs(\mathcal{D}, Spec_{\mathcal{D}}, F_{\mathsf{arb}})$.

Consider a run $\alpha = (\rho, \varphi)$ of $\mathcal{D}_I$. Let $T = (\mathcal{E}_\rho, \mathsf{vis})$ be the trace of this run. Let $\mathsf{arb} = F_{\mathsf{arb}}(\alpha, T)$ be the arbitration relation over the events of this trace defined by $F_{\mathsf{arb}}()$. Thus, in the execution of the run $A_\alpha = (\mathcal{E}_\rho, \mathsf{vis}, \mathsf{arb})$, for any query event e, $Ret(e) = Spec_{\mathcal{D}}(Ctxt_{A_\alpha}(e), Op(e), Args(e))$.

Let $q \in \mathsf{Queries}$ be a query method and let $f_q()$ be the corresponding function provided by $\mathcal{D}_I$ such that whenever a query operation q is submitted with arguments *args*, the replica whose state is S computes the return value $f_q(S, args)$.

Suppose there exist integers i and j with $Rep(\rho[i]) = r$ and $Rep(\rho[j]) = r'$ such that $\mathsf{Past}_\alpha(\rho[i]) = \mathsf{Past}_\alpha(\rho[j])$. Since q is an arbitrary query method, if we show that $f_q(S_{r_i}(\alpha, i), args) = f_q(S_{r_i}(\alpha, i), args)$ then it follows that $S_{r_i}(\alpha, i) \cong S_{r_j}(\alpha, j)$.

Since the runs of an implementation are prefix closed, it implies that $(\rho[1, \ldots, i], \varphi|_{[1,\ldots,i]})$ and $(\rho[1, \ldots, j], \varphi|_{[1,\ldots,j]})$ are valid runs of $\mathcal{D}_I$.

Since any valid run can be extended by a query operation applied at any site, for a query $q \in \mathsf{Queries}$ and arg $\in \mathsf{Univ}^*$, we can define operations o and o' such that $Rep(o) = r$, $Rep(o') = r'$ with $Op(o) = Op(o') = q$ and $Args(o) = Args(o') = args$.

Then, the run $\alpha_1 = (\rho[1, \ldots, i].o, \varphi|_{[1,\ldots,i]})$ is a valid extention of $(\rho[1, \ldots, i], \varphi|_{[1,\ldots,i]})$ for some value $v = Ret(o)$. Similarly, the run the run $\alpha_2 = (\rho[1, \ldots, j].o', \varphi|_{[1,\ldots,j]})$ is a valid extention of $(\rho[1, \ldots, j], \varphi|_{[1,\ldots,j]})$ for some value $v' = Ret(o)$. Thus $\alpha_1, \alpha_2 \in Runs(\mathcal{D}_I)$.

To show query equivalence of states $S_r(\alpha, i)$ and $S_{r'}(\alpha, j)$ is same as showing query equivalence of states $S_r(\alpha_1, i+1)$ and $S_{r'}(\alpha_2, j+1)$ which is the same as showing $Ret(o) = Ret(o')$.

Let $\mathcal{E}_1 = \mathcal{E}_{\rho[1,\ldots,i].o}$ and $\mathcal{E}_2 = \mathcal{E}_{\rho[1,\ldots,j].o'}$

Let e be the query event in $\mathcal{E}_1$ corresponding to o and e' be the query event in $\mathcal{E}_2$ corresponding to o'.

Let $T_1 = (\mathcal{E}_1, \mathsf{vis}_1)$ be the trace of α_1 and $T_2 = (\mathcal{E}_2, \mathsf{vis}_2)$ be the trace of α_2. Let $\mathsf{arb}_1 = \mathsf{arb}|_{\mathcal{E}_1}$ and $\mathsf{arb}_2 = \mathsf{arb}|_{\mathcal{E}_2}$. Let $A_1 = (\mathcal{E}_1, \mathsf{vis}_1, \mathsf{arb}_1)$ be an execution of α_1 consistent with the execution A of α. Similarly, let, $A_2 = (\mathcal{E}_2, \mathsf{vis}_2, \mathsf{arb}_2)$ be the execution of α_2 that is consistent with A of α.

Since $\mathsf{Past}_\alpha(\rho[i]) = \mathsf{Past}_\alpha(\rho[j])$, we have by construction, $\mathsf{Past}_{\alpha_1}(o) = \mathsf{Past}_{\alpha_2}(o')$. By the definition of the visibility relation, this implies that

$$Ideal_{T_1}(e) = Ideal_T(e_i) = Ideal_T(e_j) = IdealT_2e'$$

From this, we have

$$\mathsf{vis}_1|_{Ideal_{T_1}(e)} = \mathsf{vis}|_{Ideal_T(e_i)} = \mathsf{vis}|_{Ideal_T(e_i)} = \mathsf{vis}_2|_{Ideal_{T_2}(e')}$$

and

$$\mathsf{arb}_1|_{Ideal_{T_1}(e)} = \mathsf{arb}|_{Ideal_T(e_i)} = \mathsf{arb}|_{Ideal_T(e_i)} = \mathsf{arb}_2|_{Ideal_{T_2}(e')}$$

This implies that the $Ctxt_{A_1}(e) = Ctxt_{A_2}(e')$.

Thus we have,

$$\begin{aligned}
Ret(o) &= Ret(e) \\
&= Spec_{\mathcal{D}}(Ctxt_{A_1}(e), Op(e), Args(e)) \\
&= Spec_{\mathcal{D}}(Ctxt_{A_1}(e), q, args) \\
&= Spec_{\mathcal{D}}(Ctxt_{A_2}(e'), q, args) \\
&= Spec_{\mathcal{D}}(Ctxt_{A_2}(e'), Op(e'), Args(e')) \\
&= Ret(e') \\
&= Ret(o')
\end{aligned}$$

Thus $Ret(o) = Ret(o')$. Since o and o' were query events for an arbitrary query q with an arbitrarily chosen argument arg, this is true for all queries and all arguments.

Thus, $S_r(\alpha_1, i+1)$ and $S_{r'}(\alpha_2, j+1)$ are query equivalent. By construction this happens iff $S_r(\alpha, i)$ and $S_{r'}(\alpha, j)$ are query-equivalent.

Thus, we have shown that in any valid run α of the implementation $\mathcal{D}_I$, $\mathsf{Past}_\alpha(\rho[i]) = \mathsf{Past}_\alpha(\rho[j]) \implies S_{r_i}(\alpha, i) \cong S_{r_j}(\alpha, j)$.

Hence the implementation $\mathcal{D}_I$ by satisfies Strong Eventual Consistency. □

For the remainder of this thesis, we shall be focussing on the problem of verifying the correctness of a given implementation of a replicated data-type with respect to an arbitration-agnostic specification. Thus, in the subsequent section and the next chapter, when we use the term *context* of an event in the *execution*, it refers to the *view* of that event in the *trace* of the execution. From the proposition we just proved, it entails that any correct implementation with respect to the declarative specification satisfies strong eventual consistency. In the next section we shall provide an algorithm to construct a reference implementation of a replicated data type from its declarative specification.

4.3 Reference Implementations of Replicated Data Types from Specifications

For the purpose of effective automatic verification of a given implementation, it is reasonable to assume that the declarative specification is a computable function. Typically such a function would do two things :

- Based on the query method and argument to the query method, the computable function would identify the subtrace from the view to limit the scope of the decision. For example, in the case of OR-Sets, in order to answer the query $\mathsf{contains}(x)$ it is sufficient to restrict the view to only $\mathsf{add}(x)$ and $\mathsf{delete}(x)$ events in the view. The other events are not relevant for this query. Furthermore, we can restrict our scope to only those $\mathsf{add}(x)$ events which are not superceded by a $\mathsf{delete}(x)$ event in that view.
- Based on the subtrace of the view thus generated, the function would provide the response of the given query and the argument. In the aforementioned example of OR-Sets, by looking into the subtrace of view consisting of only those $\mathsf{add}(x)$ events which are not superceded by a $\mathsf{delete}(x)$, the specification can return True for $\mathsf{contains}(x)$ if and only if the subtrace computed earlier is non-empty.

Since a view of an event is a finite object, the function to extract tha relevant subtrace from the view is computable. Furthermore, the function to provide the return value of the query for given arguments based on this finite relevant subtrace is also computable. Thus, the declarative specification is computable. Formally we define it as follows

Definition 101 (Computable Specification)**.** *A specification $Spec_{\mathcal{D}}$ is said to be computable if there exists computable functions*
$\mathsf{RelevantCtxt} : Ctxt(\mathcal{D}) \times \mathsf{Queries} \times \mathsf{Univ}^* \to Ctxt(\mathcal{D})$ *and*
$\mathsf{ComputeRet} : Ctxt(\mathcal{D}) \times \mathsf{Queries} \times \mathsf{Univ}^* \to \mathsf{Rets}$ *such that*
for every context $U \in Ctxt(\mathcal{D})$, query $q \in \mathsf{Queries}$, and arguments to the query $args \in \mathsf{Univ}^*$ with $U' = \mathsf{RelevantCtxt}(U, q, args)$ we have

1. $U' \subseteq U$
2. $Spec_{\mathcal{D}}(U, q, args) = \mathsf{ComputeRet}(U', q, args)$
3. *If there exists $U'' \in Ctxt(\mathcal{D})$ such that $U' \subseteq U'' \subseteq U$, then* $U' = \mathsf{RelevantCtxt}(U'', q, args)$
4. *If there exists a U'' that is isomorphic to U', then,*

$$\mathsf{ComputeRet}(U', q, args) = \mathsf{ComputeRet}(U'', q, args)$$

The RelevantCtxt *function extracts the relevant sub-context that is required to answer the query, and the* ComputeRet *function computes the value that needs to be returned when the query is made on the relevant context.*

Condition 1 requires that the context computed by the RelevantCtxt() *function be a sub-context of the given context.*

Condition 2 says that the value returned by the specification for a given context is the value computed by the ComputeRet() *function applied to the relevant context.*

Condition 3 describes the consistency property of the RelevantCtxt() *function. Note that* RelevantCtxt() *is a filter function that retains only a subcontext of a given context that is required to answer this query. Thus if a subcontext U' happens to be a relevant context of a larger context U, then, for any other subcontext U'' which contains the original relevant context U' and is contained in the larger context U, the relevant context of U'' computed by* RelevantCtxt() *is exactly same as the subcontext U'.*

Condition 4 requires that ComputeRet() *function, when applied to any two contexts that are isomorphic to each other, the computed return value is identical. Thus, the* ComputeRet() *is only concerned with the* structure *of the context and not which replicas were the update operations applied on. This follows from the requirement that the specification function treats any pair of isomorphic contexts alike.*

Suppose $Spec_{\mathcal{D}}$ is an arbitration-agnostic, update-based, computable specification. We shall now provide an algorithm to generate an implementation of $\mathcal{D}$ whose runs that satisfy causal-delivery are correct with respect to $Spec_{\mathcal{D}}$. In the next chapter, we will show how the size of this reference implementation may be bounded in order to be used for verification of the correctness of given implementations of replicated data types.

Definition 102 (Reference Implementation with Causal Delivery)**.** *We shall provide an op-based implementation of the RDT $\mathcal{D}$ with respect to a specification $Spec_{\mathcal{D}}$. We have replicas $\mathcal{R} = [1, \ldots, N]$.*

The state at each replica S_r is an acyclic graph $G_r = (\mathcal{E}_r, \mathsf{vis}_r)$ that corresponds to the view at the replica restricted to only the update events. Initially $\forall r \in \mathcal{R} : G_r = (\emptyset, \emptyset)$

In Algorithm 102, the Lines 5–11 provide the implementation of the function f_u corresponding to the update method u. On a new update request the replica generates a new event (line 6). The replica then adds the new event to its set of events (line 8) and adds new edges between all the other events in the event set to the new event (line 9). This is equivalent to updating the visibility relation. The replica then updates its set of events and the set of edges between these events to the newly computed event set and newly computed set of edges (line 10). The replica then broadcasts this updated event set and the edge relation to all the other replicas (line 11).

Lines 13–17 provide the implementation of f_u^{rcv} corresponding to the update receive method of the update method u. The replica updates the event set and the edge set with

Algorithm 5 Reference Implementation of Replicated Data Type with Causal Delivery

Reference Implementation with Causal Delivery at replica r

$\mathcal{E}_r$ is a set of nodes corresponding to update events. Initially $\emptyset$
vis_r is an acyclic binary relation over $\mathcal{E}_r$. Initially $\emptyset$

Implementation for an update method $u \in \mathsf{Updates}$**.**
$f_u(\text{arg} \in \mathsf{Univ}^*)$
    Let $e = \mathsf{NewEvent}()$.
    Set $Op(e) := u$, $Args(e) := \text{arg}$
    Let $\mathcal{E}_r^{new} := \mathcal{E}_r \cup \{e\}$.
    Let $\mathsf{vis}_r^{new} := \mathsf{vis}_r \cup \{(e', e) \mid e' \in \mathcal{E}_r^{new}\}$
    Set $(\mathcal{E}_r, \mathsf{vis}_r) := (\mathcal{E}_r^{new}, \mathsf{vis}_r^{new})$
    Broadcast $u.\mathbf{send}(\mathcal{E}_r, \mathsf{vis}_r)$

Implementation of update-receive method for the update u
$f_u^{rcv}((\mathcal{E}', \mathsf{vis}'))$:
    $\mathcal{E}_r^{new} := \mathcal{E}_r \cup \mathcal{E}'$
    Let $\mathsf{vis}_r^{new} := \mathsf{vis}_r \cup \mathsf{vis}'$.
    Set $(\mathcal{E}_r, \mathsf{vis}_r) := (\mathcal{E}_{merge}, \mathsf{vis}_{merge})$

Implementation of a query $q \in \mathsf{Updates}$ **at replica** r**.**
$f_q(\text{arg} \in \mathsf{Univ}^*)$:
    Let $\mathsf{Ret} = Spec_{\mathcal{D}}((\mathcal{E}_r, \mathsf{vis}_r), q, \text{arg})$
    return ret

with the set of events and the set of edges that it has received as a part of the auxiliary information(lines 15–17).

Lines 19–22 provides the implementation of f_q corresponding to the query method q. The replica applies the specification function of the replicated data type on the context that it maintains in the form of the event graph to compute a value (line 21) which is returned to the client that had invoked the query method (line 22).

We will show that in any run of the reference implementation where updates are causally-delivered, the state at every replica is the view at the replica restricted to update-events.

Lemma 103. *Let $\alpha = (\rho, \varphi)$ be any run of the reference implementation where the updates are causally delivered. Let $G_r^i = (\mathcal{E}_r^i, \mathsf{vis}_r^i)$ denote the state of the replica at the end of the run $\rho[1, \ldots, i]$. Let $T = (\mathcal{E}_\rho, \mathsf{vis})$ be the trace of α.*

Let $\mathcal{E}_{refer}(\alpha) = \bigcup_{r \in \mathcal{R}} \bigcup_{i=1}^{|\rho|} \mathcal{E}_r^i$ denote the set of all the events generated by the NewEvent() *function of the reference implementation during the course of this run.*

Let $g : [1, \ldots, |\rho|] \rightharpoonup \mathcal{E}_{refer}(\alpha)$ be a partial function that associates each update operation with unique event generated by NewEvent() *method of the reference implementation. Then if $Rep(\rho[i]) = r$,*

- $\mathcal{E}_r^i = \{g(j) \mid e_j \in Ideal_T(e_i)\}$
- *For $g(j), g(k) \in \mathcal{E}_r^i$, $(g(j), g(k)) \in \mathsf{vis}_r^i \iff e_j \xrightarrow{\mathsf{vis}} e_k$.*

Proof. We shall prove this by induction over $|\rho|$. Note that for $|\rho| = 0$, the result trivially holds.

Assume that the result holds for all $|\rho| < i$. We now look at the case where $|\rho| = i$. We consider the following cases based on the nature of the operation of $\rho[i]$. Let $i' < i$ be the maximal integer such that $Rep(\rho[i']) = Rep(\rho[i]) = r$.

Case $Op(\rho[i]) \in$ Queries : In this case $\mathcal{E}_r^i = \mathcal{E}_r^{i-1} = \mathcal{E}_r^{i'}$ and $\mathsf{vis}_r^i = \mathsf{vis}_r^{i-1} = \mathsf{vis}_r^{i'}$. Further, by definition of the ideal, $Ideal_T(e_i) = Ideal_T(e_{i'})$.

By induction hypothesis, since $\mathcal{E}_r^{i'} = \{g(j) \mid e_j \in Ideal_T(e_{i'})\}$.

Thus $\mathcal{E}_r^i = \mathcal{E}_r^{i'} = \{g(j) \mid e_j \in Ideal_T(e_{i'})\} = \{g(j) \mid e_j \in Ideal_T(e_i)\}$.

For $g(j), g(k) \in \mathcal{E}_r^i$, $(g(j), g(k)) \in \mathsf{vis}_r^i$ iff $g(j), g(k) \in \mathcal{E}_r^{i'}$ and $(g(j), g(k)) \in \mathsf{vis}_r^{i'}$ which by induction hypothesis is true iff $e_j \xrightarrow{\mathsf{vis}} e_k$.

Case $Op(\rho[i]) \in$ Updates : In this case $\mathcal{E}_r^i = \mathcal{E}_r^{i'} \cup \{g(i)\}$. Also, $Ideal_T(e_i) = Ideal_T(e_{i'}) \cup \{e_i\}$.

By induction hypothesis, $\mathcal{E}_r^{i'} = \{g(j) \mid e_j \in Ideal_T(e_{i'})\}$. Thus $\mathcal{E}_r^i = \mathcal{E}_r^{i'} \cup \{g(i)\} = \{g(j) \mid e_j \in Ideal_T(e_{i'})\} \cup \{g(i)\} = \{g(j) \mid e_j \in Ideal_T(e_i)\}$.

For $j, k < i$, by induction hypothesis, $g(j), g(k) \in \mathcal{E}_r^{i'}$, we have $(g(j), g(k)) \in \mathsf{vis}_r^{i'}$ iff $e_j \xrightarrow{\mathsf{vis}} e_k$.

Now $\mathsf{vis}_r^i = \mathsf{vis}_r^{i'} \cup \{(g(j), g(i)) \mid g(j) \in \mathcal{E}_r^i\}$

Thus, for $g(j), g(k) \in \mathcal{E}_r^i$, we have $(g(j), g(k)) \in \mathsf{vis}_r^i$ iff $j, k < i$ and $(g(j), g(k)) \in \mathsf{vis}_r^{i'}$ or $j \leq i, k = i$ and $\{(g(j), g(i)) \mid g(j) \in \mathcal{E}_r^i\}$. The former is the case iff by induction hypothesis $e_j \xrightarrow{\mathsf{vis}} e_k$. The latter is true iff $e_j \xrightarrow{\mathsf{vis}} e_k$ from the fact that $\mathcal{E}_r^i = \{g(j) \mid e_j \in \mathit{Ideal}_T(e_i)\}$ and by the definition of $\mathit{Ideal}()$, $\forall e_j \in \mathit{Ideal}_T(e_i)$, $e_j \xrightarrow{\mathsf{vis}} e_i$.

Case $\mathit{Op}(\rho[i]) = \mathbf{usend}$ Note that in this case, $i' = i - 1$ and $\mathit{Op}(\rho[i']) \in \mathsf{Updates}$. $\mathcal{E}_r^i = \mathcal{E}_r^{i-1} = \mathcal{E}_r^{i'}$ and $\mathsf{vis}_r^i = \mathsf{vis}_r^{i-1} = \mathsf{vis}_r^{i'}$. Further, by definition of the ideal, $\mathit{Ideal}_T(e_i) = \mathit{Ideal}_T(e_{i'})$.

By induction hypothesis, since $\mathcal{E}_r^{i'} = \{g(j) \mid e_j \in \mathit{Ideal}_T(e_{i'})\}$.

Thus $\mathcal{E}_r^i = \mathcal{E}_r^{i'} = \{g(j) \mid e_j \in \mathit{Ideal}_T(e_{i'})\} = \{g(j) \mid e_j \in \mathit{Ideal}_T(e_i)\}$.

For $g(j), g(k) \in \mathcal{E}_r^i$, $(g(j), g(k)) \in \mathsf{vis}_r^i$ iff $g(j), g(k) \in \mathcal{E}_r^{i'}$, $(g(j), g(k)) \in \mathsf{vis}_r^{i'}$ which by induction hypothesis is true iff $e_j \xrightarrow{\mathsf{vis}} e_k$.

Case $\mathit{Op}(\rho[i]) = \mathbf{ureceive:}$ Let i'' be such that $\varphi(\rho[i]) = \rho[i'']$. Let $\mathit{Rep}(\rho[i'']) = r''$. Then, by construction, $\mathcal{E}_r^i = \mathcal{E}_r^{i'} \cup \mathcal{E}_{r''}^{i''}$. Since α is a run where all the udpates are causally delivered, from Corollary 89 $\mathit{Ideal}_T(e_i) = \mathit{Ideal}_T(e_{i'}) \cup \mathit{Ideal}_T(e_{i''})$. By induction hypothesis, $\mathcal{E}_r^{i'} = \{g(j) \mid e_j \in \mathit{Ideal}_T(e_{i'})\}$ and $\mathcal{E}_{r''}^{i''} = \{g(j) \mid e_j \in \mathit{Ideal}_T(e_{i''})\}$. Thus $\mathcal{E}_r^i = \mathcal{E}_r^{i'} \cup \mathcal{E}_{r''}^{i''} = \{g(j) \mid e_j \in \mathit{Ideal}_T(e_{i'})\} \cup \{g(j) \mid e_j \in \mathit{Ideal}_T(e_{i''})\} = \{g(j) \mid e_j \in \mathit{Ideal}_T(e_{i'}) \cup \mathit{Ideal}_T(e_{i''})\} = \{g(j) \mid e_j \in \mathit{Ideal}_T(e_i)\}$. Thus, $\mathcal{E}_r^i = \{g(j) \mid e_j \in \mathit{Ideal}_T(e_i)\}$

From the update-receive method of the reference implementation $\mathsf{vis}_r^i = \mathsf{vis}_r^{i'} \cup \mathsf{vis}_{r''}^{i''}$.

Now $g(j), g(k) \in \mathcal{E}_r^i$ iff

1. either $g(j), g(k) \in \mathcal{E}_r^{i'}$ or
2. $g(j), g(k) \in \mathcal{E}_{r''}^{i''}$ or
3. $g(j) \in \mathcal{E}_{r''}^{i''} \setminus \mathcal{E}_r^{i'}$ and $g(k) \in \mathcal{E}_r^{i'} \setminus \mathcal{E}_{r''}^{i''}$
4. $g(k) \in \mathcal{E}_{r''}^{i''} \setminus \mathcal{E}_r^{i'}$ and $g(j) \in \mathcal{E}_r^{i'} \setminus \mathcal{E}_{r''}^{i''}$

Now, if $g(j), g(k) \in \mathcal{E}_r^{i'}$ $(g(j), g(k)) \in \mathsf{vis}_r^i \iff (g(j), g(k)) \in \mathsf{vis}_r^{i'}$ which by induction hypothesis is true iff $e_j \xrightarrow{\mathsf{vis}} e_k$.

Similarly, if $g(j), g(k) \in \mathcal{E}_{r''}^{i''}$ $(g(j), g(k)) \in \mathsf{vis}_r^i \iff (g(j), g(k)) \in \mathsf{vis}_{r''}^{i''}$ which by induction hypothesis is true iff $e_j \xrightarrow{\mathsf{vis}} e_k$.

If $g(j) \in \mathcal{E}_{r''}^{i''} \setminus \mathcal{E}_r^{i'}$ and $g(k) \in \mathcal{E}_r^{i'} \setminus \mathcal{E}_{r''}^{i''}$ or $g(k) \in \mathcal{E}_{r''}^{i''} \setminus \mathcal{E}_r^{i'}$ and $g(j) \in \mathcal{E}_r^{i'} \setminus \mathcal{E}_{r''}^{i''}$ then $g(j)$ and $g(k)$ are not related in vis_r^i. Thus if we show that e_j and e_k are not related by vis, the proof for this case follows.

Suppose $g(j) \in \mathcal{E}_{r''}^{i''} \setminus \mathcal{E}_r^{i'}$ and $g(k) \in \mathcal{E}_r^{i'} \setminus \mathcal{E}_{r''}^{i''}$. By induction hypothesis, $e_j \in \mathit{Ideal}_T(e_{i''}) \setminus \mathit{Ideal}_T(e_{i'})$ and $e_k \in \mathit{Ideal}_T(e_{i'}) \setminus \mathit{Ideal}_T(e_{i''})$.

Now suppose $e_k \xrightarrow{\mathsf{vis}} e_j$, then, $e_k \in \mathit{Ideal}_T(e_{i''})$ since ideals of causally delivered runs are downward closed. But we are considering the case where $e_k \in \mathit{Ideal}_T(e_{i'}) \setminus \mathit{Ideal}_T(e_{i''})$ and thus, $e_k \notin \mathit{Ideal}_T(e_{i''})$. Thus it is not the case that $e_k \xrightarrow{\mathsf{vis}} e_j$.

Similarly, if it were the case that $e_j \xrightarrow{\mathsf{vis}} e_k$, then by downward closure, $e_j \in \mathit{Ideal}_T(e_{i'})$ which contradicts the fact that $e_j \in \mathit{Ideal}_T(e_{i''}) \setminus \mathit{Ideal}_T(e_{i'})$. Thus this is not the case.

Thus, for $g(j) \in \mathcal{E}_{r''}^{i''} \setminus \mathcal{E}_r^{i'}$ and $g(k) \in \mathcal{E}_r^{i'} \setminus \mathcal{E}_{r''}^{i''}$, e_j and e_k are not related in vis.

By similar line of reasoning we can show that for $g(k) \in \mathcal{E}_{r''}^{i''} \setminus \mathcal{E}_r^{i'}$ and $g(j) \in \mathcal{E}_r^{i'} \setminus \mathcal{E}_{r''}^{i''}$, e_j and e_k are not related in vis.

Thus, from this, and the earlier proof we can conclude that for $g(j), g(k) \in \mathcal{E}_r^i$, we have $(g(j), g(k)) \in \mathsf{vis}_r^i$ iff $e_j \xrightarrow{\mathsf{vis}} e_k$

With this we have shown that the result holds for $|\rho| = i$. By the principle of mathematical induction, the result holds for all runs α where the updates are causally delivered. □

Thus, we have shown that the state maintained by each of the replicas is the view at that replica. Since $\mathit{Spec}_{\mathcal{D}}$ is an arbitration agnostic specification, to correctly answer any query posed to the replica as per the specification, the view of the replica is sufficient. We will now formally show that the reference implementation is correct with respect to $\mathit{Spec}_{\mathcal{D}}$

Theorem 104 (Correctness of the Reference Implementation)**.**

$$\mathit{Runs}(\mathcal{D}_I^{\mathit{refer}}) \subseteq \mathit{Runs}(\mathcal{D}, \mathit{Spec}_{\mathcal{D}})$$

Proof. We will show this through induction over the length of the run in $\mathit{Runs}(\mathcal{D}_I^{\mathit{refer}})$. In case of the emptyrun, it is trivially true, since it belongs to both $\mathit{Runs}(\mathcal{D}_I^{\mathit{refer}})$ and $\mathit{Runs}(\mathcal{D}, \mathit{Spec}_{\mathcal{D}})$.

Suppose the result is correct for all runs in $\mathit{Runs}(\mathcal{D}_I^{\mathit{refer}})$ whose length is atmost n.

We will consider a run $\alpha = (\rho, \varphi)$ from $\mathit{Runs}(\mathcal{D}_I^{\mathit{refer}})$ such that $|\rho| = n$.

Note that if $\mathit{Op}(\rho[n]) \notin \mathsf{Queries}$ then since $\alpha \in \mathit{Runs}(\mathcal{D}_I^{\mathit{refer}})$, we have $\alpha' = (\rho[1, \ldots, n-1], \varphi|_{\rho[1,\ldots,n-1]}) \in \mathit{Runs}(\mathcal{D}_I^{\mathit{refer}})$ since $\mathit{Runs}(\mathcal{D}_I^{\mathit{refer}})$ is prefix closed. By inductive hypothesis, this implies that $\alpha' \in \mathit{Runs}(\mathcal{D}, \mathit{Spec}_{\mathcal{D}})$. Thus, for every $i < n$, if $\rho[i]$ is a query operation, then $\mathit{Ret}(\rho[i])$ is as per the specification. from this we can conclude that $\alpha \in \mathit{Runs}(\mathcal{D}, \mathit{Spec}_{\mathcal{D}})$ as all the return values of all query operations in this run are as per the specification.

Thus, the only case we have to consider is when $\mathit{Op}(\rho[n]) \in \mathsf{Queries}$. Let $\mathit{Rep}(\rho[n]) = r$. Let the state of replica r of the reference implementation be $(\mathcal{E}_r^n, \mathsf{vis}_r^n)$. Let $T = (\mathcal{E}_\rho, \mathsf{vis})$

be the trace of α. By definition, $\partial_T(e_n) = (Ideal_T(e_n), \mathsf{vis}|_{Ideal_T(e_n)})$. From the previous Lemma 103, we have shown that $(\mathcal{E}_r^n, \mathsf{vis}_r^n)$ is isomorphic to $(Ideal_T(e_n), \mathsf{vis}|_{Ideal_T(e_n)})$.

Thus,

$$Spec_{\mathcal{D}}((\mathcal{E}_r^n, \mathsf{vis}_r^n), Op(\rho[n]), Args(\rho[n])) = Spec_{\mathcal{D}}(\partial_T(e_n), Op(e_n), Args(e_n))$$

Now from the reference implementation

$$\begin{aligned} Ret(\rho[n]) &= Spec_{\mathcal{D}}((\mathcal{E}_r^n, \mathsf{vis}_r^n), Op(\rho[n]), Args(\rho[n])) \\ &= Spec_{\mathcal{D}}(\partial_T(e_n, Op(e_n)), Args(e_n)) \end{aligned}$$

From this, and the inductive hypothesis, the return values of all the the queries of α are as per the specification. Hence $\alpha \in Runs(\mathcal{D}, Spec_{\mathcal{D}})$. Since α is any arbitrary run of length n, this is true for all runs of $\mathcal{D}_I^{refer}$.

By the principle of mathematical induction, it follows that the result holds for all runs of $\mathcal{D}_I^{refer}$. Hence $Runs(\mathcal{D}_I^{refer}) \subseteq Runs(\mathcal{D}, Spec_{\mathcal{D}})$. □

In fact we can show that all the correct runs with respect to the specification are runs of the reference implementation if all updates in these runs are causally delivered.

Theorem 105. *If $\alpha = (\rho, \varphi) \in Runs(\mathcal{D}, Spec_{\mathcal{D}})$ and all the updates in α are causally delivered, then $\alpha \in Runs(\mathcal{D}_I^{refer})$.*

Proof. We shall show by induction that all the prefixes of α are in $Runs(\mathcal{D}_I^{refer})$. The empty prefix of α is trivially in $Runs(\mathcal{D}_I^{refer})$.

Suppose for all $j < i$, $\alpha_j = (\rho[1, \ldots, j], \varphi|_{\rho[1,\ldots,j]})$ it is the case that $\alpha_j \in Runs(\mathcal{D}_I^{refer})$.

Consider the run $\alpha_i = (\rho[1, \ldots, i], \varphi|_{\rho[1,\ldots,i]})$. Let $Rep(\rho[i]) = r$. Let T_i be the trace of α_i.

$Op(\rho[i]) \in \mathsf{Updates}$**:** Since any run of $\mathcal{D}_I^{refer}$ can be extended by an update operation, if $Op(\rho[i]) \in \mathsf{Updates}$ then $\alpha_i \in Runs(\mathcal{D}_I^{refer})$ since by inductive hypothesis $\alpha_{i-1} \in Runs(\mathcal{D}_I^{refer})$.

$Op(\rho[i]) =$ **usend:** Since α is a correct run as per the specification, this implies that $\rho[i-1] \in \mathsf{Updates}$. Since a run of the reference implementation ending with an update operation can be extended by the corresponding update-send operation, and since $\alpha_{i-1} \in Runs(\mathcal{D}_I^{refer})$, we can conclude that $\alpha_i \in Runs(\mathcal{D}_I^{refer})$

$Op(\rho[i]) =$ **ureceive** : Since α is a correct execution, if $\rho[i]$ is a **ureceive** operation, then, there exists $j < i$ such that $\rho[j] = \varphi(\rho[i])$. This implies that the broadcast of the update $\rho[j]$ wasn't delivered at replica r in α_{i-1}. Since α is a causally-delivered run, α_i is also causally delivered run. This implies that for all updates $\rho[k] \in \mathsf{Past}_{\alpha_j}(\rho[j])$, there exists a $k' : k < k' < i$ such that $Rep(\rho[k']) = r$ and $\varphi(\rho[k']) = \rho[k]$.

Since we can extend a run of the reference implementation with an update-receive operation if all the causally preceding updates of have been delivered in the original run, $\alpha_i \in Runs(\mathcal{D}_I^{refer})$ since by induction hypothesis, $\alpha_{i-1} \in Runs(\mathcal{D}_I^{refer})$ and $\rho[i]$ is such a **ureceive** operation.

$Op(\rho[i]) \in \mathsf{Queries}$: Since α is correct as per the specification,

$$Ret(\rho[i]) = Spec_{\mathcal{D}}(\partial_{T_i}(e_i), Op(e_i), Args(e_i))$$

We have shown via lemma 103 that $\partial_{T_i}(e_i)$ is isomorphic to the state of replica r, $(\mathcal{E}_r^i, \mathsf{vis}_r^i)$. Since $Spec_{\mathcal{D}}$ is a computable specification, by defintion,

$$Spec_{\mathcal{D}}(\partial_{T_i}(e_i), Op(e_i), Args(e_i)) = Spec_{\mathcal{D}}((\mathcal{E}_r^i, \mathsf{vis}_r^i), Op(e_i), Args(e_i))$$

This is same as $Spec_{\mathcal{D}}((\mathcal{E}_r^i, \mathsf{vis}_r^i), Op(\rho[i]), Args(\rho[i]))$ which is the value returned by the reference implementation for a query $Op(\rho[i])$ with arguments $Args(\rho[i])$ is issued at replica r at the end of the run α_{i-1}. Thus, the run obtained by extending α_{i-1} with $\rho[i]$ is a valid run of the reference implementation. Hence $\alpha_i \in Runs(\mathcal{D}_I^{refer})$.

□

Thus, the reference implementation covers all the correct causally delivered runs with respect to a specification.

Note: The algorithm for a reference implementation of a replicated data type $\mathcal{D}$ with causal delivery provided in Definition 102 can be slightly modified in order to arrive at a reference implementation whose runs are exactly the runs in $Runs(\mathcal{D}, Spec_{\mathcal{D}})$. We provide it below.

Definition 106 (Reference Implementation without delivery constraints)**.** *The state at each replica S_r is a labelled acyclic graph $G_r = (\mathcal{E}_r, \mathsf{vis}_r, Seen_r)$ where $Seen_r : \mathcal{E}_r \to \{\mathsf{True}, \mathsf{False}\}$ which identifies the update events from the graph that have been* seen *by the replica r, either through an update event or an update-receive event.*

If $\mathcal{E}_r^{Seen} = \{e \in \mathcal{E}_r \mid Seen_r(e) = \mathsf{True}\}$ and $\mathsf{vis}_r^{Seen} = \mathsf{vis}_r|_{\mathcal{E}_r^{Seen}}$, then, $(\mathcal{E}_r^{Seen}, \mathsf{vis}_r^{Seen})$ denotes the view of the replica r.

Initially $\forall r \in \mathcal{R} : G_r = (\emptyset, \emptyset, False)$ where $False()$ always returns False*.*

The operational details are presented in Algorithm 106.

At every replica,this generic implementation keeps track of the downward closed subtrace of the run which contains the view of the replica in the trace of the run. The $Seen_r()$ function indicates which of these update events in the downward closed subtrace have been delivered to replica r, either through an *update* operation or via an *update-receive* operation. Thus, even when the updates are delivered out of order, this implementation works as expected

Algorithm 6 Reference Implementation of a Replicated Data Type without any delivery constraints

Reference Implementation without delivery constraints replica r

1 $\mathcal{E}_r$ is a set of nodes corresponding to update events. Initially $\emptyset$
2 vis_r is an acyclic binary relation over $\mathcal{E}_r$. Initially $\emptyset$
3 $Seen_r : \mathcal{E}_r \to \{\mathsf{True}, \mathsf{False}\}$. Initially $Seen_r()$ is a function that always returns False.
4
5 **Implementation for an update method** $u \in \mathsf{Updates}$**.**
6 $u(\text{arg} \in \mathsf{Univ}^*)$
7 Let $e = \mathsf{NewEvent}()$.
8 Set $Op(e) := u$, $Args(e) := \text{arg}$
9 Let $\mathcal{E}_r^{new} := \mathcal{E}_r \cup \{e\}$.
10 Let $Seen_r^{new} : \mathcal{E}_r^{new} \to \{\mathsf{True}, \mathsf{False}\}$ such that for any $e' \in \mathcal{E}_r^{new}$,

$$Seen_r^{new}(e') = \begin{cases} \mathsf{True} \text{ if } e' = e \\ Seen_r(e') \text{ if } e' \in \mathcal{E}_r \end{cases}$$

11 Let $\mathsf{vis}_r^{new} := \mathsf{vis}_r \cup \{(e', e) \mid e' \in \mathcal{E}_r^{new} \wedge Seen_r^{new}(e') = \mathsf{True}\}$
12 Set $(\mathcal{E}_r, \mathsf{vis}_r, Seen_r) := (\mathcal{E}_r^{new}, \mathsf{vis}_r^{new}, Seen_r^{new})$
13 Broadcast $u.\mathbf{send}((\mathcal{E}_r, \mathsf{vis}_r, e))$
14
15 **Implementation of update-receive method for the update** u
16 $u.\mathbf{receive}((\mathcal{E}', \mathsf{vis}'))$:
17 Let $\mathcal{E}_r^{merge} := \mathcal{E}_r \cup \mathcal{E}'$
18 Let $\mathsf{vis}_r^{merge} := \mathsf{vis}_r \cup \mathsf{vis}'$.
19 Let $Seen_r^{merge} : \mathcal{E}_r^{merge} \to \{\mathsf{True}, \mathsf{False}\}$ such that for any $e'' \in \mathcal{E}_r^{merge}$,

$$Seen_r^{merge}(e'') = \begin{cases} \mathsf{True} \text{ if } e'' = e' \\ Seen_r(e'') \text{ if } e'' \in \mathcal{E}_r \\ \mathsf{False} \text{ Otherwise} \end{cases}$$

20 Set $(\mathcal{E}_r, \mathsf{vis}_r, Seen_r) := (\mathcal{E}_r^{merge}, \mathsf{vis}_r^{merge}, Seen_r^{merge})$
21
22 **Implementation of a query** $q \in \mathsf{Updates}$ **at replica** r**.**
23 $q(\text{arg} \in \mathsf{Univ}^*)$:
24 Let $\mathcal{E}_r^{Seen} = \{e \in \mathcal{E}_r \mid Seen_r(e) = \mathsf{True}\}$.
25 Let $\mathsf{vis}_r^{Seen} = \mathsf{vis}_r|_{\mathcal{E}_r^{Seen}}$
26 Let $v = Spec_{\mathcal{D}}((\mathcal{E}_r^{Seen}, \mathsf{vis}_r^{Seen}), q, \text{arg})$
27 **return** v

since all the queries are answered with respect to the subgraph corresponding to only the *seen* events which is exactly the the view of the replica at that point in the run. If all the the updates were causally delivered then $Seen_r(e) = \mathsf{True}$ for every $e \in \mathcal{E}_r$. Thus, in the case when the updates are causally delivered, this generic implementation reduces to the implementation given in Definition 102. For our purpose of obtaining bounded reference implementation, we would be restricting our attention to the implementation where the updates are causally delivered.

We explore these bounded reference implementations for replicated data types in the next two chapters.

5

Bounded Implementations of Replicated Data Types using Generalized Gossip

5.1 Introduction

Finite state abstractions have been widely studied in the context of formal verification. Model checking, for instance, uses techniques such as state space enumeration, abstract interpretation and symbolic execution to algorthmically verify if an abstract finite state system satisfies its specification. Finite state models such as automata over distributed words, communicating finite state machines and Petri nets have been successfully used to model and verify concurrent and distributed systems.

In this chapter, we focus on deriving a bounded implementation for *Commutative Replicated Data Types (CmRDTs)* whose replicas broadcast auxiliary information for every update they receive from a client.

In the field of asynchronous communicating automata, there is a well understood problem termed as the *gossip problem* first studied in [Mukund and Sohoni, 1997]. This problem involves an automaton keeping track of the latest information about every automaton in the system and correctly updating the latest information whenever it receives a communication from one of the other automatons in the system. In this chapter, we will define a generalization of the *gossip problem* introduced in [Mukund and Sohoni, 1997] and explore a bounded solution to this problem. We will show that under certain restrictions on the concurrency between the updates, a bounded solution is possible. We also identify a class of declarative specifications called *bounded specifications* which require the replicas to maintain only a bounded fragment of their view to answer any query. Using the bounded solution

of the gossip problem, we obtain finite state reference implementations of CmRDTs whose specifications are bounded specifications. In the next chapter, we will discuss how such finite state reference implementations can be used for formal verification of CmRDT implementations via CEGAR and also help in designing more effective test suites for CmRDT implementations.

In the following section, we will discuss bounded implementations of replicated data types which have a *bounded specification*. In this chapter and the next, we shall restrict our attention to operation based replicated data type implementations where the updates are causally delivered.

5.2 Bounded CmRDTs

Finite state implementations have played an important role in formal verification of reactive systems. In this section we study the sufficient conditions that guarantee the existence of finite state implementations of replicated data types whose specifications have certain properties. In the next chapter, we shall discuss how a finite state reference implementation can be used for the purpose of automated formal verification of a given implementation of a replicated data type. Towards this, we first define a finite state implementation, or a *bounded implementation*. We shall use the terms *finite state implementation* and *bounded implementations* interchangeably throughout this and the next chapter.

Definition 107. *We say that an implementation of a CRDT is bounded if the information maintained by every replica and the contents of each message propagating an update are bounded, regardless of the length of the run.*

Any implementation of a replicated data type with an unbounded universe Univ will have to maintain information pertaining to the elements of the universe. If the universe Univ is unbounded, in general, it is likely that the states of the implementation are also unbounded. Hence, we restrict our attention to implementations of data types where the size of the universe supported is bounded.

Even with a bounded universe, the number of states of the reference representation grows proportional to the length of the run, since the state is the view at the replica which grows proportional to the length of the run. For example, as per the OR-Set specification, an implementation of OR-set needs to keep track of all the $\mathsf{add}(x)$ operations that do not have a covering $\mathsf{delete}(x)$ operation. This is especially true when the underlying network does not guarantee causal delivery of updates. Now, a $\mathsf{delete}(x)$ operation that is a covering-delete for $\mathsf{add}(x)$ operation o may not be a covering delete for any other $\mathsf{add}(x)$ operation that happened-before o. Thus, in any such implementation of OR-set, the number of $\mathsf{add}(x)$ operations that are not covered by any $\mathsf{delete}(x)$ operations is unbounded in an arbitrarily

long run. Hence, even when the universe of the OR-set is bounded, the state of the replicas may be unbounded merely because the specification requires the implementation to maintain information pertaining to unbounded number of events.

However, we observe that for many of the well behaved data types, it is sufficient to keep track of a finite fragment of the view that will help us correctly answer all potential queries. For example in the case of MV-Register, for a view V, the specification function would be $\mathsf{RelevantCtxt}(V, \mathsf{Read}, \bot) = max(V_{\mathsf{Write}})$ where V_{Write} is the subtrace of V restricted to only the Write events. Now, since there there can be at most one maximal Write event per replica, the total number of events in $max(V_{\mathsf{Write}})$ is bounded by the number of replicas. Thus it is sufficient for the replicas to only maintain information corresponding to the maximal writes.

In the case of OR-Sets, as discussed earlier, the relevant context for a $\mathsf{contains}(x)$ query may have unbounded number of $\mathsf{add}(x)$ events which are not covered by any $\mathsf{delete}(x)$ events. But if the updates are causally delivered, then, it is suficient for the replicas to maintain information corresponding to a finite number of add and delete operations. Consider a view V in a trace T of a causally delivered run. Suppose e is an $\mathsf{add}(x)$ event which does not have a covering $\mathsf{delete}(x)$. Suppose e' is some other x-update event and e'' is a $\mathsf{delete}(x)$ event. It cannot be the case that $e \xrightarrow{\mathsf{vis}} e''$ since e does not have a covering delete. Further, if $e \xrightarrow{\mathsf{vis}} e'$, then it cannot be the case that e' is a $\mathsf{delete}(x)$ event, since that would imply that e has a covering-delete. Finally, it cannot be the case that $e \xrightarrow{\mathsf{vis}} e' \xrightarrow{\mathsf{vis}} e''$, since by causal-delivery this would imply $e \xrightarrow{\mathsf{vis}} e''$, which would imply that e has a covering delete. Thus, if there is an x-update event that is a successor of e in the view V, then that successor has to be an $\mathsf{add}(x)$ event. Thus, when updates are causally delivered, we know that there exists an $\mathsf{add}(x)$ event without a covering $\mathsf{delete}(x)$ in a view of a replica V iff there exists an $\mathsf{add}(x)$ in $max(V_x)$ where V_x is the view V restricted to only the x-events. The size of $max(V_x)$ is bounded by the number of replicas.

Thus there exists a class of specifications of replicated data types where it is sufficient for every replica to maintain bounded fragment of its view in order to correctly answer all the queries. We call such specifications *bounded specifications*.

Definition 108 (Bounded Specification)**.** *A computable specification $Spec_{\mathcal{D}}$ is said to be a bounded specification if the associated function that extracts the relevant context always produces a context of bounded size. Formally, $Spec_{\mathcal{D}}$ is* bounded specification *iff*

$$\exists K \in \mathbb{N} : \forall U \in Ctxt^{arb}(\mathcal{D}) : \ \forall q \in \mathsf{Queries} : \ \forall args \in \mathsf{Univ}^* : \ |\mathsf{RelevantCtxt}(U, q, args)| < K$$

Thus, on a network where all the updates are causally delivered, if the size of Univ is bounded by K_{Univ}, then the number of unique queries instances $\mathsf{contains}(x)$ is also bounded by K_{Univ}. For every x, the maximal events in a view restricted to only the x-events is bounded by the number of replicas N. In this case, the number of events required to answer a query at

any point in time is bounded by $N \times K_{\mathsf{Univ}}$. Thus, in a reference implementation, it suffices to keep track of only this finite fragment of the partial order at any replica. Replicas can purge the events from their view that are no longer relevant for answering any of the queries.

In the following subsection, we shall briefly describe our strategy for constructing a bounded reference implementation for replicated data types with bounded specifications. The formal details are covered in Section 5.3.

5.2.1 Strategy for constructing a bounded reference implementation

We can adapt the reference implementation provided in the Chapter 4 where, instead of maintaining an acyclic graph corresponding to the complete view of the replica, we ensure that every replica maintains a graph corresponding to some bounded fragment of the view which contains all relevant events. Since the replicated data type has a bounded specification, the number of relevant events in view of the trace of any run is bounded. Thus, it is possible to define a bounded fragment of the view containing all the relevant events. This bounded fragment is sufficient for the replicas to answer every query correctly.

When a new *update* event occurs at the replica, it will update this local graph to reflect the updated relevant view that incorporates this latest update event. In this process, some of the nodes of the graph, which correspond to prior events that are no longer relevant in the current view, can be purged from its local graph. Following this, the replica will broadcast its latest graph (referred to as the payload of the *update-receive* below) to all the other replicas. The other replicas, on receiving this broadcast via an *update-receive* event, will recompute their local graphs using the received payload. In this process, they can purge any nodes that correspond to events that are no longer relevant. Thus, the amount of information maintained by every replica is bounded, irrespective of the length of the run.

Note that in the trace of any run of a replicated data type, it is possible that two distinct update events at a replica have the same *update method* with the same arguments (for example a replica can have multiple $\mathsf{add}(x)$ requests from the clients). Hence care must be taken to represent them by two distinct nodes in the acyclic graph corresponding to the relevant view. To ensure this, it is reasonable to assume that every node has a unique identifier associated with it. This can help us distinguish between the nodes representing two distinct events. One may imagine that every replica keeps a monotonically increasing counter which gets incremented every time the replica gets a new update event. The replica then uses the current value of the counter to label the node representing this new update event. However, since the value of the counter can grow with the size of the run, such a strategy would violate our requirement of having bounded states at every replica. Hence, the identifiers for the nodes should come from a finite set. Since every replica maintains a bounded superset of relevant events, it should be possible to reuse the identifiers of prior

nodes which have been purged from the local graph. However, reusing the identifiers of a node that has been purged in a safe manner is non-trivial as we explain below.

For a replica r, suppose ℓ is the identifier of a node corresponding to an r-update event which has been purged from the local graph r. In order to safely reuse ℓ for a new r-update event e, we need to ensure the following two things:

1. The node previously labelled with ℓ is currently not present in the local graph of any other replica r'. Otherwise when r' receives the updated local graph of r through the *update-receive* event of e, it will now have no way of distinguishing between the nodes representing the latest r update event and the old r-update event, since both of them are identified by the same label ℓ. This will violate our requirement that distinct events in the local graph are unambiguously represented by distinct nodes labelled with distinct labels.

2. The node that was previously labelled with ℓ is not a part of the payload of broadcast message sent by some replica r''. It may be the case that r'' does not have that node in its local graph at present. However if r'' had sent out a broadcast for an update event previously when the node was a part of its local graph, then the node would have been in the payload of that broadcast. It is possible that some replica r' has not yet received that broadcast. Thus, if r' now receives the broadcast corresponding to the new r-update event e and sometime later it receives the broadcast message from r'', there will be two nodes representing distinct events, that will be labelled with the same label ℓ.

The ability to safely reuse labels in a distributed setting has been dealt with in earlier works such as [Mukund and Sohoni, 1997; Mukund et al., 2003], which implement a bounded timestamping algorithm in distributed systems by solving what is known as the *gossip problem*. In the standard gossip problem, each replica keeps track of the latest event that it is aware of in every other replica in the system. From time to time, they exchange their knowledge of the latest events with other replicas in the system. The recipient replicas then update their own knowledge of the latest events. In [Mukund and Sohoni, 1997] and [Mukund et al., 2003], the view of a replica r restricted to the set of the latest r' events, for every replica r' is termed as the *primary information* of r. The replicas are expected to periodically broadcast their *primary information* to the other replicas in the system. It is shown in [Mukund and Sohoni, 1997; Mukund et al., 2003] that it is possible to locally recompute the *primary information* of a replica whenever it receives an *update* or an *update-receive* event.

Since there can be at most one *latest event* for every replica r' known to r, it is evident that as the run progresses, earlier events get dropped from the primary information of r. Thus, it is possible to reuse the labels of those events which are no longer in the primary information. To do this safely, the authors define a *secondary information* at every replica.

An event e'' is in the secondary information of a replica r, iff there exists an r' event e' in the primary information of r such that e'' is present in the primary information of r' at e'. Under certain restrictions over the delivery of messages, the authors show that if an event is not in the secondary information of a replica, then that event is not in the primary information of any other replica in the system. Thus it is safe to reuse the label of such an event that is no longer in the secondary information.

Thus, for our bounded reference implementation of a replicated data type with bounded specification, we want to explore a similar mechanism which allows us to safely reuse the labels of the purged nodes. However, the key difference between our requirement and that in [Mukund and Sohoni, 1997; Mukund et al., 2003] is that we want to maintain a bounded set of events as our primary information as opposed to only the latest events of every other replica. Furthermore, unlike the replicas in [Mukund and Sohoni, 1997] which communicated via pairwise synchronization or the replicas in [Mukund et al., 2003] which performed a point to point broadcast of their primary information at the time of their choosing, in our model of operation-based replicated data types, the replicas can only initiate a broadcast to all the replicas when they receive a new update event. Thus, the techniques from [Mukund and Sohoni, 1997] and [Mukund et al., 2003] cannot be directly applied in our model.

To address this, in Section 5.3, we define a generalized version of the gossip problem. Here we generalize the notion of the primary information to any bounded fragment of the view, that is locally computable. We also define the sufficient conditions for a generalized secondary information such that if an event is not present in the generalized secondary information of a replica, then it is present in neither the generalized primary information of any replica nor is it part of a payload of any broadcast message in transit. We show that if such a generalized secondary information exists for a primary information, then the generalized gossip problem has a bounded solution, where every replica maintains a finite amount of information. In Section 5.3.1 we show that such a generalized secondary information can be constructed when certain constraints are imposed over the delivery of messages. Finally in Section 5.3.2 we show how the bounded solution to the generalized gossip problem can be used to construct a bounded reference implementation of a replicated data type with a bounded specification. We illustrate this using the example of OR-Sets where the updates are causally-delivered.

5.3 Generalized Gossip Problem

Consider a distributed system with replicas $\mathcal{R} = [1, \ldots, N]$. Whenever a replica interacts with a client, it does some local processing and broadcasts a message to all the other replicas. These broadcast messages are delivered in the causal order of the updates. This is similar to the behaviour of *operation based* replicated data types described earlier. Suppose now that every replica keeps track of the latest event it knows about every other replica in the

system. During a broadcast, along with the message, each replica r also sends across its knowledge about the latest event of every other replica r' in the system. A recipient replica r'' needs to correctly compute whether the latest event that it knows about some replica r' is more up to date than the latest event about r' in the message that it has just received. Having determined which is the more up-to-date information, the replica r'' should update its knowledge. This problem is known as the *gossip problem*, and has been studied in [Mukund and Sohoni, 1997; Mukund et al., 2003].

We propose the following generalization. Instead of maintaining only the latest events, suppose the replicas were to maintain some finite fragment of their past as defined by a computable function. In the case of CRDTs with bounded specifications, this finite fragment could be the fragment of the view containing the union of the relevant contexts for all the queries. In such a case, the information maintained by the replicas will not be merely a set of events, but a graph containing the set of events along with the visibility relation over them. We model this as an *information graph*.

Definition 109 (Information graphs)**.** *An information graph G of a trace $T = (\mathcal{E}, \mathsf{vis})$ is a subtrace $T|_E$ induced by the subset of update events $E \subseteq_{\text{fin}} \mathcal{E}_{\mathsf{Updates}}$. We denote the set of all information graphs of T by $\mathcal{G}(T)$. Let $\mathcal{G}_{\mathcal{D}} = \bigcup_{T \in \mathcal{T}(\mathcal{D})} \mathcal{G}(T)$.*

We shall denote the set of all views of a trace T, by $\mathcal{V}(T) = \bigcup_{e \in Events(T)} \{\partial_e(T)\}$ and the set of all views of a replicated data type $\mathcal{D}$ to be $\mathcal{V}(\mathcal{D}) = \bigcup_{T \in \mathcal{T}(\mathcal{D})} \mathcal{V}(T)$

In a trace T, if $V = \partial_T(e)$ for some event e. Then, for a replica r, $max_r(V)$ denotes the maximal r-update event in V, i.e,.

$$max_r(V) = e' \iff Rep(e') = r \wedge Op(e) \in \mathsf{Updates} \wedge \forall e'' \in V : e' \xrightarrow{\mathsf{vis}} e'' \implies Rep(e'') \neq r$$

We shall denote by V_r the view of $max_r(V)$, i.e $V_r = \partial_T(max_r(V))$. Thus V_r is the view of the maximal r-event in the view V. This includes all the update events that are visible to the r-update event $max_r(V)$. Since T is a trace of a causally delivered run it follows that $\forall r : V_r \subseteq V$.

We say that a view $V = \partial_T(e)$ is an r-view, if $Rep(e) = r$.

In any trace $T = (\mathcal{E}, \mathsf{vis})$, and any event e, we denote by $Pred_T(e)$ the set of events which form the immediate predecessors of e in T, i.e,

$$Pred_T(e) = \{e' \in \mathcal{E} \mid e' \xrightarrow{\mathsf{vis}} e \wedge e' \neq e \wedge \forall e'' \in \mathcal{E} : e'' \xrightarrow{\mathsf{vis}} e \implies e'' \xrightarrow{\mathsf{vis}} e' \vee e'' || e\}$$

Note that for an *update* event, a *query* event and a *update-send* event e, there is a single immediate predecessor, which is the latest event occuring prior to e at the source replica of e. However if e is an *update-receive* event, then it will have two immediate predecessors.

One of them will be the immediate predecessor on the same replica, and the other will be the *update* event associated with e.

We denote the r-event in $Pred_T(e)$, when it exists by $Pred^r_T(e)$. Note that if $r = Rep(e)$, then $Pred^r_T(e)$ always exists. This is because the event e_0 corresponding to the initialization operation $\rho[0]$ in the run $\alpha = (\rho, \varphi)$ of the trace T is an operation that is common to all the replicas.

Observe that in the case of the *standard gossip problem*, each replica maintained the latest events corresponding to every other replica that was delivered to them. This set of events is defined by a computable function which can compute the latest r' events for every other replica r' in the system. Similarly in the case of the generalized gossip problem, it is desirable that the finite fragment of the view that is maintained by every replica is determined by a computable function which can be uniformly applied at the view of every replica. This computable function is referred to as the *primary information function* and the information graph computed by this function is termed as the *primary information* at the replica.

Definition 110 (Primary information function and Primary Information)**.** *A* primary information function *is a function $f : \mathcal{V}(\mathcal{D}) \to \mathcal{G}_{\mathcal{D}}$ that assigns an information graph to each view so that the following conditions are satisfied: For any trace T and any event $e \in \mathcal{E}(T)$,*

1. *$f(\partial_T(e)) \in \mathcal{G}(\partial_T(e))$, i.e the primary information is a finite fragement of the view containing only update events.*
2. *If $Op(e) \in \mathsf{Queries} \cup \{\mathbf{usend}\}$, $f(\partial_T(e)) = f(\partial_T(e'))$ where $Rep(e) = r \wedge e' = Pred^r_T(e)$. Thus the primary information of a replica does not change with an query or an update-send event.*
3. *If e is an update event then, $e \in f(\partial_T(e))$. Thus an update event at a replica is always contained in the primary information of the replica, soon after the update event occurs.*
4. *For any $e' \in maxSet(\partial_T(e) \setminus \{e\})$, $f(\partial_T(e)) \cap \partial_T(e') \subseteq f(\partial_T(e'))$.*

The first condition ensures that the primary information in any view is a finite fragment of the view comprising of only the update events.

The second condition ensures that the primary information of a view of a replica doesn't change with query or send-events.

The third condition ensures that an update event is always in the primary information of the view of that update event.

The fourth condition says that if an update event is not present in the primary information of any event e', then that update event will not be present in any subsequent event which e to which e' is visible. This imposes a sort of a monotonicity property that things once excluded from the primary information of some view remain excluded in the primary information of any larger view.

The information graph computed by the primary information function f in any view V is deemed to be the primary information *at that view. A primary information function f is said to be* bounded *if* $\exists M\ \forall V \in \mathcal{V}(\mathcal{D}) : |Events(f(V))| \leq M$.

Every replica maintains the primary information at its latest view. After every update event, the replica will broadcast its primary information to every other replica. Thus, when a replica receives the primary information from some other replica, it needs to recompute its primary information from the primary information that it just received as a part of the *update-receive* and the primary information that it had prior to the *update-receive* event. Thus, for any event e in any trace T, the *generalized gossip problem* for a primary information function f involves computing the primary information in the view of e, i.e $f(\partial_T(e))$, from the primary information of the views of the immediate predecessors of e, i.e $f(\partial_T(e_{prev}))$ and e, where $e_{prev} \in Pred_T(e)$. We cay that the *generalized gossip problem* for f has a solution, iff the replicas can compute their primary information locally.

Suppose we have a bounded primary information function $f()$ for which the solution for the generalized gossip problem exists. Thus every replica maintains only the finite fragment of its view as defined by $f()$. Note that the replicas send each other their primary information graph on every update event. Each vertex in this graph corresponds to a unique update event in the trace. Since the solution to the generalized gossip problem exists, the replicas can locally recompute their primary information on an *update* or an *update-receive* event. In order to do that, the replicas need to unambiguously represent the update events in the primary information graph, so that any pair of distinct update events in the primary information at any replica can be distinguished whenever a replica receives the primary information from some other replica.

We observe that during every update, a new event gets added to the primary information of the source replica corresponding to the update. Suppose the source replica labels every event originating at its end with a unique identifier from a set of labels $\mathcal{L}$, such that for any two distinct events e, e' in the trace are labelled by distinct labels $\mathcal{L}(e)$, $\mathcal{L}(e')$ respectively such that $\mathcal{L}(e) \neq \mathcal{L}(e')$. As the computation grows, and more new events are added we need newer labels from $\mathcal{L}$ to uniquely identify these new events. Thus, the size of $\mathcal{L}$ in general is unbounded.

However, if we are only interested in keeping track of the primary information generated by a bounded primary information function at any given point in time, then the total number of relevant events in any view of the trace is bounded. Thus, once an event is no longer in the primary information of any maximal event in the trace, the label assigned to that event could be reused. We say that the gossip problem for f has a bounded solution if it has a solution and if the set of labels used to uniquely identify the events in the system is bounded.

In a bounded solution, the replicas will be forced to reuse labels for newer update events. However, in order to distinguish between distinct events, the replicas must take care to reuse

label of an update event only after ensuring that the update event is not in the primary information of any replica, nor is the event part of any message in transit. Furthermore, the replicas should be able to arrive at this decision locally based on the information that it locally possesses, and any information that it receives as a part of the *update-receive* message. Towards this, replicas may maintain some additional information locally, which will help them determine if a given event is in the primary information of some other replica or is a part of the transit message. This additional information is termed as the *secondary information* in the definition below. The replicas communicate with each other not only the primary information, but also the secondary information.

Definition 111 (Secondary information)**.** *Let e be an event in a trace T of a replicated data type $\mathcal{D}$. Let V denote the view of the event e in T, i.e $\partial_T(e)$. Recall that for a replica r, the view of the maximal r-update event in T is denoted V_r i.e., $V_r = \partial_T(max_r(V))$ where $max_r(V)$ is the maximal r-update event in V.*

A function $F : \mathcal{R} \times \mathcal{V}(\mathcal{D}) \to 2^{(\mathcal{E}_{\mathcal{D}}|_{\mathsf{Updates}})}$ is a secondary information function *for a primary information function f if, in any trace T, for each any event e, if $V = \partial_T(e)$, then*

1. *$Events(f(V_r)) \cap \mathcal{E}^r_{\mathsf{Updates}} \subseteq F(r, V) \subseteq Events(V) \cap \mathcal{E}^r$.*
2. *If e is not an r-event, $F(r, V) = F(r, V_r)$*
3. *If e is an r-event, $F(r, V)$ is computable from e, $f(V)$ and $F(r, \partial_T(e'))$ where $e' = Pred^r_T(e)$.*
4. *If e and e' be r-events such that $e' \in V \setminus F(r, V)$, then for any event $e'' \in T$, $e \in \partial_T(e'') \implies e' \notin f(\partial_T(e''))$*
5. *For an r-event $e' \in V \setminus F(r, V)$, if $e' \in f(\partial_T(e''))$, then $\varphi^{-1}_{r''}(e'') \xrightarrow{\mathsf{vis}} \varphi^{-1}_{r''}(max_r(V))$ for all r''.*

A secondary information function F is said to be bounded *if $\exists M\ \forall r\ \forall V : |F(r, V)| \leq M$.*

We shall refer to $F(r, V)$ as the r-secondary information in the view V.

The first condition says that the r-secondary information in any view V is a subset of the r-update events in the trace. Furthermore, the r-secondary information at V contains all the r-update events in the primary information of V_r. The second condition says that r-secondary information of any non r-view V is exactly the r-secondary information at the view of the maximal r event in V (which is V_r). The third condition says that the r-secondary information in an r-view can be computed locally using the r-secondary information of the previously maximal events in the view and the primary information of the view. These three conditions ensure that the secondary information in the view for every replica can be computed locally. The fourth condition says that once an r-update event e' goes out

of the r-secondary information in an r-view V, then, the event e' is not in the primary information of any subsequent view V' where $V \subseteq V'$. The last condition, is a bit subtle. It says that when an r-update event e' goes out of the r-secondary information in an r-view corresponding to an r-update event e, and suppose there is an any update event e'' that had e' in its primary information. Then, it says that at every replica, the broadcast message associated with e'' gets delivered before the broadcast message associated with e. This condition ensures that when an event goes out of the secondary information, it is not a part of any primary information that has been broadcast but not been delivered. The fourth and the fifth conditions, allow us to safely reuse the label of an r-event e' when it goes out of the r-secondary information of an r-view, since in any subsequent view which subsumes this r-view, e' is neither in the primary information, nor is it a part of the primary information in transit.

To summarize, the secondary information should have three properties

- The secondary information must be locally computable.
- If an r-update event is not in the secondary information of another r-update event, then, it is guaranteed that the former event is evicted from the primary information of any replica r' before it receives the latter.
- If an r-event is not in the secondary information of another r-event, then it is guaranteed any *update-receive* messages which contain the former r-event would be delivered at every replica before the latter r-event gets delivered, thereby ensuring that once the r-event is not in the secondary information, it is not a part of any message in transit.

It is clear from the definition that, as the run grows, the older update events, which are no longer in the primary information of any replica and which are no longer a part of any broadcast messages in transit, drop out of the secondary information. Note that for the purpose of arriving at a bounded solution for the *generalized gossip problem*, the secondary information maintained by every replica should also be bounded. We now show that the generalized gossip problem for a primary information function has a bounded solution if it has a bounded secondary information function.

Theorem 112 (Bounded solution for the *generalized gossip problem*)**.** *Let f be a primary information function such that the generalized gossip problem for f has a solution. It has a bounded solution if there is a bounded secondary information function F for f.*

Proof. In order to show that f has a bounded solution for the generalized gossip problem, we need to show that the size of set of labels used to uniquely identify the events in the primary information of all the replicas is bounded.

Let the gossip problem for f have a solution, and let the bound on F be M'. We fix a label set $\mathcal{L}$ of size $M'+1$. We shall label an r-update event with (r, ℓ) where $\ell \in \mathcal{L}$. We need

to show that we can safely associate a label ℓ from the set $\mathcal{L}$ to a new r-update event e_{in} such that if any other r-update event e_{out} is labelled with the same label ℓ, at every replica r', prior to receiving the broadcast message containing e_{in}, it is the case that

- e_{out} is no longer in the primary information of r'.
- r' has received all the broadcast messages which may contain e_{out} in them.

Let V be the view of replica r just prior to the r-update event e_{in}. Since the secondary information $F(r, V)$ is bounded by M' and the label set $\mathcal{L}$ has $M'+1$ events, there is at least one label in $\mathcal{L}$ which is not associated with any r-update event in $F(r, V)$. Suppose one such label is ℓ. We will show that it is safe to associate the label ℓ with the new r-update event e_{in}.

If ℓ has never been associated with any r-update event previously in the trace, then our claim is true. So, let us assume that an r-update event e_{out} was previously associated with ℓ. It is clear that $e_{out} \notin F(r, V)$, since we chose the label ℓ based on the fact that it was not labelling any event in $F(r, V)$. By the property of the secondary information, if an r-event is not in the secondary information of r at any point in the run, then, it is not in the primary information either of r at that point. Thus $e_{out} \notin f(V)$. By monotonicity of the primary and the secondary information, e_{out} will never be in the primary or secondary information any future event whose view contains the view V.

Let e be the *earliest* r-update event such that e_{out} is not in the secondary information the view of r at e. Thus $e_{out} \notin F(r, \partial_T(e))$. Thus we have that e_{out} is not in the primary information of r at e, i.e $e_{out} \notin f(\partial_T(e))$.

Let e' be the *update-receive* event of e at some replica r'. Thus $e' = \varphi_{r'}^{-1}(e)$. By definition $e \xrightarrow{\mathsf{vis}} e'$. By the property of secondary information, since $e_{out} \notin F(r, \partial_T(e))$, $e_{out} \notin f(\partial_T(e'))$. Thus, e_{out} is not in the primary information of the replica r' at event e'. Since we only consider traces where updates are causally delivered, if e'_{in} is the *update-receive* event of the new event e_{in} at replica r', then since $e \xrightarrow{\mathsf{vis}} e_{in}$, we have $e' \xrightarrow{\mathsf{vis}} e'_{in}$. Thus at r', prior to receiving the broadcast of e_{in}, the event e_{out} is no longer in the primary information.

Furthermore, suppose e_{other} be some event such that e_{out} is in the primary information its source replica at the event e_{other} in T. Thus $e_{out} \in f(\partial_T(e_{other}))$. Let e'_{other} be the *update-receive* event of e_{other} at replica r'. By the definition of secondary information, since $e_{out} \notin F(r, \partial_T(e)$ and $e_{out} \in f(\partial_T(e_{other})$ at the replica r', it is the case that e'_{other} gets delivered before e'. Thus $e'_{other} \xrightarrow{\mathsf{vis}} e'$. By causal-delivery, we have $e' \xrightarrow{\mathsf{vis}} e'_{in}$. Thus, it follows that $e'_{other} \xrightarrow{\mathsf{vis}} e'_{in}$. Since e_{other} is any event which contained e_{out} in its primary information, it is guaranteed that the broadcast messages of e_{other} are delivered at every replica prior to the broadcast messages of the new event e_{in}.

Thus, at the time of the receipt of the new event e_{in} it is guaranteed that not only has every other replica purged the old event e_{out} from its primary information, it cannot expect

e_{out} to be a part of any other broadcast message that it might receive in the future. Hence, the label ℓ associated with identify e_{in} unambiguously identifies the new event.

This shows that a bounded secondary information function implies a bounded labelling solution for the gossip problem for the primary information function f which has a solution. □

In the following section we shall show a sufficient condition for the existence of a bounded secondary information function for any primary information function that has a solution for the generalized gossip problem.

5.3.1 Constructing a Bounded Secondary Information

In the gossip problem considered in [Mukund and Sohoni, 1997; Mukund et al., 2003], the primary information f at V is the set $\{max_r(V) \mid r \in \mathcal{R}\}$ along with the secondary information $\{max_{r'}(V_r) \mid r, r' \in \mathcal{R}\}$ which ensures that the gossip problem for f is solvable. A bounded solution for f is provided in [Mukund et al., 2003] when f itself is bounded, but under additional restrictions on the traces of the system. A notion of acknowledgements is introduced, and all traces of the system are required to have at most B unacknowledged messages. An acknolwedgement, as defined in [Mukund et al., 2003] is a message sent by the recepient of the broadcast message to the sender of the broadcast message acknowledging the receipt of the broadcast. This helped the sender keep track of how many messages are potentially undelivered (Note : If a replica r does not receive an acknowledgement from another replica r' for a broadcast message sent by r, it does not necessarily mean that the broadcast message hasn't been delivered at r'. It could just be that the acknowledgement from r' hasn't reached r yet).

In [Mukund et al., 2003], replicas piggyback acknowledgements to previously received messages in their subsequent broadcasts. Hence, the bound on unacknowledged messages requires that replicas communicate with each other at regular intervals. Such a solution would not work in our model for operation based replicated data types, because replicas broadcast messages only when they receive an update request from a client. Hence, we need a different guarantee from the underlying messaging system. We identify one such restriction below

Definition 113 (B-concurrency)**.** *A trace T is B-concurrent if for every update $e \in T$, the number of other update events that are concurrent with e are bounded by B. Thus*

$$\forall e \in Events(T)|_{\mathsf{Updates}} : |\{e' \in Events(T)|_{\mathsf{Updates}} \mid e' \parallel e\}| \leq B$$

.

An implementation is B-concurrent if all its traces all its runs are B-concurrent.

In this section, we will show how to construct a secondary information function for a bounded primary information function f to which a solution to its generalized gossip exists in the case of B-concurrent traces.

Since the primary information function $f()$ is bounded, as the length of the computation grows, update events keep dropping out of the primary information at every replica. For every r-update event e that goes out of the primary information of the replica r, we try to identify the earliest r event e' which did not have e its primary information. We define this e' to be the *evict-cause* of e.

Definition 114 (Evict-Cause and Evict-order)**.** *In any trace T, we define the Evict-Cause of an r-update event e with respect to a primary information function $f()$, denoted by $\mathsf{EvictCause}_{T,f}(e)$ to be the earliest r-event e' that comes after e, such that $e \notin f(\partial_T(e'))$. If no such e' exists, then we set $\mathsf{EvictCause}_{T,f}(e) = \bot$.*

For any pair of r-update events e and e' whose evict-causes *are valid r-update events, i.e $\mathsf{EvictCause}_{T,f}(e) \neq \bot$ and $\mathsf{EvictCause}_{T,f}(e') \neq \bot$, we define the* evict-order *between the r-update events e and e', denoted by $\prec^r_{T,f}$, as follows $e \prec^r_{T,f} e'$ iff*

- *Either* $\mathsf{EvictCause}_{T,f}(e) \neq \mathsf{EvictCause}_{T,f}(e')$ *and* $\mathsf{EvictCause}_{T,f}(e) \xrightarrow{r} \mathsf{EvictCause}_{T,f}(e')$ *or*
- $\mathsf{EvictCause}_{T,f}(e) = \mathsf{EvictCause}_{T,f}(e')$ *and* $e \xrightarrow{r} e'$

Thus, in any view V of a trace T, we define the set of evicted r-events in V, denoted by $\mathsf{EvictedEvents}^r_{V,f}$, to be the r-events in the view V_r which are not in the primary information at V_r.

$$\mathsf{EvictedEvents}^r_{V,f} = \{e \in V_r \setminus f(V_r) \ \mid \ \mathit{Rep}(e) = r\}$$

Note that the *evict-order* $\prec^r_{T,f}$ restricted to the set of evicted events $\mathsf{EvictedEvents}^r_{V,f}$ is well defined and it imposes a total order on the r-events in . We shall denote this evict-order restricted to the evicted events $\mathsf{EvictedEvents}^r_{V,f}$ by $\prec^r_{V,f}$. Thus, $\prec^r_{V,f}$ imposes a total order on the r-events in $\mathsf{EvictedEvents}^r_{V,f}$.

The position of an evicted r-event e in this total order is defined to be its *evict-rank*, denoted by $\mathsf{EvictRank}^r_{V,f}(e)$, formally defined as

$$\mathsf{EvictRank}^r_{V,f}(e) = |\{e' \in \mathsf{EvictedEvents}^r_{V,f} \ \mid \ e \prec^r_{V,f} e'\}|$$

The latest n-prefix of $\mathsf{EvictedEvents}^r_{V,f}$ consisting of n events is defined to be the set of evicted events of rank atmost n. This is denoted by $\mathsf{RecentEvictedEvents}^r_{V,f}(n)$.

$$\mathsf{RecentEvictedEvents}^r_{V,f}(n) = \{e \in \mathsf{EvictedEvents}^r_{V,f} \ \mid \ \mathsf{EvictRank}^r_{V,f}(e) \leq n\}$$

Given a B-concurrent trace T and a bounded primary information function $f()$ with a bound M, we can say whether or not an r-update event e in the view V is a *recent event* whether e is present in the primary information of V or if e is in the recent $M + B$ evicted events of V. The intuition here is that as long as e event is recent, it may be present in the primary information of some replica.

Definition 115 (Recent Event in B-concurrent traces)**.** *Let f be a bounded primary information function with a bound M, the solution to whose generalized gossip problem exists.*

We say that an r-update event e is recent with respect to f in a view V of a B-concurrent trace T iff

1. *Either $e \in f(V_r)$ or*
2. *$e \in \mathsf{RecentEvictedEvents}^r_{V,f}(M + B)$*

We denote the set of all such r-recent events with respect to f in V by $\mathsf{Recent}^r_{f,B}(V)$

Note that with a bounded primary information function f with a bound M, then, in any view V of any B-Concurrent trace, $|\mathsf{Recent}^r_{f,B}(V)| \leq 2M + B$.

Definition 116 (Secondary information function for B-concurrent traces)**.** *Let T be a trace and V be any view in T. For any replica $r \in \mathcal{R}$, we define $F_{(f,B)}(r, V) = (\mathsf{Recent}^r_{f,B}(V), \prec^r_{V,f})$*

Note: Technically, the r-secondary information $F_{(f,B)}(r, V)$ is set of r-update events. Hence should just be $\mathsf{Recent}^r_{f,B}(V)$. Here, in addition to this set of events, we are also maintaining a total order $\prec^r_{V,f}$ on the update events in this set. This order can be maintained by the replicas separately. However for the ease of presentation, we tag the evict order alongside $\mathsf{Recent}^r_{f,B}(V)$, though it is not required by the definition of the secondary information.

We now show that the $F_{(f,B)}$ is locally computable.

Proposition 117. *For any event e in a B-concurrent trace T, for any replica r, $F_{(f,B)}(r, \partial_T(e))$ is computable from $f(\partial_T(e))$, e and $F_{(f,B)}(r, \partial_T(e'))$ where $e' \in Pred_T(e)$.*

Proof. We shall prove this by induction on the height of the *event* e in T.

For an empty trace, the result trivially holds.

Suppose the result holds for all events e of height smaller than n. Now consider an event e, of height n. Let $Rep(e) = r$. Let $e' = Pred^r_T(e)$. Let $V' = \partial_T(e')$.

e **is a query or an usend event:** Then, by definition, the primary information at e is the same as the primary information at e'. Since $V = V'$, no new update event has entered the view of e. Thus, by definition, $\mathsf{Recent}^{r'}_{f,B}(V) = \mathsf{Recent}^{r'}_{f,B}(V')$ for all r'.

Hence $F_{(f,B)}(r', V) = F_{(f,B)}(r', V')$ for all r'.

e **is an update event:** Since e is an r-update event, for any distinct replica $r' \neq r$, the maximum r' event in V is same as the maximum r' event in V, i.e.

$$max_{r'}(V) = max_{r'}(V')$$

Thus the views of the maximal r' events in V and V' are the same.

$$\partial_T(max_{r'}(V)) = \partial_T(max_{r'}(V'))$$

Hence, the primary information of the maximal r' event in V and V' is the same.

$$f(\partial_T(max_{r'}(V))) = f(\partial_T(max_{r'}(V')))$$

Also, by definition, the evicted r' events in $max_{r'}(V)$ is the same as the evicted r' events in $max_{r'}(V')$, and hence the evict-order over those events is the same. Thus

$$\mathsf{EvictedEvents}^{r'}_{V,f} = \mathsf{EvictedEvents}^{r'}_{V',f} \text{ and } \prec^{r'}_{V,f} = \prec^{r'}_{V',f}$$

Thus, by definition, $\mathsf{Recent}^{r'}_{f,B}(V) = \mathsf{Recent}^{r'}_{f,B}(V')$.

Hence for a replica r' distinct from r $F_{(f,B)}(r', V) = F_{(f,B)}(r', V')$.

We will now show how $F_{(f,B)}(r, V)$ can be locally computed.

Now, by definition, $f(V)$ is computable from $f(V')$ and e.

Let us denote the set of r-update events in the trace T by $Events(T)^r_{\mathsf{Updates}}$. Let us denote the recent r-events in V' which are still in the primary information of V' as E'_{in}. Thus,

$$E'_{in} = f(V') \cap Events(T)^r_{\mathsf{Updates}}$$

Let E'_{out} denote remaining r-update events in $\mathsf{Recent}^r_{f,B}(V')$.

$$E'_{out} = \mathsf{Recent}^r_{f,B}(V') \setminus E'_{in}$$

.

Similarly, let E_{in} denote the r-update events which are in the primary information at V.

$$E_{in} = f(V) \cap Events(T)^r_{\mathsf{Updates}}$$

and let E_{out} denote the r-update events which were evicted in V' along with the r-update events which recently got evicted in V. Thus

$$E_{out} = \mathsf{Recent}^r_{f,B}(V') \setminus E_{in}$$

Note since more r-update events could have been evicted from the primary information of V compared to the primary information of V', we have

$$E'_{out} \subseteq E_{out}$$

.

Furthermore, from the way we have defined these sets,

$$E_{in} \cup E_{out} = E'_{in} \cup E'_{out} \cup \{e\}$$

.

Thus, for any pair of events $e'', e''' \in E'_{out}$, since these were already evicted in V', we know their evict order from $\prec^r_{V',f}$ in $F_{(f,B)}(r, V')$. Their evict-order remains the same in V. Hence, for such events, we set the evict order

$$\prec^r_{V,f} |_{E'_{out}} = \prec^r_{V',f}$$

Suppose e'' was newly evicted in V, i.e. $e'' \in E_{out} \setminus E'_{out}$ and e''' was evicted in V', i.e. $e''' \in E'_{out}$, then, we set $e''' \prec^r_{V,f} e''$, since e'' got evicted recently.

Finally, if both e'', e''' were evicted recently in V, i.e $e'', e''' \in E_{out} \setminus E'_{out}$, then we set the evict-order between these two events as the replica order. This replica order is available from the primary information of V', i.e $f(V')$. Thus in this case,

$$e''' \prec^r_{V,f} e'' \text{ iff } e''' \xrightarrow{r} e''$$

Thus, we now have a total order $\prec^r_{V,f}$ on E_{out}. From E_{out}, we pick the latest $M + B$ r-events ordered by $\prec^r_{V,f}$. Let us denote this set by E^{M+B}_{out}.

Then, by definition $\mathsf{Recent}^r_{f,B}(\partial_T(e)) = E_{in} \cup E^{M+B}_{out}$ and the $\prec^r_{V,f}$ is a total order over E^{M+B}_{out}.

Thus, in this case, $F_{(f,B)}(r, V)$ can be locally computed.

e **is an ureceive event:** Let $e'' = \varphi(e)$. Let $Rep(e'') = r''$. Let $V'' = \partial_T(e'')$.

Note that $f(V)$ is computable from $f(V')$ and $f(V'')$. Some additional r-update events may have been evicted from the primary information of V compared to the primary information of V'. Thus, we can compute $F_{(f,B)}(r, V)$ from $F_{(f,B)}(r, V')$ and $f(V)$ as we did in the case when e was an update event. Thus for r, the function $F_{(f,B)}(r, V)$ can be locally computed.

Since the maximal r''-update event in V is $max_{r''}(V) = e''$, by definition, $\mathsf{Recent}^{r''}_{f,B}(V) = \mathsf{Recent}^{r''}_{f,B}(V'')$. Hence, $F_{(f,B)}(r'', V) = F_{(f,B)}(r'', V'')$.

Thus for r'', the function $F_{(f,B)}(r'', V)$ can be locally computed.

For any other replica $r' \notin \{r, r''\}$, let the latest r' update event the predecessor view V' be $e_{V,r'}$ and let the latest r'-update event in V'' be $e_{V'',r'}$. Since both these events are in the view V, we set

$$F_{(f,B)}(r', V) = \begin{cases} F_{(f,B)}(r', V') \text{ if } e_{V',r'} \text{ is maximal } r'\text{-update event in } V \\ F_{(f,B)}(r', V'') \text{ if } e_{V'',r'} \text{ is maximal } r'\text{-update event in } V \end{cases}$$

We now show that the maximal r'-update event in V is always $e_{V',r'}$.

Since $e_{V'',r'}$ would be in the view of V'', it is clear that

$$e_{V'',r'} \xrightarrow{\mathsf{vis}} e''$$

Furthermore, since e'' is the update event associated with the update-receive event e,

$$e'' \xrightarrow{\mathsf{vis}} e$$

Thus, by causal delivery of updates, it is clear that

$$e_{V'',r'} \xrightarrow{\mathsf{vis}} e$$

.

Now, since the replicas communicate only by broadcasting information on an update-message, the only way the $e_{V'',r'}$ could have entered the view of the replica r, is via the corresponding *update-receive* event at r. But then, such an update receive event would be in the predecessor view V'. Thus $e_{V'',r'} \in V'$.

Since we have mentioned that the maximal r' event in V' is $e_{V',r'}$, we have

$$e_{V'',r'} \xrightarrow{\mathsf{vis}} e_{V',r'}$$

.

Thus, in the view V, the maximal r' event is the maximal r' event in the predecessor view V'. From this, it follows that $F_{(f,B)}(r', V)$ is the same as $F_{(f,B)}(r', V')$. Thus, even in this case, for all the replicas r', we can compute $F_{(f,B)}(r', V)$ locally.

Thus, $F_{(f,B)}(r', V)$ is locally computable. □

We next show that between the time when an r-update event is in the primary information of the replica r to the time when that event ceases to be a recent event at the replica r, there would have been more than B update events at r.

Lemma 118. *Let T be a B-concurrent trace. Let e_{out} be an r-update event in T. Let e_{prev} be the latest r-event such that e_{out} is in the primary information at e_{prev}. Let e be the earliest r-event such that e_{out} is not a recent event at e, i.e $e_{out} \notin \mathsf{Recent}^r_{f,B}(\partial_T(e))$. Then, there are more than B update events between e_{prev} and e.*

Proof. Let us denote $V = \partial_T(e)$, and $V_{prev} = \partial_T(e_{prev})$.

Since $e_{out} \notin \mathsf{Recent}^r_{f,B}(V)$, it is clear that $e_{out} \notin f(V_r)$ and $e_{out} \not/ \mathsf{RecentEvictedEvents}^r_{V,f}(M+B)$. Hence, there have been more than $M + B$ r-update events that have been evicted after e_{out}. Thus we have

$$\mathsf{EvictRank}^r_{V,f}(e) > M + B$$

Note that since $f()$ is a bounded primary information function with a bound M, the number of events in $f(V_{prev})$ is bounded by M. This implies that the number of r-update events in $f(V_{prev})$ is bounded by M.

Furthermore, if e' is any r-event between e_{prev} and e, then, if e' is a query or an update-receive event, it does not introduce a new r-update event into the primary information of r. If e' is an update event, then it introduces exactly one new r-event into the primary information of r, which is e' itself.

Thus, if there are k update events between e_{prev} and e, the number of distinct r-update events that have been in the primary information of the replica r between e_{prev} to e (including e_{prev}) is bounded by $M + k$.

If e_{cause} is the immediate r-successor of e_{prev}. Let us denote by V_{cause} the view $\partial_T(e_{cause})$. Then since e_{prev} is the latest r-event containing e_{out} in its primary information, we have

$$e_{out} \notin f(V_{cause})$$

Since the number of r-update events in $f(V_{prev})$ is bounded by M, and since in the extreme case, all of them can be evicted out from $f(V_{cause})$, we have

$$\mathsf{EvictRank}^r_{V_{cause},f}(e_{out}) \leq M$$

.

Since e_{prev} and e , there can be at most k new r-update events that can be introduced into the primary information of r, and since in the extreme case, all these events can be evicted at e, we have

$$\mathsf{EvictRank}^r_{V,f}(e_{out}) \leq M + k$$

.

Thus, we have

$$M + k \geq \mathsf{EvictRank}^r_{V,f}(e_{out}) > M + B \implies k > B$$

which completes the proof. □

We now show that any update event containing an r-update event in its primary information will be delivered at replica r before that r-update event ceases to be a recent event at r.

Lemma 119. *Let e_{out} be an r-update event in a B-concurrent trace T. Let e_{other} be some update event such that $e_{out} \in f(\partial_T(e_{other}))$. Let e be an r event such that $e_{out} \notin \mathsf{Recent}^r_{f,B}(\partial_T(e))$. Then, $e_{other} \xrightarrow{\mathsf{vis}} e$.*

Proof. We prove this by contradiction. Suppose $e_{other} \not\xrightarrow{\mathsf{vis}} e$.

Let e_{prev} be the latest r-event such that $e_{out} \in f(\partial_T(e_{prev}))$. Then, from lemma 118 we know that there are atleast $B+1$ r-update events between e_{prev} and e.

Let e_i be one such r-update. By definition,

$$e_{out} \notin f(\partial_T(e_i))$$

Now if $e_i \xrightarrow{\mathsf{vis}} e_{other}$, since $e_{out} \notin f(\partial_T(e_i))$ from the Property 4 of primary information, would imply that $e_{out} \notin f(\partial_T(e_{other}))$ which is a contradiction. Hence,

$$e_i \not\xrightarrow{\mathsf{vis}} e_{other}$$

it cannot be the case that $e_i \xrightarrow{\mathsf{vis}} e_{other}$ since that would mean that $e_{out} \notin f(\partial_T(e_{other}))$.

It cannot be the case that $e_{other} \xrightarrow{\mathsf{vis}} e_i$ since by causal delivery that would imply that $e_{other} \xrightarrow{\mathsf{vis}} e$ as $e_i \xrightarrow{\mathsf{vis}} e$. Thus, we have

$$e_i \parallel e_{other}$$

However e_i only one of the least $B+1$ updates between e_{prev} and e. Thus there are at least $B+1$ updates which are concurrent with e_{other}, which contradicts the fact that T is B-concurrent trace.

Hence our initial assumption that $e_{other} \not\xrightarrow{\mathsf{vis}} e$ is incorrect.

Hence, $e_{other} \xrightarrow{\mathsf{vis}} e$. □

We next show that this function satisfies all the requirements of a secondary information function

Lemma 120. *$F_{(f,B)}$ defined above is a secondary information function for B-concurrent traces.*

Proof. Note that by definition of recent events, for $V = \partial_T(e)$ for an r-event e, and any replica r', $f(V_{r'}) \cap \mathcal{E}^{r'}_{\mathsf{Updates}} \subseteq \mathsf{Recent}^{r'}_{f,B}(V) \subseteq V \cap \mathcal{E}^{r'}_{\mathsf{Updates}}$. Since $F_{(f,B)}(r', V) = (\mathsf{Recent}^{r'}_{f,B}(V), \xrightarrow{r'})$ contains $\mathsf{Recent}^{r'}_{f,B}(V)$, the first constraint for secondary information is satisfied.

As we have shown in the proposition 117, for any replica $r' \neq r$, $\mathsf{Recent}^{r'}_{f,B}(V) = \mathsf{Recent}^{r'}_{f,B}(V_{r'})$. Thus, $F_{(f,B)}(r', V) = F_{(f,B)}(r', V_{r'})$. Thus the second condition is satisfied.

We have shown through proposition 117 that the $F_{(f,B)}(r, V)$ is locally computable, thus satisfying the third condition.

Suppose an r-event $e' \notin \mathsf{Recent}^{r}_{f,B}(V)$. By definition, $e' \notin f(V_r)$ and since $V_r \subseteq V$, $e' \notin f(V)$. Since $V = \partial_T(e)$, for any event e'' such that $e \xrightarrow{\mathsf{vis}} e''$, by definition of the primary information function, $e' \notin f(\partial_T(e''))$. Thus, the fourth condition of secondary information function is satisfied.

Finally, suppose an r-event $e' \notin \mathsf{Recent}^{r}_{f,B}(V)$. Let e'' be the some update event at a replica r'' such that $e' \in f(\partial_T(e'')$. By lemma 119, we know $e'' \xrightarrow{\mathsf{vis}} e$. Thus by causal delivery, it follows that at any replica r', we have $\varphi^{-1}_{r'}(e'') \xrightarrow{r'} \varphi^{-1}_{r'}(e)$. Thus, the fifth condition is satisfied.

Hence, $F_{(f,B)}$ is a well-defined secondary function. □

Thus, we have shown that $F_{(f,B)}$ is a bounded secondary information function for an M-bounded primary information f for an implementation B-concurrent traces. The bound on $F_{(f,B)}$ is $M + B$. From this and Theorem 112), we can conclude the following.

Theorem 121. *If f is a bounded primary function defined on a B-concurrent implementation, then, there there is a bounded solution for the generalized gossip problem for f.*

5.3.2 Bounding CmRDTs using Generalized Gossip Problem

We recall the definition of the bounded specification from the previous chapter.

Definition 122 (Bounded Specification)**.** *A computable specification $Spec_{\mathcal{D}}$ is said to be a bounded specification if the associated function that extracts the relevant context always produces a context of bounded size. Formally, $Spec_{\mathcal{D}}$ is* bounded specification *iff*

$$\exists K \in \mathbb{N} : \forall U \in Ctxt^{arb}(\mathcal{D}) : \ \forall q \in \mathsf{Queries} : \ \forall args \in \mathsf{Univ}^* : \ |\mathsf{RelevantCtxt}(U, q, args)| < K$$

Since we are interested in arbitration agnostic specification, we will look at U as a view instead of a context. Thus, the $\mathsf{RelevantCtxt}$ function picks for every view, a bounded subview that is sufficient to answer a query with some argument. Suppose the size of the universe $|\mathsf{Univ}|$ is bounded. Then since there are finitely many queries, there are finitely many arguments for the query, since every query has a fixed arity say m. Thus, the maximum number of arguments is bounded by Univ^m. Thus, the total number of combinations of queries from $\mathsf{Queries}$ and arguments from Univ^* is bounded.

In any view, we define the set of events that feature in the relevant context of some query for some argument as the set of Relevants Events. Formally

Definition 123 (Relevant Events)**.** *Given a replicated data type $\mathcal{D}$ over a bounded universe* Univ*, and bounded specification $Spec_{\mathcal{D}}$, with* RelevantCtxt *being the relevant-context function which is bounded, we define the set of relevant events in a view V of any trace of $\mathcal{D}$, denoted by, $\mathcal{E}^{Rel}_{Spec_{\mathcal{D}}}(V)$, as follows*

$$\mathcal{E}^{Rel}_{Spec_{\mathcal{D}}}(V) = \bigcup_{q \in \mathsf{Queries}} \bigcup_{args \in \mathsf{Univ}^{\mathsf{Arity}(q)}} Events(\mathsf{RelevantCtxt}(V, q, args))$$

Now suppose V' is a subview of V, that is $V' \subseteq V$ such that V' contains all the relevant events of V with respect to $Spec_{\mathcal{D}}$. Thus $\mathcal{E}^{Rel}_{Spec_{\mathcal{D}}}(V) \subseteq Events(V')$. Thus, for any query q and for any valid arguments to the query *args*, the relevant context $\mathsf{RelevantCtxt}(V, q, args) \subseteq V'$. Thus we have

$$\mathsf{RelevantCtxt}(V, q, args) \subseteq V' \subseteq V$$

From definition 101 in the previous chapter, the relevant context of for q and *args* in V' is the same as the relevant context for q and *args* in V. Thus,

$$\mathsf{RelevantCtxt}(V', q, args) = \mathsf{RelevantCtxt}(V, q, args)$$

Thus it is sufficient for replica with a view V to maintain the subview V' in order to answer every query correctly. Suppose there exists a function $f_{Spec_{\mathcal{D}}}()$ which can extract such a subview V' for a view V. We define such a function to be be a *specification subview function.*

Definition 124 (Specification Subview function)**.** *Given a replicated data type $\mathcal{D}$ over a bounded universe* Univ*, and bounded specification $Spec_{\mathcal{D}}$, with* RelevantCtxt *being the relevant-context function which is bounded, we define the* specification subview function *of a view V, denoted by $f_{Spec_{\mathcal{D}}}(V)$, such that*

- *$f_{Spec_{\mathcal{D}}}(V)$ is a subview of V,*
$$f_{Spec_{\mathcal{D}}}(V) \subseteq V$$
- *$f_{Spec_{\mathcal{D}}}(V)$ contains all the relevant events in V with respect to the specification $Spec_{\mathcal{D}}$.*
$$\mathcal{E}^{Rel}_{Spec_{\mathcal{D}}}(V) \subseteq Events(f_{Spec_{\mathcal{D}}}(V))$$

We now provide a sufficient condition for a CmRDT to have a bounded implementation.

Theorem 125. *A CmRDT $\mathcal{D} = (\mathsf{Univ}, \mathsf{Queries}, \mathsf{Updates})$ has a bounded implementation in a distributed system whose underlying network guarantees B-concurrent traces if there exists a locally computable specification-subview function $f_{Spec_{\mathcal{D}}}$ that is a bounded primary information function.*

Proof. From Theorem 121, we know that the generalized gossip problem has a bounded solution in a distributed system whose traces are B-concurrent. It is given that $f_{Spec_{\mathcal{D}}}()$ is a bounded primary information function which has a solution to the generalized gossip problem. Thus a replica r with a V would have the bounded primary information $f_{Spec_{\mathcal{D}}}(V)$. Since $f_{Spec_{\mathcal{D}}}$ is also a specification-subview function for $\mathcal{D}$, by definition, the information maintained is sufficient to answer every query correctly as per the specification of $Spec_{\mathcal{D}}$.

Thus, whenever a replica gets an update request $u(args)$ from the client, it is sufficient if it annotates the new event e with $u(args)$ and invokes the bounded solution to the generalized gossip problem. Also, it is sufficient to implement each query operation $q \in \mathcal{Q}$ as per the specification of $Spec_{\mathcal{D}}(f_{Spec_{\mathcal{D}}}(V), q, args)$. This provides a bounded implementation for $\mathcal{D}$. □

We will now show that OR-Sets has a bounded implementation over a bounded universe when the all its traces are B-Bounded.
Let us revisit the case of OR-Sets for which the universe Univ is bounded by an integer N_{Univ}.

There is only one query, $\mathsf{contains} \in \mathsf{Queries}$. Further, $\mathsf{Arity}(\mathsf{contains}) = 1$. In a view V, let V_x denote the set of all $\mathsf{add}(x)$ and $\mathsf{delete}(x)$ events in V. Further, for a replica r, let $max^r_x(V)$ denote the set of maximal r-update event in V_x. Note that the size of this set is either 0 or 1.

Definition 126 (Relevant Context for OR-Set)**.** *Let V be any view in a B-concurrent trace T of OR-set. We define the relevant context as follows. Let $x \in \mathsf{Univ}$, be some element of the universe. We define the relevant context for $\mathsf{contains}(x)$ in V to be*

$$\mathsf{RelevantCtxt}_{ORSet}(V, \mathsf{contains}, x) = max(V_x)$$

Since the $max(V_x)$ can contain at most one event per replica and since the number of replicas is bounded i.e $|\mathcal{R}| = N$, for any $x \in \mathsf{Univ}$, $|\mathsf{RelevantCtxt}_{ORSet}(V, \mathsf{contains}, x)| \leq N$.

It can be seen that

1. For any view V, $\mathsf{RelevantCtxt}_{ORSet}(V, \mathsf{contains}, x) = max(V_x) \subseteq V$ thus satisfying the first condition of relevant context definition 101.

2. Suppose for any view V' such that $\mathsf{RelevantCtxt}_{ORSet}(V, \mathsf{contains}, x) \subseteq V' \subseteq V$, then since $\mathsf{RelevantCtxt}_{ORSet}(V, \mathsf{contains}, x) = max(V_x)$, by maximality of events we have $max(V'x) = max(V_x)$.

   But $max(V'_x)$, by definition is the same as $\mathsf{RelevantCtxt}_{ORSet}(V', \mathsf{contains}, x)$. Thus, we have

   $$\mathsf{RelevantCtxt}_{ORSet}(V', \mathsf{contains}, x) = \mathsf{RelevantCtxt}_{ORSet}(V, \mathsf{contains}, x)$$

   thus satisfying the third requirement of a relevant context from definition 101.

Thus, the relevant context of OR-Set above is well-defined. We now define the relevant events for OR-Set as follows

Definition 127. *Let V be a view in some trace of OR-Set. We define the relevant events of $OR-Set$ in V, denoted by $\mathcal{E}^{Rel}_{Spec_{ORSet}}(V)$ to be the set*

$$\mathcal{E}^{Rel}_{Spec_{ORSet}}(V) = \bigcup_{x \in \mathsf{Univ}} Events(\mathsf{RelevantCtxt}_{ORSet}(V, \mathsf{contains}, x)))$$

We now provide the bounded specification of OR-Set as follows.

Definition 128 (Bounded Specification for OR-Set)**.** *Suppose T is a trace of a run of the OR-Set. Let V be a view in T. Let $x \in \mathsf{Univ}$ be some element in the universe. Then the specification of OR-Set is as follows*

$$Spec_{ORSet}(V, \mathsf{contains}, x) = \mathsf{True} \iff \exists e \mathsf{RelevantCtxt}_{ORSet}(V, \mathsf{contains}, x) : Op(e) = \mathsf{add}$$

Thus, the bounded specification for OR-Sets where the updates are causally delivered searches for an $\mathsf{add}(x)$ from among the maximal x events in a view.

From this, we can define the specification subview function for OR-Sets.

Definition 129 (Specification Subview for OR-Sets)**.** *Let V be a view in some trace of OR-Set. Let $\mathcal{E}_{max}(V)$ denote the set of events*

$$\mathcal{E}_{max}(V) = \bigcup_{r \in \mathcal{R}, x \in \mathsf{Univ}} max^r_x(V)$$

which contains the maximal x event for every replica r for every element x in the universe.

Then, we define the function $f_{Spec_{ORSet}}()$ over the view V to be the subview of V defined by $\mathcal{E}_{max}(V)$. Thus

$$f_{Spec_{ORSet}}(V) = V|_{\mathcal{E}_{max}(V)}$$

Note that for any event r-update event e that is a relevant event in V we have

$$\begin{aligned}
e \in \mathcal{E}^{Rel}_{Spec_{ORSet}}(V) &\iff Args(e) = x \wedge e \in \mathsf{RelevantCtxt}_{ORSet}(V, \mathsf{contains}x) \\
&\iff e \in max(V_x) \\
&\implies e \in max^r_x(V) \\
&\implies e \in \mathcal{E}_{max}(V) \\
&\implies e \in Events(f_{Spec_{ORSet}}(V))
\end{aligned}$$

Thus, $\mathcal{E}^{Rel}_{Spec_{ORSet}}(V) \subseteq Events(f_{Spec_{ORSet}}(V))$ thus making $f_{Spec_{ORSet}}$ a specification subview function.

Note that since the universe is bounded by N_{Univ}, in any view V, $|\mathcal{E}_{max}(V)| \leq N \times N_{\mathsf{Univ}}$. Thus, $f_{Spec_{ORSet}}$ is a bounded function with a bound $N \times N_{\mathsf{Univ}}$.

If we show that $f_{Spec_{ORSet}}$ is a primary information function that is locally computable, it follows from Theorem 125 that OR-Sets have a bounded implementation where the underlying network permits only B-Concurrent runs, such that the bounded implemenatation is correct with respect to $Spec_{ORSet}$.

Lemma 130. *$f_{Spec_{ORSet}}$ is a primary information function.*

Proof. We need to show that $f_{Spec_{ORSet}}$ satisfies the four conditions of the primary information function.

1. In any view V, $f_{Spec_{ORSet}}(V)$ is a finite (since V is finite) fragment of V containing only the update events. Thus, the output of $f_{Spec_{ORSet}}$ is an information graph $\mathcal{G}(V)$ of V.

2. In any trace T, if e is a Query or an **usend** event at a replica r, then by definition, $\partial_T(e) = \partial_T(e')$ where $e' = Pred^r_T(e)$. Thus, $f_{Spec_{ORSet}}(\partial_T(e)) = f_{Spec_{ORSet}}(\partial_T(e'))$.

3. Let e be an update event at replica r. Let $V = \partial_T(e)$. Let $Args(e) = x$. By definition, $e \in max^r_x(V)$. Thus, $e \in \mathcal{E}_{max}(V)$. Hence, $e \in \mathcal{E}^{Rel}_{Spec_{ORSet}}(V)$. Which by definition implies $e \in f_{Spec_{ORSet}}(V)$.

4. Suppose $e' \in Pred_T(e)$. And some event $e'' \in f_{Spec_{ORSet}}(\partial_T(e) \cap \partial_T(e')$. Let e'' be an r-event with $Args(e'') = x$. Then, by defintion, $e'' \in max^r_x(\partial_T(e))$. Also since $e'' \in \partial_T(e')$, it implies that $e'' \in max^r_x(\partial_T(e'))$. Thus, by definition $e'' \in f_{Spec_{ORSet}}(\partial_T(e'))$.

Thus, $f_{Spec_{ORSet}}$ is a primary information function. □

We now show that $f_{Spec_{ORSet}}$ is locally computable.

Lemma 131. *In any causally-delivered trace T of an OR-Set, for any event e, the primary information $f_{Spec_{ORSet}}(\partial_T(e))$ is computable from e, and $f_{Spec_{ORSet}}(\partial_T(e'))$ where $e' \in Pred_T(e)$.*

Proof. We shall prove this by the height of the event e. The base case for an empty trace T is trivial since $f_{Spec_{ORSet}}$ computes an empty view.

Assume that the result is true for all events of height smaller than n.

Now consider an event e of height n. Let $Rep(e) = r$. Let $e' = Pred^r_T(e)$. Let $V = \partial_T(e)$, and $V' = \partial_T(e')$.

e **is a query or a usend event:** In this case $V = V'$ and by definition, $f_{Spec_{ORSet}}(V) = f_{Spec_{ORSet}}(V')$. Thus, in these cases, $f_{Spec_{ORSet}}$ is locally computable.

e **is an update operation:** Let $Args(e) = x$.

Let $f_{Spec_{ORSet}}(V') = (E', <')$ and $f_{Spec_{ORSet}}(V) = (E, <)$. Note that $e \in max^r_x(V)$.

For any $e'' \in E'$, if $Rep(e'') = r''$ and $Args(e'') = x''$, then, $e'' = max^{r''}_{x''}(V')$. If either $r'' \neq r$ or $x'' \neq x$, then, $max^{r''}_{x''}(V) = max^{r''}_{x''}(V')$. For $r'' = r$ and $x'' = x$, $max^{r''}_{x''}(V) = max^r_x(V) = e$. Thus, if $E_{in} = \{e'' \in E' \mid Rep(e'') \neq r \vee Args(e'') \neq x\}$ then, $E = E_{in} \cup \{e\}$.

For any pair of events $e'', e''' \in E_{in}$, we have the ordering between them via $<'$. Thus we set $< |_{E_{in}} =<' |_{E_{in}}$. For all $e'' \in E_{in}$, $e'' < e$. Thus, $<=<' |_{E_{in}} \cup \{(e'', e) \mid e'' \in E_{in}\}$.

Thus, in this case $f_{Spec_{ORSet}}$ can be locally computed.

e **is an ureceive operation:** Let $e'' = \varphi(e)$. Let $Rep(e'') = r''$. Let $Args(e'') = x''$. Let $V'' = \partial_T(e'')$. Let $f_{Spec_{ORSet}}(V') = (E', <')$ and $f_{Spec_{ORSet}}(V) = (E, <)$ and $f_{Spec_{ORSet}}(V'') = (E'', <'')$.

Note that $V'' \setminus \{e''\} \subseteq V'$, since due to causal delivery, all the updates in V'' apart from e'' have already been delivered at r at or before e'. Hence they are in V'.

Thus, for any replica r' and an element x', such that either $r' \neq r''$ or $x' \neq x''$, $max^{r'}_{x'}(V'') \xrightarrow{\mathsf{vis}} max^{r'}_{x'}(V')$. Thus, $r' \neq r''$ or $x' \neq x''$, $max^{r'}_{x'}(V) = max^{r'}_{x'}(V')$.

We know that $e'' \in max^{r''}_{x''}(V)$. Thus, if $E_{in} = \{e''' \in E' \mid Rep(e''') \neq r'' \vee Args(e''') \neq x''\}$, since the events in E are $\bigcup_{x \in \mathsf{Univ}} \bigcup_{r \in \mathcal{R}} max^r_x(V)$, we can write $E = E_{in} \cup \{e''\}$.

For events in E_{in}, $<'$ provides the ordering among the events.

Further, for any event $e''' \in E' \setminus E''$, if $Rep(e''') = r' \neq r''$ or $Args(e''') = x' \neq x''$, $e''' \in max^{r'}_{x'}(V')$. Thus, it is not the case that $e''' \xrightarrow{\mathsf{vis}} e''$ since otherwise it will imply that $e''' \xrightarrow{\mathsf{vis}} max^{r'}_{x'}(V'')$ thereby contradicting the fact that $max^{r'}_{x'}(V'') \xrightarrow{\mathsf{vis}} max^{r'}_{x'}(V')$. Similarly it cannot be the case that $e'' \xrightarrow{\mathsf{vis}} e'''$ since by causal delivery that would imply $e'' \in E'$, which we know is not the case since the update e'' became known to r through the update-receive event e. Thus, for all events $e''' \in E' \setminus E''$, $e'''||e'$.

For $e''' \in E_{in} \cap E''$, the ordering between e''' and e'' is given by $<''$.

Thus $<=<' |_{E_{in}} \cup <'' |_{(E_{in} \cup \{e''\}) \cap E''}$.

Thus $(E, <)$ can be computed locally. Hence $f_{Spec_{ORSet}}$ can be computed locally in this case as well.

Thus, we have shown that for every event, we can locally compute specification subview $f_{Spec_{ORSet}}$ from the event and the the specification subviews at its predecessors. □

From Lemmas 130 and 131, along with Theorem 125, we can conclude the following.

Theorem 132. *Over a finite universe* Univ*, there exists a bounded implementation of OR-Sets whose runs are B-concurrent and which is correct as per the specification $Spec_{ORSet}$.*

5.4 Summary

Borrowing ideas from Mazurkiewicz trace theory, in this chapter we have formulated a generalization of the gossip problem and shown that this can be used to derive bounded implementations for replicated data types, provided we have an additional guarantee of bounded concurrency. Though bounded concurrency seems like a very strong property, it is automatically achieved if we combine causal message delivery with bounded message delays. The only complication that can arise is from a replica crashing. However, if we assume that when a replica wakes up from a crash, it first processes all pending receive actions before initiating any sends, we retain bounded concurrency. Note that causal delivery is also infeasible if we do not make similar assumptions about how a crashed process recovers. Our main contribution in this chapter is a systematic approach to construct bounded reference implementations for replicated data types. We will show in the next chapter that this kind of implementation is useful for both verification and testing.

6

Bounded Reference implementations of Replicated Data Types using Later Appearance Records (LAR)

In the previous chapter, we had described a principled approach to constructing a bounded reference implementation for replicated data type from its bounded declarative specification. This construction produced a distributed reference implementation where each replica only keeps a fragment of its local view of the overall run in the form of primary and secondary information. This required an intricate distributed timestamping protocol [Mukund and Sohoni, 1997; Mukund et al., 2003, 2015a] in order to safely reuse the timestamps so that the state at every replica remains boudned. Moreover, it required strong assumptions about B-Concurrent runs which had to be guaranteed by the underlying network have to be directly incorporated into the reference implementation.

In this chapter, we propose a simple *global* reference implementation for replicated data types with declarative specifications and simple conditions under which this is guaranteed to be finite. Our implementation uses the technique of *Later Appearance Record* (LAR). We note that the main aim of generating a reference implementation is to come up with an effective verification procedure for a given CRDT implementations. The key observation of this chapter is that a *global* reference implementation is sufficient for this purpose. In a global reference implementation, we can directly keep track of the *happened-before* relation between update events without exchanging additional information between replicas. In fact, we show that we can maintain a local sequential history for each replica in terms of a later appearance record (LAR) [Gurevich and Harrington, 1982], from which we can faithfully reconstruct the causality relation. This greatly simplifies the construction. Furthermore, the LAR-based construction does not require imposition of any bound over the concurrent update operations. Towards the end of the chapter we also outline a methodology for effective verification of CRDT implementations using CEGAR. This chapter is based on our work [Mukund et al.,

2015b].

6.1 Global Implementation of a Replicated Datatype

Recall that an abstract run is a pair (ρ, φ) where ρ is a sequence of operations of a replicated data type $\mathcal{D}$ and φ is a function that identifies the update (at a remote replica) corresponding to each receive operation in ρ. When we consider an implementation of a CRDT, its runs will typically be just sequences of operations. The function φ is not provided along with the run, but it is reasonable to assume that the implementation has enough extra information to identify the update operation corresponding to each receive event. One way to model this is to associate each operation with an *identifier*, which is typically a a natural number. Further, we assign the same identifier for an *update-receive* operation and its matching *update* operation. Since we are interested in finite-state CRDT implementations also, we would like to use a bounded linearly ordered set ID of identifiers as timestamps. It is simplest to assume that $ID \subseteq \mathbb{N}$.

We call such operations annotated with identifiers as *annotated operations*. Let $\Sigma(\mathcal{D}, ID) = \Sigma(\mathcal{D}) \times ID$ denote the set of annotated operations of the CRDT $\mathcal{D}$.

Definition 133 (Annotated Operations)**.** *We denote by ID the set of identifiers. It is simplest to assume that* $ID \subseteq \mathbb{N}$.

An annotated operation is an operation of a replicated data type annotated with an identifier from a set $ID \subseteq \mathbb{N}$.

For an annotated operation $o' = (o, id) \in \Sigma(\mathcal{D}) \times ID$*, we define* $Id(o') = id$ *and* $\psi(o') = \psi(o)$ *for* $\psi() \in \{Rep(), Op(), Ret(), Args()\}$.

Thus, we can now define a annotated run to be a sequence of annotated operations. However, it cannot be an arbitrary sequence, since we need to ensure that for every *update-receive* operation, its identifier is the same as the identifier on the matching *update* operation. We formally define a *well-formed* annotated run below.

Definition 134 (Well-formed Annotated Run)**.** *We say that an* annotated run $\rho' \in (\Sigma(\mathcal{D}) \times ID)^*$ *is* well-formed *if timestamps are assigned sensibly, as follows.*

- *For every* update-send *operation* $\rho'[j]$*, the previous operation,* $Op(\rho'[j-1]) \in \mathsf{Updates}$ *and has the same identifier as the* update-send

$$Id(\rho'[j]) = Id(\rho'[j-1])$$

- *For every* update-receive *operation* $\rho'[j]$*, there is* $i < j$ *such that* $Id(\rho'[i]) = Id(\rho'[j])$*, and* $\rho'[i]$ *is an* update operation *such that for all* $k \in [i+1..j-1]$*,*

$$Op(\rho'[k]) = \textbf{ureceive} \implies Rep(\rho'[k]) \neq Rep(\rho'[j]) \vee Id(\rho'[k]) \neq Id(\rho'[j]).$$

- *For $i < j$, if $\rho'[i]$ and $\rho'[j]$ are update operations and $Id(\rho'[i]) = Id(\rho'[j])$, then for every replica $r \neq Rep(\rho'[i])$, there is a $k \in [i+1..j-1]$ such that $Op(\rho'[k]) =$* **ureceive***, $Rep(\rho'[k]) = r$ and $Id(\rho'[k]) = Id(\rho'[i])$.*

These constraints ensure that before the timestamp corresponding to an update *operation is reused later on in the run, the corresponding* update-receive *are delivered to the all the other replicas in the run.*

The first condition ensures that the timestamps match a send operation to the previous update operation. The second condition captures the fact that timestamps unambiguously match receive events to update operations. The third condition prevents a timestamp from being reused before it has been received by all replicas.

Given a annotated run, we can define the appropriate abstract run as follows.

Definition 135 (Abstract Run associated with a Annotated Run)**.** *The run associated with a well-formed annotated run $\rho' = ((o_1, id_1), (o_2, id_2), \ldots, (o_m, id_m))$ is a pair (ρ, φ) such that*

- $\rho = o_1 o_2 \cdots o_m$
- *For any $i \leq |\rho'|$, if o_i is a* **ureceive***-operation,*

$$\varphi(i) = max\{j < i \mid id_j = id_i \text{ and } Op(o_j) \in \mathsf{Updates}\}$$

In what follows, we consider only well-formed annotated runs.

Lemma 136. *For every run (ρ, φ) of $\mathcal{D}$, we can identify a set ID such that there is a well-formed annotated run $\rho' \in (\Sigma(\mathcal{D}) \times ID)^*$ whose associated run is (ρ, φ).*

Proof. All query operations can be labelled with a fixed identifier (say 0, for concreteness). Each update operation $\rho[i]$ is labelled with the smallest identifier in ID that does not label any undelivered update operation in $\rho[1 : i-1]$. Every receive operation $\rho[i]$ is labelled by the same identifier that labels $\rho[j]$, where $\varphi(i) = j$. □

We recall the definition of an implementation of a replicated data type.

Definition 137 (Implementation of a replicated data type and its runs)**.** *An implementation of a CRDT $\mathcal{D}$ is a tuple $\mathcal{D}_I = (\mathcal{C}, C_\perp, ID, \to)$ where:*

- *$\mathcal{C}$ is set the configurations, where each configuration represents a the states of all the replicas at a given time..*
- *$C_\perp \in \mathcal{C}$ is the initial configuration corresponding to the initial configuration.*
- *$ID \subseteq \mathbb{N}$ is the set of identifiers, which serve as identifiers.*

- $\rightarrow \subseteq C \times (\Sigma(\mathcal{D}) \times ID) \times C$ *is the transition relation.*

A annotated run $\rho' = o'_1 \cdots o'_n$ *is accepted by* $\mathcal{D}_I$ *if there exists a sequence of configurations* $C_0 C_1 \cdots C_n$ *such that* $C_0 = C_\perp$*, and for every* $i \leq n$*,* $C_{i-1} \xrightarrow{o'_i} C_i$*.* (ρ, φ) *is a run of* $\mathcal{D}_I$ *if it is the run associated with a well-formed annotated run* ρ' *accepted by* $\mathcal{D}_I$*. We denote the set of all runs of* $\mathcal{D}_I$ *by* $Runs(\mathcal{D}_I)$*.*

The implementation is correct with respect to $Spec_{\mathcal{D}}$ iff $Runs(\mathcal{D}_I) \subseteq Runs(\mathcal{D}, Spec_{\mathcal{D}})$.

We now present a reference implementation of a replicated data type

$$\mathcal{D} = (\mathsf{Univ}, \mathsf{Updates}, \mathsf{Queries}, \mathsf{Ret})$$

with a declarative specification $Spec_{\mathcal{D}}$. The reference implementation, denoted $\mathcal{D}_{ref}$, satisfies the property that $Runs(\mathcal{D}_{ref}) = Runs(\mathcal{D}, Spec_{\mathcal{D}})$.

6.1.1 Reference Implementation

Before we describe the reference implementation, we present the ingredients needed. The aim is to maintain as little information as possible to respond to each query. The key observation is that the reference implementation is *global*—it can pool together information stored at all the replicas without paying the cost of communication. If we have a declarative specification $Spec_{\mathcal{D}}$ of $\mathcal{D}$ that is computable via $\mathsf{RelevantCtxt}$ and $\mathsf{ComputeRet}$, then each replica needs to maintain $\bigcup_{q,args} \mathsf{RelevantCtxt}(V, q, args)$, where V is the view of some replica r at any point in time. The important ingredient in $\mathsf{RelevantCtxt}$ is the precedence relation between events, and hence the reference implementation needs to store enough information to recover this. The implementation also needs to intelligently discard information that will no longer prove useful.

The most direct implementation would store (as part of the "state" of each replica) the relevant suffix of the trace—the upward closure of the events in

$$\bigcup_{q,args} \mathsf{RelevantCtxt}(V, q, args).$$

But we choose a more compact representation called *Later Appearance Records* (LARs), from which the information needed to answer queries can be recovered. An LAR is a set of sequences rather than a partial order, and hence easier to manipulate.

We now define the building blocks of the reference implementation.

Let $\mathcal{L}$ be a (potentially infinite) set of labels, equipped with a total order $\leq$. We use labels to distinguish between multiple occurrences of the same update method at the same replica with the same arguments. Update operations equipped with labels are called nodes.

Definition 138 (Node)**.** *A* node *is a tuple* $(u, r, args, l) \in \mathsf{Updates} \times \mathcal{R} \times \mathsf{Univ}^* \times \mathcal{L}$*. For* $v = (u, r, args, l)$*, we define* $Op(v) = u$*,* $Rep(v) = r$*,* $Args(v) = args$ *and* $Label(v) = l$*. The set of all nodes is denoted by* $\mathcal{N}$*.*

Definition 139 (Later Appearance Record)**.** *A Later Appearance Record (LAR) is a* sequence *of distinct nodes. For a node* v *and an LAR* L*, we write* $v \in L$ *to denote that* v *appears in the sequence of nodes in* L*.*

For nodes $v_1, v_2 \in L$*,* $v_1 \leq_L v_2$ *if* v_1 *occurs earlier than* v_2 *in* L*. If* L *is an LAR and* W *is a set of nodes then* $L - W$ *is the subsequence of* L *consisting of nodes not in* W*. The set of all LARs is denoted by* LARS*.*

Each replica uses the LAR to record the order in which it has seen updates, originating locally as well as remotely. In an actual implementation, on an update request from the client, the replicas generate auxiliary information which is then broadcast to the other replicas over the network. The behaviour of the network is not under the control of the implementation. The network might sometimes provide additional guarantees about message delivery (such as *causal delivery* or *FIFO delivery*), and we can sometimes make use of these facts to simplify the implementation. Here we present the general case, without any assumptions about the network. In our reference implementation we would like to keep track of the delivery status of the auxiliary information broadcast by an update to the other replicas in the system. We model this information as a *network node*. Recall that a node is an update operation along with an unique label. A network node attaches to a node the identifier associated with the annotated update operation as well information about the state of replicas that have received the update.

Definition 140 (Network node)**.** *A network node is a member of* $\mathcal{N} \times ID \times 2^{\mathcal{R}}$*. The set of all network nodes is denoted by* $\mathcal{N}_{net}$*. For a network node* $v_{net} = (v, id, R)$ *we define* $Node(v_{net})$ *to mean* v*,* $Id(v_{net})$ *to mean* id *and define* $Rep(v_{net})$*,* $Id(v_{net})$*,* $Args(v_{net})$ *and* $Label(v_{net})$ *to be the corresponding functions applied on* v*. We use* $Delivered(v_{net})$ *to denote* R *which is the subset of the replicas which have received the broadcast corresponding to the update operation associated the identifier* id

A configuration of a reference implementation consists of the LAR of each replica along with the network nodes pertaining to undelivered updates. In order to keep only the relevant information, one of the aims of the reference implementation is to try to purge nodes from LARs whenever possible. A *consistent configuration* is one where these purges have been done safely. Specifically, replica r does not purge a node modelling a local update operation as long as that node it is present in the LAR of some other replica. Also, if a local update is yet to be delivered to some other replica, then r does not purge the corresponding node.

Definition 141 (Configuration)**.** *A configuration C is a member of $\mathsf{LARS}^{|\mathcal{R}|} \times 2^{\mathcal{N}_{net}}$. For any configuration $C = ((L_1, L_2, \ldots, L_N), V_{net})$, we denote by $C[r]$ the LAR L_r. We shall denote by C_{net} the set of network nodes V_{net}.*

We say that a configuration C is consistent iff

- *$\forall r, r'$ if there exists $v \in C[r]$ such that $Rep(v) = r'$ then $v \in C[r']$. This ensures that when a node is present in a remote LAR it is guaranteed to be present in the local LAR. This is a crucial property for recovering the causality relation between update operations modelled by the nodes in the LARs.*

- *$\forall v_{net} \in C_{net}$ if $r \in Delivered(v_{net})$ then $Node(v_{net}) \in C[r]$. This ensures that the delivery status of a broadcast of an update matches the state of the replica where the broadcast is delivered. Thus, if the network node, modelling the status of the broadcast by the network, says that the broadcast of an update is delivered at some replica, then the LAR of that replica should contain the node modelling that update operation.*

The trivial configuration denoted by $C_\perp$ is one where $\forall r \in \mathcal{R} : C_\perp[r]$ is the empty LAR and $C^0_{net} = \emptyset$. We denote the set of all consistent configurations by $\mathcal{C}_{ref}$.

Using the LARs of all the replicas, we can reconstruct the happened before relation for all events that are mentioned in a configuration. Suppose r sees two updates u' and u'' originating at r' and r''. Since updates are seen at the originating replica first before being seen by others, the relation between u' and u'' can be determined by their relative order of appearances in the LARs of r' and r''. Here we crucially use the fact that our implementation is *global*.

Definition 142 (Precedence and Concurrency)**.** *Let C be a consistent configuration. Let r be a replica and $v_i, v_j \in C[r]$ with $Rep(v_i) = r'$ and $Rep(v_j) = r''$. We say that v_i* precedes *v_j in C, denoted by $v_i \leq_C v_j$, if $(v_i \in C[r''] \wedge v_i \leq_{C[r'']} v_j) \wedge (v_j \in C[r'] \implies v_i \leq_{C[r']} v_j)$. (In other words, both r' and r'' locally see v_i before v_j.)*

If neither $v_i \leq_C v_j$ nor $v_j \leq_C v_i$ for any $v_i, v_j \in C[r]$, then we say that v_i and v_j are concurrent in C, denoted by $v_i \parallel_C v_j$.

For a consistent configuration C and replica r, the view of r in C, denoted by $\partial_C(r)$, and can be represented by the acyclic graph $(C[r], \leq_C)$.

If a node v in the LAR $C[r]$ of a replica r in a configuration C is such that it is in the relevant context of the view of that replica for some query q and argument *args*, then such a node should not yet be purged from the LAR of the replica in order that $\partial_C(r)$ is consistent with the view of the replica at that point. Otherwise the replica r would not be able to correctly answer the query $q(args)$ as per the specification. Hence such a node is termed to be a *relevant node*.

Definition 143 (Relevant node)**.** *Let $Spec_{\mathcal{D}}$ be a specification of $\mathcal{D}$ computable via* RelevantCtxt *and* ComputeRet*. We say that a node v in a consistent configuration C is relevant with respect to $Spec_{\mathcal{D}}$ if there exists a replica r, query $q \in$ Queries and $args \in \mathsf{Univ}^*$, such that $v \in \mathsf{RelevantCtxt}(\partial_r(C), q, args)$.*

We shall now provide the details of the reference implementations and how the configurations evolve along a run.

6.1.2 Details of the reference implementation

The reference implementation is formally presented below. In this implementation each replica in the configuration maintains an LAR to which it appends information pertaining to each local update. On receiving information about a remote update, it again appends this to the LAR, and also seeks to purge from all LARs nodes that have ceased to become relevant and have been seen by all replicas. This enables the reuse of labels. Since in any view V of a replica in a trace T, the relevant nodes subsume all the subtraces of the form $\mathsf{RelevantCtxt}(V, q, args)$, it follows that the implementation never purges information that is needed to answer a query.

Let $Spec_{\mathcal{D}}()$ be the declarative specification of a CRDT $\mathcal{D}$ computable via functions RelevantCtxt() and ComputeRet(). The global reference implementation of the CRDT is defined to be $\mathcal{D}_{ref} = (\mathcal{C}_{ref}, C_\perp, ID, \rightarrow_{ref})$ where $ID = \mathbb{N}$ and $\rightarrow_{ref}$ is defined as follows.

Let $C, C' \in \mathcal{C}$ and let o be an operation from $\Sigma(\mathcal{D}) \times ID$ with $Rep(o) = r$, $Args(o) = args$, $Id(o) = id$. Then the transition relation due of the reference implementation for the operation o, that transforms the configuration C into the configuration C', denoted by $C \xrightarrow{o}_{ref} C'$, iff one of the following holds:

- $Op(o) = q \in$ Queries and $ret =$ ComputeRet$(\partial_C(r), q, args)$ and $C' = C$.
- $u = Op(o) \in$ Updates, $\forall v_{net} \in C_{net} : Id(v_{net}) \neq id$, and C' is defined as follows:
  - $\forall r' \in \mathcal{R} : r' \neq r \implies C'[r'] = C[r']$.
  - $C'[r] = C[r].v$, with $v = (u, r, args, l)$ where l is a label such that $\forall v' \in C[r] : Label(v') \neq l$.
  - $C'_{net} = C_{net} \cup \{(v, id, \{r\})\}$.
- $Op(o) =$ **usend**, and $C' = C$ and there exists a node v such that $(v, id, \{r\}) \in C_{net}$
- $Op(o) =$ **ureceive** and there exists a node v and $R \subseteq \mathcal{R}$ such that $(v, id, R) \in C_{net}$ and $r \notin R$, and C' is defined as follows:

  Let C'' be an intermediate configuration defined as

- $\forall r' \neq r : C''[r'] = C[r']$.
- $C''[r] = C[r].v$.
- $C''_{net} = C_{net} \cup \{(v, id, R \cup \{r\})\} \setminus \{(v, id, R)\}$.

If $R \cup \{r\} \neq \mathcal{R}$ then $C' = C''$ else

- $\forall r' \in \mathcal{R} : C'[r'] = C''[r'] - W$, where

$$W = \{v' \in \bigcap_{r' \in \mathcal{R}} C''[r'] \mid v' \text{ is not relevant in } C''\}.$$

- $C'_{net} = C''_{net} \setminus \{(v, id, R \cup \{r\})\}$.

Thus the configuration C' is the same as the original configuration C' on a query operation and the return value of the query should be as per the return value provided by the specification of the replicated data type when applied to the view of the replica r, where the view is constructed from the LAR $C[r]$ of the replica r

On an update operation at replica r, we append the LAR of the replica r in the configuration C with the new node v corresponding to the new update operation. We also add this node into the list of network nodes V_{net} while marking that this node has been delivered at replica r.

On an update-send operation at replica r, the configuration C' is the same as the original configuration. Furthermore it is expected that the original configuration has a network node in $C.net$ corresponding to the update operation that would have occurred just before this update-send. We identify this node by verifying the *id* of the network node and also the fact that the set of replicas tracking the delivered update should only contain r.

Finally, on an update-receive operation at replica r, we append the LAR of the replica r with the new node v. We also update the list of network nodes V_{net} to mark that the update v is delivered at replica r. Following this, we check if the update v has been delivered at all the replicas. If so, it gives us an opportunity to compute whether due to the arrival of v some other nodes in the configuration have ceased to become relevant. If that is the case, we purge all those node v' which have been delivered at all the replicas, and which are no longer relevant at the replicas. Finally, we purge the network node corresponding to v from the list of network nodes, since the update has been delivered at all the replicas.

We will now show the correctness of the global reference implementation.

6.1.3 Correctness of the reference implementation

Lemma 144. *Every reachable configuration C of $\mathcal{D}_{ref}$ is consistent.*

Proof. The initial configuration is trivially consistent, and each transition purges only those nodes that are no longer relevant and are delivered to every replica. This proves the lemma. □

In the presentation below, we shall denote the trace of the run (ρ, φ) as $\mathcal{T}(\rho, \varphi)$. Further suppose $T = \mathcal{T}(\rho, \varphi)$, and r is a replica and $max_r(T)$ denotes the maximal r-event in T, then we shall denote by $\partial_T(r)$ the view of the replica r in T which is $\partial_T(max_r(T))$.

Lemma 145. *Suppose $\rho' \in (\Sigma(\mathcal{D}) \times ID)^*$ is accepted by $\mathcal{D}_{ref}$ and that $C_0 \xrightarrow{\rho'}_{ref} C$. Let (ρ, φ) be the run associated with ρ' and $T = \mathcal{T}(\rho, \varphi)$. Then, for all r, q and $args$,*

$$\mathsf{RelevantCtxt}(\partial_T(r), q, args)$$

is isomorphic to

$$\mathsf{RelevantCtxt}(\partial_C(r), q, args).$$

Proof. The proof is by induction on the length of ρ'. The case when $\rho' = I$ is trivial. So let $\rho' = \sigma'.o$. Let C' be a configuration such that $C_0 \xrightarrow{\sigma'}_{ref} C' \xrightarrow{o}_{ref} C$. Let (σ, φ) be the run corresponding to σ' and let $T' = \mathcal{T}(\sigma, \varphi)$. We assume by the induction hypothesis that for all r, q and $args$, $\mathsf{RelevantCtxt}(\partial_{T'}(r), q, args)$ is isomorphic to $\mathsf{RelevantCtxt}(\partial_{C'}(r), q, args)$. There are three cases to be considered.

o is a query operation: In this case $C = C'$ and $\partial_T(r) = \partial_{T'}(r)$ since the update operations visible to r are the same in both the trace. So the lemma follows.

o is an update operation: Suppose $Rep(o) = r$. For $r' \neq r$, it is clear from the transition rules that $C[r'] = C'[r']$. It is also the case that $\partial_T(r') = \partial_{T'}(r')$, so the result is true holds for the relevant context of any query at replicas other than r.

On the other hand, $C[r] = C'[r].v$ where v is a node with a fresh label l, corresponding to the operation o. Since v is the latest node in $C[r]$ and $v \notin C[r']$ for any other r', it is clear that $v' \leq_C v$ iff $v' \in C[r]$. But $v' \in C[r]$ iff v' corresponds to an update received by r or originating in r. Thus $Events(\partial_C(r)) = Events(\partial_{C'}(r)) \cup \{v\}$, with v being the maximal element in $\partial_{C'}(r)$. It is easy to see that the maximal r-event in the trace T is greater than all other events in $\partial_{T'}(r)$. Thus $\mathsf{RelevantCtxt}(\partial_C(r), q, args)$ is isomorphic to $\mathsf{RelevantCtxt}(\partial_T(r), q, args)$.

o is a receive operation: Suppose $Rep(o) = r$. We add a node at the end of $C[r]$, but also purge all the LARs of some irrelevant nodes (those that are received by every replica). Since irrelevant nodes do not feature in $\mathsf{RelevantCtxt}(\partial_T(r'), q, args)$ for any r' and $q(args)$, all we need to show is that the order among relevant nodes is captured correctly. But the order between update events does not change at the point of time of a receive. It can be checked that $\leq_C = \leq_{C'}$, and thus the lemma follows.

□

We next show that the runs of our global reference implementation are correct with respect to the declarative specification of the replicated data type.

Lemma 146. *Suppose a well-formed annotated run $\rho' \in (\Sigma(\mathcal{D}) \times ID)^*$ is accepted by $\mathcal{D}_{ref}$. Let (ρ, φ) be the run associated with ρ'. Then $(\rho, \varphi) \in Runs(\mathcal{D}, f)$.*

Proof. Suppose $C_0 \xrightarrow{\rho'[1,\ldots,i]}_{ref} C_i$. Let $T_i = \mathcal{T}(\rho[1,\ldots,i], \varphi_{[1,\ldots,i]})$. Since $\mathsf{RelevantCtxt}(\partial_{C_i}(r), q, args)$ is isomorphic to $\mathsf{RelevantCtxt}(\partial_{T_i}(r), q, args)$ and since $\mathsf{ComputeRet}$ returns the same values on isomorphic traces, it easily follows that for all query operations $\rho[i]$ of the form $(q, r, args, ret)$, we have $Ret(\rho[i]) = Spec_{\mathcal{D}}(\partial_{T_i}(r), q, args)$. Thus $(\rho, \varphi) \in Runs(\mathcal{D}, Spec_{\mathcal{D}})$. □

We now show that every correct run of the replicated data type as per the specification is accepted by our global reference implementation.

Lemma 147. *Suppose $(\rho, \varphi) \in Runs(\mathcal{D}, f)$. Let $\rho' \in (\Sigma(\mathcal{D}) \times ID)^*$ be a well-formed annotated run whose associated run is (ρ, φ). Then ρ' is accepted by $\mathcal{D}_{ref}$.*

Proof. We prove the lemma for $\rho'[1:i]$, by induction on i. The base case, when $i = 0$ is trivial. So let $i > 0$. Suppose $\rho'[1:i-1]$ is accepted by $\mathcal{D}_{ref}$ by an execution ending in configuration C. Let (σ, φ) and (σ', φ) be the abstract runs associated with $\rho'[1:i-1]$ and $\rho'[1:i]$ respectively. Let $T = \mathcal{T}(\sigma, \varphi)$ and $T' = \mathcal{T}(\sigma', \varphi)$. Let $o = \rho'[i] = ((m, r, args, ret), id)$. There are three cases to consider.

$m \in \mathsf{Queries}$ **or** $m =$ **usend:** In this case $\partial_T(r) = \partial_{T'}(r)$. We know that

$$ret = Spec_{\mathcal{D}}(\partial_{T'}(r), m, args) = Spec_{\mathcal{D}}(\partial_T(r), m, args).$$

But we also know that $\mathsf{RelevantCtxt}(\partial_C(r), m, args)$ is isomorphic to

$$\mathsf{RelevantCtxt}(\partial_r(t), m, args).$$

Thus it follows that $ret = Spec_{\mathcal{D}}(\partial_C(r), m, args)$. Hence $C \xrightarrow{o} C'$ and $\rho'[1:i]$ is accepted by $\mathcal{D}_{ref}$.

$m \in \mathsf{Updates}$**:** Since $\rho'[1:i]$ is well-formed, it has to be the case that either id is not used in $\rho'[1:i-1]$, or if it is used in an update operation $\rho'[j]$, every replica has received that update in $\rho'[j+1:i-1]$. Thus, there is no node $v_{net} \in C_{net}$ with $Id(v_{net}) = id$. So, o is enabled at C and $\rho[1:i]$ is accepted by $\mathcal{D}_{ref}$.

$m =$ **mreceive:** Since $\rho'[1:i]$ is well-formed, it has to be the case that there is an earlier update at some other replica with the same identifier that has not yet been communicated to r. Thus there exists a node v and $R \subseteq \mathcal{R}$ such that $(v, id, R) \in C_{net}$ and $r \notin R$. It follows that o is enabled at C and $\rho'[1:i]$ is accepted by $\mathcal{D}_{ref}$.

□

From the previous two lemmas we can conclude the following:

Theorem 148. $Runs(\mathcal{D}_{ref}) = Runs(\mathcal{D}, f)$

Thus, the reference implementation for $\mathcal{D}$ thus provided covers all the runs that are valid as per the specification $Spec_{\mathcal{D}}$

6.1.4 Bounding the reference implementation

For effective verification, we need to ensure that the set of traces of the replicated data type has a finite representation. The reference implementation constructed in the previous section is not necessarily finite-state. The unboundedness arises due to the following reasons.

- If the size of the universe is not bounded, the number of nodes, and hence the number of configurations, will not be bounded.
- If there is no bound on the number of undelivered messages, then the number of network states would be unbounded, and therefore the size of C_{net} of any configuration C is unbounded.
- If the specification of the replicated itself is not a bounded specification, then the number of relevant nodes in the configuration is unbounded, even when the universe Univ is finite.

With some reasonable assumptions, we can ensure that the reference implementation is finite-state.

1. **Universe Size:** We assume that the size of the universe is bounded by a parameter N_{Univ}. This is a reasonable assumption since most replicated data-type implementations treat the elements of the universe in a uniform manner. Hence for the purpose of verification, it suffices to consider a universe whose size is bounded.
2. **Delivery Constraints:** We assume that the number of undelivered messages in the network is bounded by the parameter B. Again, this is a reasonable assumption since most practical implementations of strong eventual consistency also requires that messages are reliably delivered to all the replicas. We can pick a sufficiently large B that correctly characterizes the network guarantee of the actual implementation.

3. **Bounded Specification:** We assume that the specification function $Spec_{\mathcal{D}}$ computable via $\mathsf{RelevantCtxt}$ and $\mathsf{ComputeRet}$ comes with a bound M. Let k be the maximum arity of any $q \in \mathsf{Queries}$. If the universe if bounded, the number of query instances is bounded by $|\mathsf{Queries}| \times N_{\mathsf{Univ}}^k$. Since the specification function has a bound M, the size of the relevant nodes in a configuration is bounded by $\ell = M \times |\mathsf{Queries}| \times N_{\mathsf{Univ}}^k$.

   For example, in case of OR-sets where the updates are causally delivered, to answer the query $\mathsf{contains}(x)$ it is sufficient to keep track of the maximal x-events. Since the number of replicas $\mathcal{R}$ is bounded by N the number of maximal x-events is bounded by N. Hence if the universe is bounded by N_{Univ} then the number of relevant nodes in a configuration is no more than $N_{\mathsf{Univ}} \times N$.

We now prove that, with these assumptions, the size of the reference implementation is bounded. Each configuration of $\mathcal{D}_{ref}$ consists of an LAR for each replica, and a set of network nodes. As is clear from the transition rules, the only network nodes we retain are those that are still undelivered to some replicas. Thus, if there is a bound on the number of undelivered messages, there is also a bound on the number of network nodes present in each configuration. But the set of network nodes that occur in all configurations might still be unbounded. To bound this, we need to bound the set of all nodes and the set ID. The size of the set ID can be bounded by B, the number of undelivered messages, as explained below.

Let C be a reachable configuration of $\mathcal{D}_{ref}$ and o an update operation enabled at C. Now it has to be the case that only if there are at most $B-1$ network nodes in C_{net} (otherwise, there would be more than B undelivered messages in the run upto and including o). Thus as long ID has B elements, the reference implementation can always attach a fresh timestamp to o. (Formally this means that we can map any annotated run of $\mathcal{D}_{ref}$ to an equivalent run which uses at most B timestamps.)

We now turn to bounding the set of all nodes. The only unbounded component in this is the set $\mathcal{L}$ of labels.

Lemma 149. *If the number of undelivered messages is bounded by B and the number of relevant events is bounded by K then it is sufficient to have a label set $\mathcal{L}$ of size $B+K$*

Proof. Let $\rho = \rho'.o$ be any run of the reference implementation such that the number of undelivered messages in ρ is bounded by B. Let o be an update operation at replica r. Let C' be the configuration of the reference implementation at the end of ρ'.

Note that the number of undelivered update operations in ρ' is strictly less than B; otherwise, ρ would have more than B undelivered messages. It follows that the number of undelivered nodes in C' is at most $B-1$. (A node v is undelivered in C' if $(v, R) \in C'_{net}$ for some $R \subsetneq \mathcal{R}$.) A node v is present in some LAR $C'[r']$ if v is undelivered or v is relevant. Thus the number of distinct nodes in C' is at most $B+K-1$. Thus if $|\mathcal{L}| = B+K$, there

is at least one free label in $\mathcal{L}$ to label the new node $C[r] \setminus C'[r]$. Thus, it is sufficient to have a label set $\mathcal{L}$ of size $B + K$. □

From the above, we can conclude that the number of nodes in $\mathcal{N}$ is bounded by $|\mathsf{Updates}| \times N \times N_{\mathsf{Univ}}^{k'} \times (B + K)$ (where, as before, k' is the maximum arity of any $u \in \mathsf{Updates}$).

Since the set ID is also bounded (by B, as already explained), the set of network nodes is bounded (by $|\mathcal{N}| \times |ID| \times 2^N$).

From Lemma 149 it is clear that the number of distinct nodes in any configuration cannot exceed $B + K$. Since the number of undelivered messages are bounded by B, the number of network nodes is bounded by B. Thus, the set of all configurations $\mathcal{C}$ is bounded as follows:

$$|\mathcal{C}| \leq |\mathcal{N}|^{(B+K)} \times |\mathcal{N}_{net}|^B.$$

Theorem 150. *If the number of undelivered messages and the size of the universe are bounded and we have a bounded specification for the replicated data-type, then the reference implementation is bounded.*

6.2 Applications to verification

A bounded reference implementation of a replicated data type can be used for to formally verify the correctness of a given implementation of replicated data type. It can also be used to design effective test suites to uncover the bugs in a given implementation. We discuss this in the current section.

6.2.1 Effective verification using Bounded Reference Implementation via CEGAR

Verifying implementations of replicated data types is a challenging task. For instance, consider an implementation that uses a bounded set of timestamps as we have proposed, except that the size of this set is too small. Under certain circumstances, a replica may be forced to reuse a timestamp even when a previous update with the same timestamp has not been delivered. To detect such an error, we have to explore a run that exceeds the bound in the implementation. Unfortunately, we typically do not have access to the internal details of the implementation, so this bound is not known in advance. This results in an unbounded verification task.

Alternatively, we have seen that by making reasonable restrictions on the universe of the datatype and the behaviour of the underlying messasge delivery system, we can generate a bounded reference implementation. Once we have such a bounded reference implementation, we can use Counter Example Guided Abstract Refinement (CEGAR) [Clarke et al., 2003] to

effectively verify a given CRDT implementation with respect to the assumptions made on the environment.

Counterexample Guided Abstraction Refinement, or CEGAR, is an iterative technique to verify reachability properties of software systems [Clarke et al., 2003]. In the CEGAR approach, one uses abstraction techniques from program analysis and other domains to build a finite-state abstraction of a given implementation. This abstraction is designed to over-approximate the behaviour of the original system.

The finite-state approximation is run through a model-checker to verify if the safety property is met. If no unsafe state is reachable, it means that the original system is safe since the abstracted system over-approximates the actual behaviour. On the other hand, if the model-checker asserts that an unsafe state is reachable, the counterexample generated by the model-checker is executed on the original system. If the counterexample is valid, a bug has been found. If the counterexample is infeasible, the abstraction was too coarse and a refinement of the abstraction is calculated. This process is iterated until a safe abstraction is reached or a valid bug is detected.

In the case of replicated data types, as we have noted above, a given implementation would need to keep track of metadata about past operations in order to reconcile conflicts. These are typically done using unbounded objects such as counters or vector clocks. To apply CEGAR to such an implementation, we can derive a finite state abstraction and run it synchronously with our bounded reference implementation. Thus our bounded reference implementation acts as a model checker. We can characterize each reachable state of the finite state abstraction of the given implementation as legal or illegal depending on whether or not it is query equivalent to the reference implementation.

More formally, given a implementation of a replicated data type with bounded specification, let us assume suitable bounds on the size of the universe, N_{Univ}, and the number of undelivered messages, B. We fix the bounded set of timestamps ID accordingly. We assume the existence of an abstraction function that provides a finite state abstraction $\mathcal{D}_I = (\mathcal{C}_I, C_{I\perp}, ID, \rightarrow_I)$ of the implementation, whose runs are in $(\Sigma(D) \times ID)^*$.

We then construct the synchronous product $\mathcal{M}_{sync} =$
$((\mathcal{C}_I \times \mathcal{C}_{ref}) \cup \{C_{err}\}, (C_{I,\perp}, C_{ref\perp}), ID, \rightarrow_{sync})$, where $\rightarrow_{sync}$ is defined as follows:

- The action $o \in \Sigma(\mathcal{D}) \times ID$ is enabled at the product state (C_I, C_{ref}) iff o is enabled at C_I in $\mathcal{D}_I$. If o is enabled then we define
  - $(C_I, C_{ref}) \xrightarrow{o}_{sync} s_{err}$, if o is not enabled at C_{ref} in $\mathcal{D}_{ref}$
  - $(C_I, C_{ref}) \xrightarrow{o}_{sync} (C'_I, C'_{ref})$, if $C_I \xrightarrow{o}_I C'_I$ and $C_{ref} \xrightarrow{o}_{ref} C'_{ref}$.
- $\forall o \in \Sigma_{\mathcal{D}} : o$ is not enabled at s_{err}

From this construction, we can conclude the following.

Lemma 151. *If ρ is a run of $\mathcal{M}_{sync}$ resulting in the state s_{err} starting from the initial state $(C_{I\perp}, C_{ref\perp})$, then $\rho \in Runs(\mathcal{D}_I) \setminus Runs(\mathcal{D}, Spec_{\mathcal{D}})$.*

Thus any run ρ leading to the state s_{err} in the synchronous product is a potential counter example. As usual, we can use the finite abstraction to try trace an actual run in original implementation corresponding to ρ. If we succeed in finding such a run, we have found a bug in the original implementation. If the abstract counterexample turns out to be infeasible, then we refine our abstraction using the feedback obtained from our failure to construct a valid run. We repeat this process until a bug is found or we are satisfied with the level of abstraction to which we have verified the system.

6.2.2 Testing of distributed systems

While verification approaches such as CEGAR can be used in a white box setting where we have access to internal details of the implemenation under test, in a black box scenario we have to rely on testing.

Effective testing of distributed systems is a challenging task. The first problem is that we cannot typically test the system globally, so we have to apply tests locally using notations such as TTCN [Willcock et al., 2005]. Even when such a methodology is available, there are two criteria that are difficult to establish for test suites: coverage and redundancy. In both cases, the main source of complexity is the presence of concurrency. In a concurrent system, it is very difficult to estimate if a test suite covers a reasonable set of reachable global states because of many different linearizations possible. Secondly, it is not obvious to what extent tests are overlapping, again because of reordering of independent events.

For the coverage problem, we can construct test suites that cover different portions of the state space of the reference implementation. If this coverage is widespread, we can have more confidence in the coverage of the implementation under test. For redundancy, once again we can use the underlying independence relation to identify when two tests overlap by checking how much the corresponding traces overlap as partial orders. In the replicated data type scenario, we may in fact want to generate redundant test cases that differ only in the order of concurrent events in order to validate strong eventual consistency.

6.3 Summary and Related Work

In this chapter, we have shown how to construct a reference implementation for a CRDT that is described using a bounded declarative specification. By imposing reasonable constraints on the universe of the datatype and the underlying message delivery subsystem, the reference implementation can be made finite-state. This can be exploited to verify any given implementation using CEGAR.

The key observation in this chapter is that a global reference implementation suffices for verification. This greatly simplifies the construction compared to the distributed reference implementation from [Mukund et al., 2015a] described in the previous chapter, which requires an intricate distributed timestamping procedure due to the local nature of the information available at each replica.

The other interesting feature of our reference implementation is that the basic construction using LARs is independent of the assumptions that we make on the set of data values and the nature of message delivery in order to bound the set of timestamps used. Thus, the reference implementation relies only on the declarative specificaton of the replicated data type. We can then separately reason about the size of this implementation under various constraints on the operating environment.

We now look at some of the related work in the area of formal specification and verification of Replicated Data Types. [Burkhardt et al., 2014] was the first work to propose a framework for specifying replicated data types using relation over events. This framework if further refined in the extended work by one of the authors [Burckhardt, 2014]. In this thesis we have adopted a simplified version of this framework for providing the specifications of replicated data types. The verification strategy followed in [Burkhardt et al., 2014] is replication-aware simulations whose states can be unbounded. In constrast, our approach in this thesis has been to synthesize distributed as well as global bounded reference implementations which can be used to verify the correctness of replicated data types with the aid of CEGAR.

[Zeller et al., 2014] provide a formal framework for the analysis and verification of CRDTs. They follow the specification framework from [Bouajjani et al., 2014a] and [Burkhardt et al., 2014] by altering it for the purposes of specifying CRDTs. They encode this framework using the Isabelle/HOL proof assistant and provide the proof of correctness for a number of CRDT implementations. The verification strategy differs from our work in they use a theorem proving approach while we rely on concepts from automata theoryto verify the correctness of replicated data types.

Like [Zeller et al., 2014], [Gomes et al., 2017] focusses on verifying the correctness correctness of Conflict-free Replicated Data Types (CRDTs) through a modular framework in Isabelle/HOL interactive proof assistants for verifying the correctness of CRDT algorithms. They include the network model in our formalization, thereby having the ability to show the correctness of the CRDT algorithms under various network behaviours. They provide machine checked proofs for the correctness of Replicated Growable Array, the Observed-Remove Set, and the PN Counter.

A recent work by [Blau, 2020] describes how δ-state CRDTs retain the advantages of both state-based CRDTs as well as operation-based CRDTs, while being equivalent with the latter two. The paper formalizes the intuition through a pair of reductions between the three kinds of CRDTs. They demonstrate how state based CRDTs satisfy strong eventual consistency using the Isabelle/HOL interactive proof assistant. Furthermore they show that

even δ-CRDTs maintain SEC when only communicating δ-state fragments. This strategy is targetted toward δ-state CRDTs which is beyond the scope of this thesis.

With this, we come to an end of our study of the behaviours of the replicated data types from the perspective of the replicas. In the next chapter we focus our attention towards the investigation of the behaviours of the replicated data stores as seen by the end-users or the clients.

7

Formalizing and Checking Multilevel Consistency

7.1 Motivation

In the earlier chapters, we have studied the behaviours of replicated implementations of abstract data types using declarative specifications (Chapter 4) and bounded reference implementations (Chapters 5 and 6). As mentioned in Chapter 1, the behaviours of the replicated implementations are observed from the perspective of the communicating replicas. These replicas maintain data and additional metadata in order to ensure that they correctly model the abstract data type as per the specification and in order to satisfy the strong eventual consistency criterion. Furthermore, they communicate with other replicas by sharing suitable fragments of their data and metadata so that eventually all the replicas converge to equivalent states. Since the behaviours explicitly encoded the communications between replicas through update-send/receive and merge-send/receive operations, we could derive the visibility relation (Definition 78 in Chapter 4) as a combination of the replica order, the update-send/receive order and the merge order. This visibility relation allowed us to determine the causal past of the replicas at any point in time, thereby allowing us to reason about the state of the replicas, and thereby the query-responses. However, the users or the clients interacting with these replicated data types are oblivious to implementation details such as the number of replicas involved, the data and the metadata that they maintain, the frequency at which they communicate and the constraints on message delivery. Thus, the behaviours of the replicated data types as seen by the individual clients are sequences of update and query method invocations and the corresponding responses provided by the data type implementation. Since there can be multiple clients concurrently interacting with the replicated data type, the update requests issued by some clients will influence the responses provided to the other clients. Thus, reasoning about the correctness of the query responses for a client requires us to identify the update requests from other clients. Since

the behaviours observed by the clients do not reveal information as to these updates are ordered by the backend replicated implementation, we need a different strategy to test the correctness of these behaviours.

Furthermore, the *strong eventual consistency* criterion ensures that replicas that have received the same set of update operations should be query-equivalent. The guarantees provided by this consistency criterion is only meaningful from the perspective of the replicas and not the clients, since since the clients are not aware of if they are interacting with a pair of replicas that have received the same updates or not. Thus, the modern replicated data stores often provide other consistency guarantees such as eventual consistency [Terry et al., 1995] or causal consistency [Lamport, 1979], which can be reasoned about from the behaviours observed at the client side. Despite them being around for a while, many of them have been formalized only recently [Bouajjani et al., 2014b; Burkhardt et al., 2014; Perrin et al., 2016; Bouajjani et al., 2017]. In this chapter we turn our attention towards modelling and testing the correctness of the behaviours of replicated data stores, as observed by the clients.

From the perspective of the clients, programming applications on top of weakly-consistent data-stores is difficult. Some form of synchronization is often unavoidable to preserve correctness. Therefore, popular data-stores such as Amazon DynamoDB and Apache's Cassandra provide different levels of consistencies, ranging from weaker forms to strong consistency. Applications can tag queries to the data-store with a suitable level of consistency depending on their needs.

Implementations of large-scale data-stores are difficult to build and test. For instance, they must account for partial failures, where some components or the network can fail and produce incomplete results. Ensuring fault-tolerance relies on intricate protocols which are difficult to design and reason about. The black-box testing framework Jepsen[1] found a remarkably large number of subtle problems in many production distributed data-stores.

Testing a data-store raises two issues: (1) deriving a suitable set of testing scenarios, e.g., faults to inject into the system and the set of operations to be executed, and (2) efficient algorithms for checking whether a given execution satisfies the considered consistency models. The Jepsen framework shows that the first issue can be solved using randomization, e.g., introducing faults at random and choosing the operations randomly. The effectiveness of this solution has been proved formally in recent work [Ozkan et al., 2018]. The second issue is dependent on a suitable formalization of the consistency models.

In this chapter, we consider the problem of specifying data-stores which provide multiple levels of consistency and derive algorithms to check whether a given execution adheres to such a multilevel consistency specification.

We build on the specification framework in [Burkhardt et al., 2014], presented earlier in

[1]Available at `http://jepsen.io`

chapter 4, which formalizes consistency models using two auxiliary relations: (i) a *visibility* relation, which specifies the set of operations observed by each operation, and (ii) an *arbitration order*, which specifies the order in which concurrent operations should be viewed by all replicas. An execution is said to satisfy a consistency criterion if there exists a visibility relation and an arbitration order that obey an associated set of constraints [Burckhardt, 2014]. For the case of a data-store providing multiple levels of consistency, we consider multiple visibility relations and arbitration orders, one for each level of consistency. Then, we consider a set of formulas which specifies each consistency level in isolation, and also, how visibility relations and arbitration orders of different consistency levels are related.

Based on this formalization, we investigate the problem of checking whether a given execution satisfies a certain multilevel consistency specification. In general, this problem is known to be NP-COMPLETE [Bouajjani et al., 2017]. However, for a certain class of data stores known as *data-independent* stores [Wolper, 1986] we show that this problem is PTIME for many practically-interesting multilevel consistency specifications. Since practical data-store implementations are data-independent, i.e., their behaviour doesn't depend on the concrete values read or written in the transactions, it suffices to consider executions where each value is written at most once. This complexity result uses the idea of *bad patterns* introduced in [Bouajjani et al., 2017] for the case of causal consistency. Intuitively, a bad pattern is a set of operations occurring in a particular order corresponding to a consistency violation. In this chapter, we provide a *systematic methodology* for deriving bad patterns characterizing a wide range of consistency models and combinations thereof.

Our contributions form an effective algorithmic framework for the verification of modern data-stores providing multiple levels of consistency. To the best of our knowledge, we are the first to investigate the asymptotic complexity for such a wide class of consistency models and their combinations, despite their prevalence in practice. This chapter is based on our work [Bouajjani et al., 2020].

7.2 Multilevel consistency in the wild

In this section we present some real-world instances of multilevel consistency. We restrict our attention to distributed read-write key-value data-stores (henceforth referred to as read-write stores), consisting of unique memory locations addressed by *keys* or *variables*. We use *keys* and *variables* interchangeably in this work. The contents of these memory locations come from a domain, called *values*.

The read-write data-store provides two APIs to access and modify the contents of a particular memory location. The API to read the content of a particular memory location is typically named *Read* or *Get*, and the API to store a value into a particular memory location is typically named *Write* or *Put*. In this chapter, we refer to these two methods as Read and

Write respectively. The Read method does not update the state of the data-store and only reveals part of the state to the application session which invokes the method. The Write method on the other hand modifies the state of the data-store.

Typically, an application reads a location of the data-store, performs some local computation and writes a value back to the data-store to the same or some other location. A sequence of related read and write operations performed by an application is called a *session*.

Applications expect some sort of consistency guarantee from the data-store in terms of how *fresh* or *stale* the data value is that they read from the data-store. They also seek some guarantees pertaining to monotonicity of the results that are presented to them. These guarantees provided by the data-store to the applications are called *consistency criterion*. Some of the popular consistency criteria include *Read-Your-Writes*, *Monotonic Reads*, *Monotonic Writes*, *Causal Consistency*, *Sequential Consistency* which were introduced with examples in Chapter 1.

- **Read-Your-Writes**: The effects of prior operations in the session will be visible to later operations in the same session.
- **Monotonic Reads**: Once the effect of an operation becomes visible within a session, it remains visible to all subsequent operations in that session.
- **Monotonic Writes**: If the effect of a remote operation is visible in a session, then the effects of all prior operations in the session of the remote operation are also visible.
- **Causal consistency**: Effects of prior operations in a session are always visible to later operations. Further, if the effect of an operation is visible to another operation, then every operation that has seen the effects of the latter would have seen the effects of the former.
- **Sequential Consistency**: Effects of the operations can be explained from a single sequential execution obtained by interleaving the reads and writes performed at individual sessions.

Most of the existing literature on testing the behaviour of read-write stores focuses on testing the correctness with respect to specific consistency criteria [Bouajjani et al., 2014b; Furbach et al., 2014; Bouajjani et al., 2017]. However, there are cases where data-stores such as DynamoDB and Cassandra offer to applications the choice of specifying the consistency level per read-operation [Damien., 2018]. There are distributed data-store libraries that allow consistency rationing [Kraska et al., 2009] and also allow incremental consistency guarantees for the read operations [Guerraoui et al., 2016]. In each of these cases we need to reason about the correctness of the behaviour of the data-store with respect to more than one consistency criterion.

We now look at some examples of multilevel consistency in the real world. In this work, we assume that the Read and the Write APIs are as follows.

Definition 152 (Read and Write APIs)**.** *Let x be a key/variable, val denote a value read-from/written-to the data-store and level denote the consistency level.*

- Write(x, val) *: Updates the content of the memory location addressed by the key/variable x with the value val.*
- Read$(x, val, level)$ *: This says that the content of the memory location associated with the variable x is val with respect to the consistency level level.*

Read-Write Stores with strong and weak reads

Consider the case of the data-store Cassandra, which allows the application a more fine grained choice of consistency levels, such as `ANY`, `ONE`, `QUORUM`, `ALL`. It achieves this by ensuring that when the Read is executed with `ANY`, the return value is provided by consulting any available replica of the data store. Similarly, if the Read operation is submitted with `ONE`, the return value is provided by consulting a replica that is known to contain at least one value for that key. On the other hand, if the Read is executed with `QUORUM`, the data-store returns the value after consulting a majority of the replicas. Finally, if Read is executed with `ALL`, then all the replicas are consulted before returning the response. Clearly, `ANY` is the weakest consistency criterion while `ALL` is the strongest consistency criterion. In general, a data-store offers responses pertaining to different consistency criteria by consulting the required subset of replicas to answer the query.

Typically a read operation under the stronger consistency criterion will take more time, since it might have to wait for all pending operations to become visible, or run a consensus protocol before returning the result. In certain cases, applications may be satisfied with Read operations that return values that are correct with respect to some weaker consistency criterion. Consider a web-application that displays the available seats in a movie theater. The application can choose to read the available seats based on a weaker consistency criterion, since:

- The number of users attempting to book seats is usually more than the seats available. Waiting for a consensus or a quorum can slow down the reads for everyone. So a quicker response is desirable.
- There is a lag between the time the user gets to see available seats and the time when the user decides to book particular seats. Since concurrent bookings are ongoing, the data displayed can become stale by the time the user books the seat.

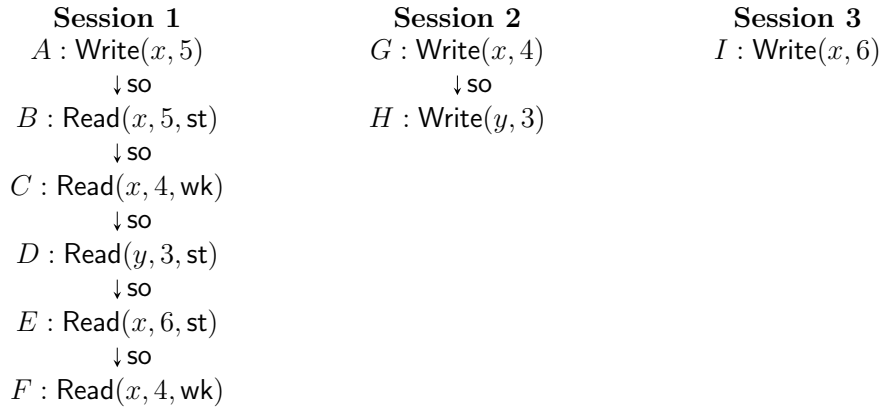


Figure 7.1: An example of a read-write store behaviour with strong and weak reads. The so relation relates read and write operations from the same session in the order in which they happened in that session.

- Users can change their minds before finally settling on a set of seats, and paying for them.

Thus, the web-application can opt for a read satisfying a weaker consistency criterion while allowing the user to pick a seat, and then perform a read satisfying a stronger consistency criterion only when the user pays for it.

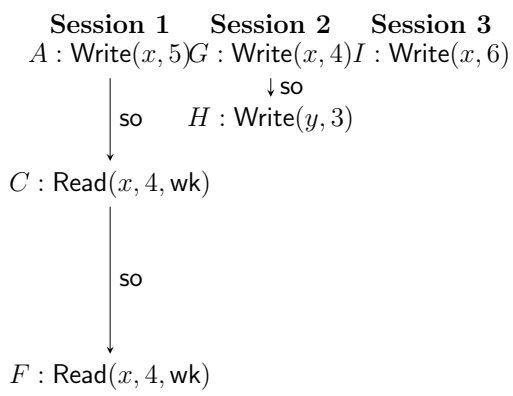
Consider the example in Figure 7.1 where all write requests are processed at the same replica. For each session, there is a (potentially different) designated replica from which the responses to the weak reads are returned.

In this scenario, the strong reads (corresponding to the consistency level `ALL`) satisfy sequential consistency while the weak reads obey monotonic reads consistency. Hence, the fragment consisting of all the writes and the weak reads should be correct with respect to monotonic reads. Similarly, the fragment consisting of all the writes and the strong reads should be correct with respect to sequential consistency.

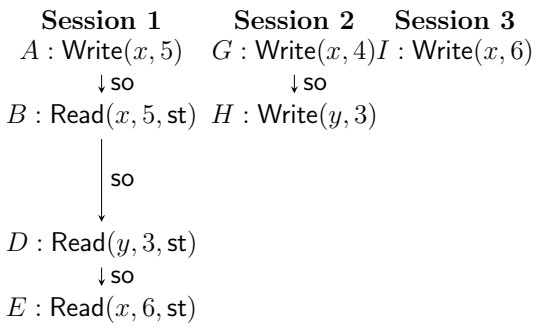
The weak fragment corresponding to the example in Figure 7.1 can be seen in Figure 7.2(a). This fragment is correct with respect to monotonic reads; once the write G is visible at session 1 to the read C, it remains visible throughout the session. The write I is not visible to any of the other sessions yet.

The strong fragment is represented in Figure 7.2(b). This is correct with respect to sequential consistency, where the order of the operations obtained by consensus is $A \rightarrow B \rightarrow G \rightarrow H \rightarrow I \rightarrow D \rightarrow E$.

However, since the strong reads correspond to the level `ALL` where all the replicas have seen the prior writes and have agreed on the order of the concurrent writes, it behooves a weak read following a strong read to take into consideration the effects seen by the earlier



(a) Weak Fragment from Figure 7.1



(b) Strong Fragment from Figure 7.1

Figure 7.2: Strong and Weak fragments of the hybrid behaviour

strong read. Thus, the data-store imposes an additional constraint. Once a write is visible to a strong read in a session, it is visible to all the subsequent weak reads in that session. This ensures that the weaker reads do incorporate the prior results seen by the session. Similarly, a write visible to a weak read is made from a replica which participates in the subsequent strong reads corresponding to the level `ALL`. Thus, the effects visible to the prior weak reads in a session are also visible to the subsequent strong reads.

With these additional constraints, we can no longer explain the read operation F, since the effects of writes G and I are both visible at read F. The strong consistency criterion has already guaranteed that write I has happened after write G, thereby effectively overwriting the value 4 with the value 6. Hence this behaviour is incorrect in the multilevel setting.

Now consider the behaviour of Cassandra where writes are performed at one of the replicas (corresponds to the level `ONE`), weak reads are performed at one of the replicas (corresponds to the level `ONE`) and strong reads are performed at a quorum of replicas (corresponds to the level `QUORUM`). In this situation, it is not necessary that the effects of writes visible to prior weaker reads are visible at subsequent stronger reads, since the replica from which the weaker read is performed may be missing from the quorum of replicas from which the stronger read is made. Similarly, the effects of writes visible to prior stronger reads of a session need not be visible to the subsequent weaker reads in the session, as the writes from the quorum may not have reached the replica from which the weaker read is performed. Thus, the stronger and weaker reads can be independent of each other.

Finally consider the case of Amazon DynamoDB Accelerator (DAX) [Documentation, 2019], which contains a write-through cache sitting between the application and the DynamoDB backend. Every write made by the application is first submitted to the DynamoDB backend and also updated at the cache. By default, the reads are eventually consistent, i.e., the reads are performed from the cache. If the item does not exist in the cache, then it is

fetched from the backend data-store and the cache is updated with the item before the value is returned to the application. However, the application can also request strongly consistent reads by invoking `ConsistentRead`. In this case, the value is read from the backend and returned to the application, without caching the results. Any subsequent eventually consistent reads made by the application may not reflect the value returned by the prior strongly consistent read. In the case of DAX, it can be observed that the effects of the writes visible to the weak eventually consistent reads are also visible to the subsequent strongly consistent reads as those writes are also present in the DynamoDB backend. However, it is not necessary that the effects of writes visible to the strongly consistent reads are visible to the subsequent weak eventually consistent reads.

From these examples of multilevel consistency, we can see that the presence of another consistency criterion can impose additional constraints on the choice of the visibility and arbitration relations chosen to explain the correctness of the history. In the next section, we provide a formal framework for modelling behaviours of read-write data-stores with multiple consistency levels.

7.3 Formalizing Multilevel Consistency

We extend the formal framework provided in [Burckhardt, 2014] for modelling the behaviours of read-write stores. Each operation submitted to the data-store by the application is either a Read or a Write operation whose signature is given in Definition 152.

We denote the set of all variables in the read-write store by *Vars* and assume that each value written to the read-write store is a natural number $val \in \mathbb{N}$. We assume that all variables are initially undefined, with value $\perp$.

For simplicity, we assume only two consistency levels, weak and strong, denoted by wk and st, respectively, where the consistency criterion corresponding to wk-level is strictly weaker than then the consistency criterion corresponding to the st-level. Comparison between consistency criteria is formally defined in Definition 158.

The behaviour of the read-write data-store as observed by an application is the sequence of reads and writes that it performs on the stores. The sequence of related read and write operations is termed a *session*. Thus, the behaviour of the read-write store seen by each session is a total order of read/write operations performed in that session.

The behaviour of the read-write store is the collection of behaviours seen by all the sessions. In Figure 7.1 we saw the behaviour of the data-store as observed by the three sessions accessing the data-store. We call such a behaviour a *hybrid history*, formally defined as follows:

Definition 153 (Hybrid History)**.** *A hybrid history of a read-write store is a pair* $H = (\mathcal{O}, \mathsf{so})$ *where* $\mathcal{O}$ *is the set of read-write operations and* so *is a collection of total orders called*

session orders.

For a history H, we define the following subsets of $\mathcal{O}$.

- $\mathcal{O}_{\mathsf{Read}}$ *is the set of read operations occurring in H.*
- $\mathcal{O}_{\mathsf{Write}}$ *is the set of write operations occurring in H.*
- $\mathcal{O}_{\mathsf{wk}} = \mathcal{O}_{\mathsf{Write}} \cup \{\mathsf{Read}(x, val, level) \in \mathcal{O}_{\mathsf{Read}} \mid level = \mathsf{wk}\}$ *(the set of weak operations occurring in H).*
- $\mathcal{O}_{\mathsf{st}} = \mathcal{O}_{\mathsf{Write}} \cup \{\mathsf{Read}(x, val, level) \in \mathcal{O}_{\mathsf{Read}} \mid level = \mathsf{st}\}$ *(the set of strong operations occurring in H).*

The weak fragment of the history H is denoted H_{wk} and defined to be $(\mathcal{O}_{\mathsf{wk}}, \mathsf{so} \cap (\mathcal{O}_{\mathsf{wk}} \times \mathcal{O}_{\mathsf{wk}}))$. Similarly the strong fragment of the history H is denoted H_{st} and is defined to be $(\mathcal{O}_{\mathsf{st}}, \mathsf{so} \cap (\mathcal{O}_{\mathsf{st}} \times \mathcal{O}_{\mathsf{st}}))$. Note that we take the write operations to be part of both the strong and weak fragments.

- For $X \subseteq \mathcal{O} \times \mathcal{O}$ and $\ell \in \{\mathsf{Read}, \mathsf{Write}, \mathsf{wk}, \mathsf{st}\}$, we denote by $X \restriction_\ell$ the set X restricted to $\mathcal{O}_\ell$. Thus,
$$X \restriction_\ell = X \cap (\mathcal{O}_\ell \times \mathcal{O}_\ell)$$
- For $X, Y \subseteq \mathcal{O} \times \mathcal{O}$, $X;Y$ denotes composition of X and Y, i.e., $\{(x, y) \mid \exists z : (x, z) \in X \text{ and } (z, y) \in Y\}$.
- For $X \subseteq \mathcal{O} \times \mathcal{O}$, $\mathsf{total}(X)$ is used to mean that X is a total order.

When a replica of the read-write store receives an operation from an application, it decides how the effects of the older operations known to the replica (either received from applications, or from other replicas of the data-store) should be made visible to the new operation. A visibility relation over a history specifies the set of operations visible to an operation. This is analogous to the visibility order over the events of the trace of a run that we had defined in Chapter 4. However, in this chapter, the visibility relation is defined on Read and Write operations of a hybrid history.

Definition 154 (Visibility Relation)**.** *A visibility relation* vis *over a history $H = (\mathcal{O}, \mathsf{so})$ is an acyclic relation over $\mathcal{O}$. For $o, o' \in \mathcal{O}$, we write $o \xrightarrow{\mathsf{vis}} o'$ to indicate that the effects of the operation o are visible to the operation o'.*

If a pair of operations o, o' are not related by vis*, we term them concurrent operations, denoted by $o \parallel_{\mathsf{vis}} o'$.*

We define the view *of an operation o with respect to a visibility relation* vis*, denoted $\partial_{\mathsf{vis}}(o)$ to be the set of all the* Write *operations visible to it.*

For the history in Figure 7.1, we can define a visibility relation to be

$$\{A \xrightarrow{\mathsf{vis}} B, G \xrightarrow{\mathsf{vis}} C, G \xrightarrow{\mathsf{vis}} D, H \xrightarrow{\mathsf{vis}} D, G \xrightarrow{\mathsf{vis}} E, H \xrightarrow{\mathsf{vis}} E, I \xrightarrow{\mathsf{vis}} E, G \xrightarrow{\mathsf{vis}} F\}$$

When the replicas communicate with each other, they need to reconcile the effects of concurrent write operations in order to converge to the same state eventually. In case of convergent data-stores this is done using a rule such as *Last Writer Wins* which totally orders all write operations. This is abstracted by an arbitration relation, which is a total order over all write operations in the history. We will denote the arbitration relation by arb. We assume that the arbitration relation is consistent with the visibility relation, in the sense that for a pair of writes o and o', if o is visible to o' then o is before o' in arb.

Definition 155 (Arbitration Relation)**.** *An arbitration relation* arb *over a hybrid history* $H = (\mathcal{O}, \mathsf{so})$ *is a total order over* $\mathcal{O}_{\mathsf{Write}}$*. For* $o_i, o_j \in \mathcal{O}$*, we say* $o_i \xrightarrow{\mathsf{arb}} o_j$ *to indicate that operation* o_i *has been ordered before the operation operation* o_j*.*

For the history in Figure 7.1 the arbitration relation can be the total order

$$A \xrightarrow{\mathsf{arb}} G \xrightarrow{\mathsf{arb}} H \xrightarrow{\mathsf{arb}} I$$

We define the correctness of a hybrid history in terms of the *functional specification* of read-write stores.

Let H be a hybrid history. Let vis and arb be visibility and arbitration relations over H.

We say that a write operation o' is a *related-write* of a read operation o iff o' is in the view of o and both o and o' operate on the same variable. The set of all related writes of o, denoted as $RelWrites_{\mathsf{vis}}(o)$ is defined to be $\{o' \in \partial_{\mathsf{vis}}(o) \mid o \text{ and } o' \text{ operate on the same variable}\}$.

$MaxRelWrites_{\mathsf{vis}}(o)$, the set of maximal elements among these related writes with respect to vis, is defined to be

$$\{o' \in RelWrites_{\mathsf{vis}}(o) \mid \forall o'' \in RelWrites_{\mathsf{vis}}(o) : o'' \xrightarrow{\mathsf{vis}} o' \vee o'' \parallel_{\mathsf{vis}} o'\}$$

The effective write of a read-operation o, denoted by $EffWrite^{\mathsf{arb}}_{\mathsf{vis}}(o)$ is defined to be the maximum write operation from the set of maximal related writes of o as per the arbitration relation.

$$EffWrite^{\mathsf{arb}}_{\mathsf{vis}}(o) = \begin{cases} max(\mathsf{arb} \restriction_{MaxRelWrites_{\mathsf{vis}}(o)}) & \text{if } MaxRelWrites_{\mathsf{vis}}(o) \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$

Definition 156 (Functional Correctness for Read-Write Stores)**.** *Let* $H = (\mathcal{O}, \mathsf{so})$ *be a hybrid history of a read-write data store with visibility relation* vis *and arbitration relation* arb*. We say that* $(H, \mathsf{vis}, \mathsf{arb})$ *is* functionally correct *iff for every read operation* $o = \mathsf{Read}(x, val, level)$*, the following conditions hold.*

- *$EffWrite^{\mathsf{arb}}_{\mathsf{vis}}(o) = \bot$ iff $val = \bot$ (i.e., there was no write operation on x when o happened).*
- *If $o' = EffWrite^{\mathsf{arb}}_{\mathsf{vis}}(o)$ then o' wrote the value val.*

Next, we formally define *consistency criteria* in terms of a set of formulas. Our definition is adapted from the definitions of constraints in [Emmi and Enea, 2018].

Definition 157 (Consistency Criteria)**.** *A* relation term τ *is a composition of the form* $t_1; \cdots ; t_k$ *($k \geq 1$), where each* $t_i \in \{so, vis\}$*. A* consistency criterion *is a subset of*

$$\{\tau \subseteq vis \mid \tau \text{ is a relation term}\} \cup \{\mathsf{total}(vis)\}.$$

Thus, a consistency criterion is a possibly empty collection of visibility constraints and an optional totality constraint. For simplicity of notation, we usually write a constraint as a conjunction.

Note that so and vis are variables which are usually interpreted as restrictions of the so *and* vis *relations in a history. As we will see below, we always require an additional constraint that* $\mathsf{vis}\restriction_{\mathsf{Write}} \subseteq \mathsf{arb}$ *(and hence it is not explicitly included in the consistency criteria).*

For a consistency criterion α, $RelTerms(\alpha)$ is the set of all relation terms occurring in α, and $VisBasic(\alpha)$ is the collection of all visibility constraints in α excluding the totality constraint $\mathsf{total}(vis)$

Definition 158 (Consistency Criterion in a history)**.** *Let $H = (\mathcal{O}, \mathsf{so})$ be a hybrid history, let* vis *and* arb *be a visibility and arbitration relation over H, and let α be a consistency criterion. We say that $H, \mathsf{vis} \models \alpha$ iff:*

1. *for every $\tau \subseteq vis$ in α, $\tau[so := \mathsf{so}, vis := \mathsf{vis}] \subseteq \mathsf{vis}$, and*
2. *if $\mathsf{total}(vis) \in \alpha$, then $\mathsf{total}(\mathsf{vis})$ holds.*

Further we say that $H, \mathsf{vis}, \mathsf{arb} \models \alpha$ iff $H, \mathsf{vis} \models \alpha$ and $\mathsf{vis}\restriction_{\mathsf{Write}} \subseteq \mathsf{arb}$.

Some well known consistency criteria are given in Table 7.1. *Basic Eventual Consistency* requires that eventually, when no new Write operations are made, all the Read operations will eventually converge to the same value. This is more of a *liveness* property which is satisfied in the limit. Hence, there is no constraint on the visibility and the arbitration relation.

Read Your Writes requires that the effects of the Write operations made within a session are visible to the subsequent Read operations within the same session. Hence the *session-order* is contained within the *visibility relation*.

Monotonic Reads requires that once a remote Write becomes visible in a session, it remains visible for the subsequent operations of that session. This is modelled by the constraint

$vis; sovar \subseteq vis$. Thus, if $o_1 \xrightarrow{\mathsf{vis}} o_2$ and $o_2 \xrightarrow{\mathsf{so}} o_3$ then *monotonic reads* guarantees that $o_1 \xrightarrow{\mathsf{vis}} o_3$.

Monotonic Writes requires that once a remote Write becomes visible to some Read in a session, all the Write operations prior to that Write are visible to the Read operation. This is modelled by the constraint $so; vis \subseteq vis$. Thus, if $o_1 \xrightarrow{\mathsf{so}} o_2$ and $o_2 \xrightarrow{\mathsf{vis}} o_3$ then *monotonic writes* guarantees that $o_1 \xrightarrow{\mathsf{vis}} o_3$.

FIFO Consistency provides all the guarantees provided by *Read Your Writes*, *Monotonic Reads* and *Monotonic Writes*. Hence, all the constraints that apply to each of these consistency criteria should be simultaneously satisfied by a history which guarantees *FIFO consistency*.

Causal Consistency requires that the transitive closure of the session order and the visibility relation is contained in the visibility relation, i.e $(so \cup vis)^* \subseteq vis$. We rewrite the same constraints as $so \subseteq vis$ along with $vis; vis \subseteq vis$ in order to satisfy the syntax of a $\tau \subseteq vis$.

Sequential Consistency requires that the visibility relation be a total order of over all the operations where the total order is the interleaving of all the session orders. Thus, it is stronger than *Causal Consistency* while requiring the visibility relation to be a total order.

Name	Description
Basic Eventual Consistency (BEC)	$\top$
Read Your Writes (RYW)	$so \subseteq vis$
Monotonic Reads (MR)	$vis; so \subseteq vis$
Monotonic Writes (MW)	$so; vis \subseteq vis$
FIFO Consistency (FIFO)	$so \subseteq vis \ \wedge \ vis; so \subseteq vis \ \wedge \ so; vis \subseteq vis$
Causal Consistency (CC)	$so \subseteq vis \ \wedge \ vis; vis \subseteq vis$
Sequential Consistency (SEQ)	$so \subseteq vis \ \wedge \ vis; vis \subseteq vis \ \wedge \mathsf{total}(vis)$

Table 7.1: Well known consistency criteria

We say that a consistency criterion α is at least as *strong* as another consistency criterion α' if for every history H, visibility relation vis, and arbitration relation arb over H, if $H, \mathsf{vis}, \mathsf{arb} \models \alpha$ then $H, \mathsf{vis}, \mathsf{arb} \models \alpha'$.

Suppose $H = (\mathcal{O}, \mathsf{so})$ is a hybrid history. Let α_w and α_s respectively be the wk and st consistency criteria. We then want to choose wk and st visibility relations $\mathsf{vis}_{\mathsf{wk}}$, $\mathsf{vis}_{\mathsf{st}}$, respectively, and an arbitration relations arb such that $H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{arb} \models \alpha_w$ and $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}}, \mathsf{arb} \models \alpha_s$.

As we had noted in the previous section, in a multilevel setting, it is not sufficient to separately satisfy the constraints corresponding to the wk and st consistency criteria. We now proceed to modelling multilevel consistency constraints.

Modelling Multilevel Consistency

Taking inspiration from DAX [Documentation, 2019] and the cache-hierarchy in modern processors, we can model multilevel consistency as a series of data-stores arranged in increasing order of the consistency they guarantee, such that the data-store offering the weakest level of consistency is closest to the application, and the data-store offering the strongest level of consistency is farthest away from the application. We shall further assume that these data-stores use the same arbitration strategy to order concurrent write operations and every weaker data-store has the capability to update its state to match that of a stronger data-store.

For the purpose of this chapter, since we are restricting ourselves to only two levels, namely wk and st, this will reduce to having just two data-stores, where the data-store corresponding to the weaker consistency criterion sits as a cache between the application and the data-store corresponding to the stronger consistency criterion.

All the wk-reads are performed from the wk data-store.

There are two possible ways in which the writes can be performed.

1. Write-Through: The write is first performed at the st-data-store and eventually will be propagated to the wk-data-store.

2. Write-Back: The write is first performed at the wk-data-store and eventually will be propagated to the st-data-store.

There are two possible ways in which st-reads can be performed.

(a) Read-Through: The result of the st-read performed at the st-data-store is directly sent to the application bypassing the wk-data-store.

(b) Read-Back: The result of the st-read is updated at the wk-data-store before it is propagated to the application.

Thus, the system picks one of two ways to perform the write, and one of the two ways to perform the st-read.

Note that a system which picks the Write-Through strategy for performing the write will ensure that any write visible at the wk data-store will also be visible to the st data-store, as all the writes are first performed at the st data-store before they are propagated to the wk one. Hence, the effects of write operations visible to a wk-read operation are also visible to the subsequent st-operations in the session.

Similarly a system which picks the Read-Back strategy for performing the st-reads will ensure that any write that is visible to a *strong*-read will also be visible at a subsequent wk-read in the session as before returning the result of the st-read to the application, the result is merged into the wk data-store.

However, the Write-Back and Read-Through strategies do not provide any guarantees between the effects of writes visible to wk (resp. st) reads in relation to the subsequent st (resp. wk) reads in that session.

We now define the guarantees provided by each of these four strategies in the form of a constraint.

Definition 159 (Multilevel Constraints)**.** *We define the following formulas:*

- $\psi^{write}_{thru} := (vis^{wk}; so) \restriction_{\mathsf{st}} \subseteq vis^{st}$
- $\psi^{write}_{back} := \top$
- $\psi^{read}_{thru} := \top$
- $\psi^{read}_{back} := (vis^{st}; so) \restriction_{\mathsf{wk}} \subseteq vis^{wk}$

A multilevel constraint φ is a conjunction $\psi^{read} \wedge \psi^{write}$, where $\psi^{read} \in \{\psi^{read}_{thru}, \psi^{read}_{back}\}$ and $\psi^{write} \in \{\psi^{write}_{thru}, \psi^{write}_{back}\}$.

Suppose $H = (\mathcal{O}, \mathsf{so})$ is a history, and $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ are two visibility relations respectively over $\mathcal{O}_{\mathsf{wk}}$ and $\mathcal{O}_{\mathsf{st}}$. Let φ be a multilevel constraint. We say that $H, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}} \models \varphi$ iff $\varphi[so := \mathsf{so}, vis^{wk} := \mathsf{vis}_{\mathsf{wk}}, vis^{st} := \mathsf{vis}_{\mathsf{st}}]$ is true.

The formula ψ^{write}_{thru} imposes the constraint that the strong operations see the effects seen by the prior weak operations in the session. Similarly, the formula ψ^{read}_{back} imposes the constraint that the weak operations see the effects seen by the prior strong operations in the session. These two guarantee that the effect seen by reads of one consistency level remain monotonically visible to the subsequent reads of another consistency level.

Consider Cassandra's multilevel consistency with writes performed at level `ONE`, weak-reads at level `ONE` and strong-reads at level `ALL` which ensure that weaker reads see the effects visible to prior stronger reads and vice-versa. This can be modelled using $\psi^{write}_{thru} \wedge \psi^{read}_{back}$.

On the other hand, Cassandra's multilevel consistency with writes performed at level `ONE`, weak-reads at level `ONE` and strong-reads at level `QUORUM` neither ensures that weaker reads see the effects visible to prior stronger reads nor the converse. This can be modelled using $\psi^{write}_{back} \wedge \psi^{read}_{thru}$.

The DynamoDB's DAX case can be modelled using $\psi^{write}_{thru} \wedge \psi^{read}_{thru}$ which only allows for the effects of prior weak reads to be visible to subsequent stronger reads, but not the converse.

We now formally define when a hybrid history is correct.

Definition 160 (Multilevel Correctness of a Hybrid History)**.** *A hybrid history $H = (\mathcal{O}, \mathsf{so})$ of a read-write store is said to be* multilevel correct *with respect to a* wk*-consistency criterion α_w,* st*-consistency criterion α_s and multilevel consistency constraint φ, iff there exists visibility relations $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ over H_{wk} and H_{st} respectively and arbitration relation* arb *such that*

- $(H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{arb})$ *and* $(H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}}, \mathsf{arb})$ *are functionally correct,*
- $H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{arb} \models \alpha_w$,
- $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}}, \mathsf{arb} \models \alpha_s$, *and*
- $H, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}} \models \varphi$.

7.4 Testing Multilevel Correctness of a Hybrid History

Given a read-write hybrid history $H = (\mathcal{O}, \mathsf{so})$, we want to test it for multi-level correctness with respect to weak and strong consistency criteria α_w and α_s and multilevel constraints given by φ.

We note that for the history to be correct, for every read operation that returns a value that is not $\perp$, there should exist a write operation writing the same value to the variable that was read. The *reads-from* relation associates a write operation to the read that reads its effect. Our strategy for testing the multilevel correctness of H is to enumerate all such *reads-from* relations rf, for each rf we find visibility relations $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$, respectively, containing $\mathsf{rf}_{\mathsf{wk}}$ and $\mathsf{rf}_{\mathsf{st}}$, such that they satisfy the visibility constraints imposed by the individual consistency criteria, as well as the multilevel constraints, i.e., $H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}} \models \alpha_w$, $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}} \models \alpha_s$ and $H, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}} \models \varphi$. We then check for the presence of a finite number of *bad-patterns* in these visibility relations. The presence of a bad-pattern implies that for every arbitration relation arb, there is some level $\ell \in \{\mathsf{wk}, \mathsf{st}\}$ such that either the arbitration constraint $\mathsf{vis}_\ell \restriction_{\mathsf{Write}} \subseteq \mathsf{arb}$ is not satisfied, or the history $(H_\ell, \mathsf{vis}_\ell, \mathsf{arb})$ is not functionally correct.

If the history is multi-level correct, then we will find a witness consisting of a *reads-from* relation rf and visibility relations $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ extending $\mathsf{rf}_{\mathsf{wk}}$ and $\mathsf{rf}_{\mathsf{st}}$ such that all the constraints are satisfied and there are no bad-patterns. If the history is not multi-level correct, then for every pair of weak and strong visibility relation extending every *reads-from* relation, either some constraint is not satisfied or there exists a bad-pattern.

We present the bad-pattern characterization for multilevel correctness of a hybrid history in the next subsection. In the following subsection, we provide a procedure for computing the minimal visibility relations $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ for a given *reads-from* relation rf that satisfies α_w, α_s and φ.

7.4.1 Bad Pattern characterization for multilevel correctness

We now characterize the correctness of hybrid histories based on the non-existence of certain bad patterns. This is a generalization of the bad-pattern characterization for causal consistency in [Bouajjani et al., 2017].

Given a hybrid history, we can associate each Read with a unique write operation from the history whose effect the Read operation reads from. We call this the *reads-from* relation.

Definition 161 (Reads-From)**.** *A* reads-from *relation* rf *over a history* $H = (\mathcal{O}, \mathsf{so})$ *is a binary relation such that*

1. $(o_i, o_j) \in \mathsf{rf} \implies o_i$ *is a* Write, o_j *is a* Read*, both on the same variable, such that the value returned by* o_j *is the value written by* o_i*.*
2. $(o_i, o_j) \in \mathsf{rf} \wedge (o_k, o_j) \in \mathsf{rf} \implies o_i = o_k$*.*
3. *For all* $o_j = \mathsf{Read}(x, val, level) \in \mathcal{O}_{\mathsf{Read}}$

$$[\exists\ o \in \mathcal{O}_{\mathsf{Write}} \textit{ which writes } val \textit{ to } x \implies \exists\ o_i \in \mathcal{O}_{\mathsf{Write}} : (o_i, o_j) \in \mathsf{rf}.]$$

Condition 1 associates a read operation with a write operation only if they operate on the same variable and that the return value of the read operation matches the argument of the write operation.

Condition 2 ensures that a read operation is associated with at most one write operation.

Finally, Condition 3 insists that if a Read is not related to any Write via rf, it is only because there is no matching Write in the hybrid history (i.e. a write of the same value to the same variable).

Let rf be a *reads-from* relation on a hybrid history $H = (\mathcal{O}, \mathsf{so})$. For a Read operation $o \in \mathcal{O}$, if there exists a Write operation o' such that $(o', o) \in \mathsf{rf}$, then we say that $\mathsf{rf}^{-1}(o) = o'$. If no such o' exists, we set $\mathsf{rf}^{-1}(o) = \bot$.

Further, we denote by $\mathsf{rf}_{\mathsf{wk}}$ and $\mathsf{rf}_{\mathsf{st}}$ the *reads-from* relation restricted to H_{wk} and H_{st} respectively.

Suppose rf_ℓ is a *reads-from* relation over H_ℓ. We say that a visiblity relation vis_ℓ over H_ℓ *extends* rf_ℓ iff $\mathsf{rf}_\ell \subseteq \mathsf{vis}_\ell$. Suppose arb is an arbitration relation over H_ℓ. Then, we say that $(\mathsf{vis}_\ell, \mathsf{arb})$ *realize* rf_ℓ iff for all read operations $o \in \mathcal{O}_\ell$, $\mathsf{rf}_\ell^{-1}(o) = \mathit{EffWrite}^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o)$.

Given a *reads-from* relation rf_ℓ and a visibility relation vis_ℓ that extends it, we can define a conflict relation that orders all the remaining maximal related writes in $\mathit{MaxRelWrites}_{\mathsf{vis}_\ell}(o)$ of a read-operation o before the write-operation $\mathsf{rf}_\ell^{-1}(o)$. The conflict relation captures the essence of the arbitration relation for a given *reads-from* relation and a visibility relation extending it.

Definition 162 (Conflict Relation)**.** *Let* $H_\ell = (\mathcal{O}_\ell, \mathsf{so}_\ell)$ *be a history. Let* rf_ℓ *be a* reads-from *relation over* H_ℓ*. Let* $\mathsf{vis}_\ell \supseteq \mathsf{rf}_\ell$ *be a visiblity relation over* H_ℓ*. We define the conflict relation for* rf_ℓ *and* vis_ℓ*, denoted* $\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell)$*, as the set*

$$\{(o'', o') \mid \exists o \in \mathcal{O}_\ell \restriction_{\mathsf{Read}} : o'', o' \in \mathit{MaxRelWrites}_{\mathsf{vis}}(o) \wedge o' = \mathsf{rf}_\ell^{-1}(o)\}.$$

We now define the bad patterns that characterize the correctness of the hybrid history.

Definition 163 (Bad Patterns for a hybrid history)**.** *Let $H = (\mathcal{O}, \mathsf{so})$ be a hybrid history with weak and strong consistency criteria α_w and α_s respectively and multilevel constraints φ. Let* rf *be a* reads-from *relation over H. For $\ell \in \{\mathsf{wk}, \mathsf{st}\}$, let vis_ℓ be a relation over $\mathcal{O}_\ell$ with $\mathsf{vis}_\ell \supseteq \mathsf{rf}_\ell$ such that $H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}} \models \alpha_w$, $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}} \models \alpha_s$ and $H, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}} \models \varphi$. We define the following bad patterns for $(H, \mathsf{rf}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}})$. For some $\ell \in \{\mathsf{wk}, \mathsf{st}\}$:*

- BADVISIBILITY*:* $\mathsf{Cyclic}(\mathsf{vis}_\ell)$
- THINAIR*:* $\exists o \in \mathcal{O}_{\mathsf{Read}}\restriction_\ell$*: o returns a value that is not $\perp$, but* $\mathsf{rf}_\ell^{-1}(o) = \perp$
- BADINITREAD*:* $\exists o \in \mathcal{O}_{\mathsf{Read}}\restriction_\ell$*: o returns $\perp$ but* $RelWrites_{\mathsf{vis}_\ell}(o) \neq \emptyset$
- BADREAD*:* $\exists o \in \mathcal{O}_{\mathsf{Read}}\restriction_\ell$: $\mathsf{rf}_\ell^{-1}(o) \notin MaxRelWrites_{\mathsf{vis}_\ell}(o)$
- BADARB*:* $\mathsf{Cyclic}(\bigcup_{\ell \in \{\mathsf{wk},\mathsf{st}\}} (\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell) \cup (\mathsf{vis}_\ell)_{\mathsf{Write}}))$

BADVISIBILITY says that one of the visibility relations has a cycle.

THINAIR says that there exists a read in the history which reads a non-initial value which is not written by any write operation in the hybrid history.

BADINITREAD says that there is a read operation on a variable which reads the initial value despite having a non-initial write to that variable in its view.

BADREAD says that the write operation from which the read-operation reads is not a maximal write, and there are other writes in the view of the read operation that would have overwritten the value written by that write.

BADARB says that the union of the conflict relations along visibility relation restricted to only the Write operations has a cycle indicating that there exists no total-order arb over $\mathcal{O}_{\mathsf{Write}}$, such that $(\mathsf{vis}_\ell, \mathsf{arb})$ realizes rf_ℓ.

Multi-level correctness of a hybrid history can be characterized in terms of non-existence of these bad patterns. We prove this in the next section 7.5.

Theorem 164 (Bad patterns characterization)**.** *A hybrid history $H = (\mathcal{O}, \mathsf{so})$ is said to be multilevel correct with respect to weak and strong consistency criteria α_w, α_s and multilevel constraint φ iff there exists a* reads-from *relation* rf*, and relations* $\mathsf{vis}_{\mathsf{wk}} \supseteq \mathsf{rf}_{\mathsf{wk}}$ *and* $\mathsf{vis}_{\mathsf{st}} \supseteq \mathsf{rf}_{\mathsf{st}}$ *respectively over $\mathcal{O}_{\mathsf{wk}}$ and $\mathcal{O}_{\mathsf{st}}$ such that $H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}} \models \alpha_w$, $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}} \models \alpha_s$ and $H, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}} \models \varphi$ and no bad pattern exists in $(H, \mathsf{rf}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}})$.*

7.5 Correctness of the Bad Patterns Charecterization

Lemma 165. *If* rf_ℓ *is a* reads-from *relation over the history H_ℓ and* $(\mathsf{vis}_\ell, \mathsf{arb})$ *realize* rf_ℓ*. Then,* $\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell) \subseteq \mathsf{arb}$*.*

Proof. Suppose $(o'', o') \in \mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell)$. By definition, there exists a Read operation o such that both o', o'' are in the maximal related writes of o and $o' = \mathsf{rf}_\ell^{-1}(o)$. Since rf_ℓ is realized by $(\mathsf{vis}_\ell, \mathsf{arb})$, by definition, $\mathsf{rf}_\ell^{-1}(o) = \mathit{EffWrite}^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o)$. Hence o' is the effective write of o.

Now by the definition, the arbitration relation arb orders an effective write of a read operation after all the other maximal related writes of that read operation. Thus, $(o'', o) \in \mathsf{arb}$. □

We now prove the correctness of Theorem 164

Let $H = (\mathcal{O}, \mathsf{so})$ be a hybrid history and let α_w and α_s respectively be the weak and strong consistency criteria. Let the multilevel constraints be defined by φ. We need to show that H is multilevel correct with respect to α_w, α_s and φ iff there exists a *reads-from* relation rf and visibility relations $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ that extend $\mathsf{rf}_{\mathsf{wk}}$ and $\mathsf{rf}_{\mathsf{st}}$ respectively such that $H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}} \models \alpha_w$, $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}} \models \alpha_s$, $H, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}} \models \varphi$ and none of the bad patterns $\{\mathsf{BADVISIBILITY}, \mathsf{THINAIR}, \mathsf{BADINITREAD}, \mathsf{BADREAD}, \mathsf{BADARB}\}$ exists in $(H, \mathsf{rf}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}})$.

In the proof below, and the ones that follow we shall use the following notation:

- For a read-operation $o = \mathsf{Read}(x, \mathit{val}, \mathit{level})$ in $\mathcal{O}$, we denote by $\mathit{Var}(o)$ the variable x, $\mathit{Ret}(o)$ the return value val and $\mathit{Level}(o)$ the level level.
- Similarly , for a write-operation $o = \mathsf{Write}(x, \mathit{val})$ in $\mathcal{O}$, we denote by $\mathit{Var}(o)$ the variable x, $\mathit{Args}(o)$ the input value val.

Proof. ($\Longrightarrow$): Suppose hybrid history H is correct. Then, there exists visibilty relations $\mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}}$ and arbitration relations arb such that $(H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{arb})$ and $(H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}}, \mathsf{arb})$ are functionally correct, $H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{arb} \models \alpha_w$, $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}}, \mathsf{arb} \models \alpha_s$ and $H, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}} \models \varphi$.

Thus, we have

- $H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}} \models \alpha_w$ and $\mathsf{vis}_{\mathsf{wk}} \restriction_{\mathsf{Write}} \subseteq \mathsf{arb}$
- $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}} \models \alpha_s$ and $\mathsf{vis}_{\mathsf{st}} \restriction_{\mathsf{Write}} \subseteq \mathsf{arb}$

For $\ell \in \{\mathsf{wk}, \mathsf{st}\}$,we set $\mathsf{rf}_\ell = \{(\mathit{EffWrite}^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o), o) \mid o \in \mathcal{O}_{\mathsf{Read}} : \mathit{Level}(o) = \ell\}$. $\mathsf{rf} = \mathsf{rf}_{\mathsf{wk}} \cup \mathsf{rf}_{\mathsf{st}}$. By definition $\mathsf{vis}_{\mathsf{wk}}$ extends $\mathsf{rf}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ extends $\mathsf{rf}_{\mathsf{st}}$.

We will now show that none of the aforementioned bad patterns exists for the tuple $(H, \mathsf{rf}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}})$.

Since H is multilevel correct, $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ by definitions are acyclic relations. So $\mathsf{BADVISIBILITY}$ bad pattern doesn't exist.

Further, due to functional correctness of $(H_\ell, \mathsf{vis}_\ell, \mathsf{arb})$, for any read operation o of level ℓ, $\mathit{EffWrite}^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o) = \bot$ iff $\mathit{Ret}(o) = \bot$. Since for every read operation o, $\mathsf{rf}_\ell^{-1}(o) = \mathit{EffWrite}^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o)$, it follows that $\mathsf{rf}_\ell^{-1} = \bot$ iff $\mathit{Ret}(o) = \bot$. Thus, the $\mathsf{THINAIR}$ bad pattern doesn't exist.

Since $(H_\ell, \mathsf{vis}_\ell, \mathsf{arb})$ is functionally correct, for any read operation o with level ℓ such that $Ret(o) = \bot$, $EffWrite^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o) = \bot$ which implies that $RelWrites_{\mathsf{vis}_\ell}(o) = \emptyset$. Hence, the BADINITREAD pattern doesn't exist.

For a functionally correct history H_ℓ, for any read operation o with level ℓ, if $Ret(o) \neq \bot$, then $EffWrite^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o) \neq \bot$. This implies that $EffWrite^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o) \in MaxRelWrites_{\mathsf{vis}_\ell}(o)$. But we have set $\mathsf{rf}^{-1}_\ell(o) = EffWrite^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o)$. Thus $\mathsf{rf}^{-1}_\ell(o) \in EffWrite^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o)$. Hence, the BADREAD pattern doesn't exist.

By construction, rf_ℓ is realized by $(\mathsf{vis}_\ell, \mathsf{arb})$. Hence from lemma 165 for $\ell \in \{\mathsf{wk}, \mathsf{st}\}$, $\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell) \subseteq \mathsf{arb}$. Due to functional correctness of $(H_\ell, \mathsf{vis}_\ell, \mathsf{arb})$, we have $\mathsf{vis}_\ell \restriction_{\mathsf{Write}} \subseteq \mathsf{arb}$. Hence $\bigcup_{\ell \in \{\mathsf{wk},\mathsf{st}\}} (\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell) \cup \mathsf{vis}_\ell \restriction_{\mathsf{Write}}) \subseteq \mathsf{arb}$. By definition arb is a total order. Thus, BADARB would imply a cycle in arb which is not true. Hence, the BADARB pattern doesn't exist.

This completes one side direction of the proof.

($\Longleftarrow$): Suppose there exists a $(\mathsf{rf}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}})$ such that $\mathsf{vis}_{\mathsf{wk}} \supseteq \mathsf{rf}_{\mathsf{wk}}$, $\mathsf{vis}_{\mathsf{st}} \supseteq \mathsf{rf}_{\mathsf{st}}$, $H, \mathsf{vis}_{\mathsf{wk}} \models \alpha_w$ and $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}} \models \alpha_s$, $H, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}} \models \varphi$ and $(H, \mathsf{rf}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}})$ does not have any bad-patterns. To show that H is multi-level correct, we need to show that there exists an arbitration relation arb such that $\mathsf{vis}_\ell \restriction_{\mathsf{Write}} \subseteq \mathsf{arb}$ and $(H_\ell, \mathsf{vis}_\ell, \mathsf{arb})$ is functionally correct for $\ell \in \{\mathsf{wk}, \mathsf{st}\}$.

We first construct the arbitration relation arb. Since the BADARB bad pattern doesn't exist, $\bigcup_{\ell \in \{\mathsf{wk},\mathsf{st}\}} (\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell) \cup \mathsf{vis}_\ell \restriction_{\mathsf{Write}})$ is an acyclic relation. We set arb to be a topological sort of this acyclic relation along with the Write operations from o, not appearing in this acyclic relations. Thus, arb is a total order. By construction, $\mathsf{vis}_\ell \restriction_{\mathsf{Write}} \subseteq \mathsf{arb}$ for $\ell \in \{\mathsf{wk}, \mathsf{st}\}$. From this, and what is given we can conclude that $H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{arb} \models \alpha_w$ and $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}}, \mathsf{arb} \models \alpha_s$.

We now only need to show that for each $\ell \in \{\mathsf{wk}, \mathsf{st}\}$, $(H_\ell, \mathsf{vis}_\ell, \mathsf{arb})$ is functionally correct.

Let o be a read operation with level ℓ. Suppose $MaxRelWrites_{\mathsf{vis}_\ell}(o) = \emptyset$. Then $EffWrite^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o) = \bot$. Since $\mathsf{rf}_\ell \subseteq \mathsf{vis}_\ell$, $\mathsf{rf}^{-1}_\ell(o) = \bot$. Since THINAIR bad pattern doesn't exist, it has to be the case that $Ret(o) = \bot$. Thus, if $EffWrite^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o) = \bot$ then $Ret(o) = \bot$. Conversely, suppose $Ret(o) = \bot$. Then, since BADINITREAD bad pattern doesn't exist, $RelWrites_{\mathsf{vis}_\ell}(o) = \emptyset$. Thus, by definition, $EffWrite^{\mathsf{arb}}_{\mathsf{vis}\ell}(o) = \bot$. Thus, we can conclude that $EffWrite^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o) = \bot \iff Ret(o) = \bot$.

Suppose $MaxRelWrites_{\mathsf{vis}_\ell}(o) \neq \emptyset$. Since BADINITREAD bad pattern doesn't exist, $Ret(o) \neq \bot$. Further, since THINAIR badpattern doesn't exist, $\mathsf{rf}^{-1}_\ell(o) \neq \bot$. Let $\mathsf{rf}^{-1}_\ell(o) = o'$. Since BADREAD bad pattern doesn't exist, $\mathsf{rf}^{-1}_\ell(o) = o' \in MaxRelWrites_{\mathsf{vis}_\ell}(o)$. For any $o'' \in MaxRelWrites_{\mathsf{vis}_\ell}(o)$ we have $(o'', o') \in \mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell)$. Now, by construction of arb, we have $\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell) \subseteq \mathsf{arb}$.

Thus, for any $o'' \in MaxRelWrites_{\mathsf{vis}_\ell}(o)$, $o'' \xrightarrow{\mathsf{arb}} o'$. Thus, by definition, $EffWrite^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o) = o'$. However, since $o' = \mathsf{rf}^{-1}_\ell(o)$, by definition of a *reads-from* relation, $Ret(o) = Args(o')$. Thus, $o' = EffWrite^{\mathsf{arb}}_{\mathsf{vis}_\ell}(o)$ wrote the value read by o.

Since o is an arbitrary Read operation with level ℓ in H, what we have shown holds for all Read operation with level ℓ. Hence $(H_\ell, \mathsf{vis}_\ell, \mathsf{arb})$ is functionally correct. Hence H is multi-level correct. □ □

7.5.1 Constructing Minimal Visibility Relations

Suppose $H = (\mathcal{O}, \mathsf{so})$ is a hybrid history. Let α_w and α_s be the formulas defining the weak and strong consistency criteria, and let φ be the formula defining the multilevel constraints. Let $\alpha'_w = \mathit{VisBasic}(\alpha_w)$ and $\alpha'_s = \mathit{VisBasic}(\alpha_s)$.

We provide a procedure that iterates over all the possible *reads-from* relations and constructs a minimal visibility relation extending the *reads-from* relation such that it satisfies α_w, α_s and φ. The pseudo-code for the procedure is presented in Algorithm 7 and 8.

Algorithm 7 Constructing minimal visibility relations

```
1   MinVisOne(O_ℓ, so_ℓ, vis_ℓ, α_ℓ):
2       Let vis_o := vis_ℓ;
3
4       while (True):
5           Let vis_p := vis_o;
6           for τ ∈ RelTerms(α_ℓ)):
7               vis_n := vis_p ∪ τ[so_ℓ, vis_p];
8               vis_p := vis_n;
9           if (vis_n == vis_o)
10              return vis_n
11          vis_o := vis_n
12
13
14  ComputeVisSet(O_ℓ, so_ℓ, vis_ℓ, α_ℓ)
15      if total(vis) is a subformula in α_ℓ:
16          visSet_ℓ := {totvis|totvis
                         is a
                         total order over
                          O_ℓ such that
                          vis_ℓ ⊆ totvis}
17      else :
18          visSet_ℓ := {vis_ℓ}
19
20      return visSet_ℓ
21  MinVisMulti(O, so, vis_wk, vis_st, ψ)
22      if ψ ∈ {ψ^write_back, ψ^read_thru}:
23          return (vis_wk, vis_st)
24      else if ψ ∈ {ψ^write_thru}:
25          Let ℓ = st, ℓ' = wk;
26      else if ψ ∈ {ψ^read_back}:
27          Let ℓ = wk, ℓ' = st;
28
29      Let vis^o_ℓ := vis_ℓ;
30      Let vis^o_ℓ' := vis_ℓ';
31
32      if ψ ∈ {ψ^write_thru, ψ^read_back}:
33          Let vis^n_ℓ := vis^o_ℓ ∪ (vis^o_ℓ'; so)↑_ℓ;
34          Let vis^n_ℓ' := vis^o_ℓ';
35
36       if ψ ∈ {ψ^read_back}:
37          return (vis^n_ℓ, vis^n_ℓ')
38      else if ψ ∈ {ψ^write_thru}:
39          return (vis^n_ℓ', vis^n_ℓ)
40
41
42
```

In Lines 1-12 we have a method MinVisOne that takes as input a visiblity relation vis_ℓ for the history $(\mathcal{O}_\ell, \mathsf{so}_\ell)$ and constructs an extension vis_n that satisfies the formula $\mathit{VisBasic}(\alpha_\ell)$. We achieve this by iterating over the *RelTerms* appearing in $\mathit{RelTerms}(\alpha_\ell)$ (Line 6) and

extending the previous visibility relation $\mathsf{vis_p}$ with the evaluation of the term (Line 7). We do this until we obtain a relation $\mathsf{vis_n}$ which we can no longer extend (Line 9). This final visibility relation $\mathsf{vis_n}$ extends vis_ℓ and satisfies the formula $VisBasic(\alpha_\ell)$.

In Lines 21-40, we have the procedure MinVisMulti which takes as inputs the hybrid history $(\mathcal{O}, \mathsf{so})$, visibility relations $\mathsf{vis_{wk}}$ and $\mathsf{vis_{st}}$ and an individual conjunct ψ appearing in the multilevel constraint φ. Since every visibility relation trivially satisfies ψ_{back}^{write} or ψ_{thru}^{read}, for these multilevel constraint, we simply return without modifying $\mathsf{vis_{wk}}$ or $\mathsf{vis_{st}}$ (Lines 22-23). In the remaining cases, when the multi-level constraint is either ψ_{back}^{write} or ψ_{thru}^{read}, for $\ell, \ell' \in \{\mathsf{wk}, \mathsf{st}\}$, the multilevel constraints relates the write operations visible to the operations of level ℓ in terms of the writes seen by operations of level ℓ' that have occured previously in the session. Depending on the conjunct ψ, we set ℓ and ℓ' appropriately(Lines 24-27). We then extend the visibility relation for level ℓ by relating each ℓ-operation to the Writes that have been seen by any of the ℓ'-operations prior to the ℓ-operation in its session (Line 33). The visibility relation for level ℓ' remains unchanged in this case (Line 34).

We return these extended visibility relations as a pair, where the wk visibility extension is followed by st visiblity extension (Lines 36-39).

In the algorithm titled *Testing multilevel correctness of hybrid history* on Page 198, in Lines 1-19 we have the procedure ComputeStableExt which takes history $(\mathcal{O}, \mathsf{so})$ a pair of visibility relations $\mathsf{vis_{wk}}$ and $\mathsf{vis_{st}}$ and extends it to $\mathsf{vis^n_{wk}}$ and $\mathsf{vis^n_{st}}$ such that they individually satisfy $VisBasic(\alpha_w)$ (Line 7) and $VisBasic(\alpha_s)$ (Line 9) respectively and jointly satisfy φ (Lines 11-14). We repeat this till we can extend these relations no longer, which implies that they have satisfied all the constraints (Lines 16-17).

The procedure TestMultiCorrect in Lines 20-35 takes as input a hybrid history $H = (\mathcal{O}, \mathsf{so})$ whose multilevel correctness we want to check with respect to formulas α_w , α_s and φ.

We first enumerate the set of possible *reads-from* relations on the history (line 21). We then iterate through each of the *reads-from* relations rf to see whether it can be extended to construct a minimal visibility relation satisfying all the constraints and having no bad-patterns (Lines 22-33). For each rf, we construct minimal visibility relations $\mathsf{vis^{min}_{wk}}$ and $\mathsf{vis^{min}_{st}}$ extending $\mathsf{rf_{wk}}$ and $\mathsf{rf_{st}}$ respectively and satisfying the subformulas $VisBasic(\alpha_w)$ and $VisBasic(\alpha_s)$ respectively (Lines 23,26).

If α_w (resp. α_s) contains the subformula $\mathsf{total}(vis)$, we enumerate the set of all the total orders extending $\mathsf{vis^{min}_{wk}}$ (resp. $\mathsf{vis^{min}_{st}}$) in the set $\mathsf{visSet_{wk}}$ (resp. $\mathsf{visSet_{st}}$) in Line 24 (resp. Line 27). If α_w (resp. α_s) does not contain the subformula $\mathsf{total}(vis)$, then, $\mathsf{visSet_{wk}}$ (resp. $\mathsf{visSet_{st}}$) will contain the only minimum visibility relation extending $\mathsf{rf_{wk}}$ (resp. $\mathsf{rf_{st}}$), i.e., $\mathsf{vis^{min}_{wk}}$ (resp. $\mathsf{vis^{min}_{st}}$.).

For each pair of visibility relations from $\mathsf{visSet_{wk}}$ and $\mathsf{visSet_{st}}$ we compute their stable extensions $\mathsf{vis^{stb}_{wk}}$ and $\mathsf{vis^{stb}_{st}}$ which individually satisfy α_w and α_s, respectively, and jointly satisfy φ (line 30). We then check if this computed extension has a bad pattern (Line 32). If no bad patterns are found, we return the $(\mathsf{rf}, \mathsf{vis_{wk}}, \mathsf{vis_{st}})$ as the witness.

Algorithm 8 Testing multilevel correctness of a hybrid history

```
1  ComputeStableExt(O, so, vis_wk, vis_st, α_w, α_s, φ):
2      Let vis^o_wk := vis_wk,
         vis^o_st := vis_st
3
4      while (True):
5          Let vis^p_wk := vis^o_wk,
             vis^p_st := vis^o_st
6
7          Let vis^n_wk :=
             MinVisOne(O_wk, so_wk, vis^p_wk, α_w);
8
9          Let vis^n_st :=
             MinVisOne(O_st, so_st, vis^p_st, α_s);
10
11         for each subformula ψ_i
             in the conjunction φ:
12             vis^p_wk := vis^n_wk, vis^p_st := vis^n_st
13
14             (vis^n_wk, vis^n_st) =
             MinVisMulti(O, so, vis^p_wk, vis^p_st, ψ_i)
15
16         if vis^n_wk = vis^o_wk and vis^n_st = vis^o_st:
17             return (vis^n_wk, vis^n_st)
18
19         vis^o_wk := vis^n_wk, vis^o_st := vis^n_st

20 TestMultiCorrect(O, so, α_w, α_s, φ):
21     Let rfSet := {rf|rf is a reads-from
                   relation over (O, so)}
22     for rf ∈ rfSet:
23         Let vis^min_wk :=
           MinVisOne(O_wk, so_wk, rf_wk, α_w);
24         Let visSet_wk =
           ComputeVisSet(O_wk, so_wk, vis^min_wk, α_w);
25
26         Let vis^min_st :=
           MinVisOne(O_wk, so_st, rf_st, α_s);
27         Let visSet_st :=
           ComputeVisSet(O_st, so_st, vis^min_st, α_s);
28
29         for vis_wk ∈ visSet_wk, vis_st ∈ visSet_st:
30             Let (vis^stb_wk, vis^stb_st) :=
               ComputeStableExt(O, so, vis_wk,
                                vis_st, α_w, α_s, φ);
31
32             if BadPatterns(O, so, rf,
                              vis^stb_wk, vis^stb_st) ==
                                 NoBadPatterns:
33                 return (rf, vis^stb_wk, vis^stb_st)
34
35     return BadHistory
```

If none of the rf can be extended to obtain the required visibility relation, we declare that the history is a bad history. We formally prove the correctness of TestMultiCorrect in the next section 7.6.

Theorem 166 (Correctness of TestMultiCorrect procedure)**.** *For a hybrid read-write history $H = (\mathcal{O}, \mathsf{so})$ with weak and strong consistency criteria α_w and α_s respectively and multilevel constraints given by φ, the procedure* TestMultiCorrect *returns a witness $(\mathsf{rf}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}})$ over H iff H is multi-level correct with respect to α_w, α_s and φ.*

7.6 Correctness of the testing procedure

We will first prove a set of lemmas with respect to the termination and the correctness of the helper procedures.

Lemma 167 (Termination of Helper functions)**.** *For a given hybrid history, and a given visibility relations over the history, the methods* MinVisOne, MinVisMulti *and* ComputeStableExt *terminate.*

Proof. We first observe that MinVisMulti terminates since it doesn't have any loops. The visibility relations it outputs is a superset of the input visibility relations.

We will now show the termination of MinVisOne. Let $\mathsf{vis}_{\mathsf{n}}^{i,j}$ denote the value of $\mathsf{vis}_{\mathsf{n}}$ at the end of the j^{th} iteration of the inner **for**-loop within the i^{th} iteration of the outer **while**-loop. Let $\mathsf{vis}_{\mathsf{n}}^{i}$ denote the value of $\mathsf{vis}_{\mathsf{n}}$ at the end of the i^{th} iteration of the outer **while**-loop.

We note that for $j > 0$, $\mathsf{vis}_{\mathsf{n}}^{i,j} \supseteq \mathsf{vis}_{\mathsf{n}}^{i,j-1}$ since we only keep extending $\mathsf{vis}_{\mathsf{n}}$ inside the inner **for**-loop by adding to it the result of evaluation of the *RelTerms* in α_ℓ. $\mathsf{vis}_{\mathsf{n}}^{i,0} = \mathsf{vis}_{\mathsf{n}}^{i-1}$. If $|RelTerms(\alpha_\ell)| = k$, then, $\mathsf{vis}_{\mathsf{n}}^{i} = \mathsf{vis}_{\mathsf{n}}^{i,k}$. Since $\mathsf{vis}_{\mathsf{n}}^{i,k} \supseteq \mathsf{vis}_{\mathsf{n}}^{i,0}$, it follows that $\mathsf{vis}_{\mathsf{n}}^{i} \supseteq \mathsf{vis}_{\mathsf{n}}^{i-1}$. At the end of the outer-while loop we check if $\mathsf{vis}_{\mathsf{n}} = \mathsf{vis}_{\mathsf{o}}$ which is equivalent to checking $\mathsf{vis}_{\mathsf{n}}^{i} = \mathsf{vis}_{\mathsf{n}}^{i-1}$. If true, the function returns. Since $\mathsf{vis}_{\mathsf{n}} \subseteq \mathcal{O} \times \mathcal{O}$, and since $\mathcal{O}$ is a finite set, it will be the case that $\mathsf{vis}_{\mathsf{n}} = \mathsf{vis}_{\mathsf{o}}$ after a finite number of iterations. Hence the procedure terminates.

In case of ComputeStableExt , we note that it obtains the new values for $\mathsf{vis}_{\mathsf{wk}}^{\mathsf{n}}$ and $\mathsf{vis}_{\mathsf{st}}^{\mathsf{n}}$ individually by invoking the procedure MinVisOne, which returns a relation that is a superset of the input visibility relation. Similarly, in the inner **for**-loop, we obtain the new values for the pair $(\mathsf{vis}_{\mathsf{wk}}^{\mathsf{n}}, \mathsf{vis}_{\mathsf{st}}^{\mathsf{n}})$ by calling MinVisMulti, which returns visibility relations that are supersets of the corresponding input relations. Thus, at the end of each iteration of **while**-loop, either the values of $\mathsf{vis}_{\mathsf{wk}}^{\mathsf{n}}$ and $\mathsf{vis}_{\mathsf{st}}^{\mathsf{n}}$ are the same as their values at the end of the previous iteration of the **while**-loop, or they are a superset of their values at the end of the the previous generation. Since both $\mathsf{vis}_{\mathsf{wk}}^{\mathsf{n}}$ and $\mathsf{vis}_{\mathsf{wk}}^{\mathsf{n}}$ are binary relations over $\mathcal{O}_{\mathsf{wk}}$ and $\mathcal{O}_{\mathsf{st}}$, their maximal size is bound by $|\mathcal{O}|^2$. Thus, the iterations of the outer while loop are bounded by $O(|\mathcal{O}|^2)$ iterations. Hence ComputeStableExt terminates. □

Theorem 168 (Termination of Testing Procedure)**.** *For any given hybrid-history H, and consistency criteria α_w, α_s and multilevel constraints φ, the procedure* TestMultiCorrect *terminates.*

Proof. Since H is a finite history, the number of *reads-from* relations that can be defined over it are finite. Further, for each rf from the set of *reads-from* relations, the extensions $\mathsf{vis}_{\mathsf{wk}}^{\min}$ and $\mathsf{vis}_{\mathsf{st}}^{\min}$ are finite. In the worst case when both α_w as well as α_s contain the subsformula $\mathsf{total}(vis)$, the sizes of $\mathsf{visSet}_{\mathsf{wk}}$ and $\mathsf{visSet}_{\mathsf{st}}$ is finite. Since the procedures called within the inner **for**-loop, i.e. ComputeStableExt and BadPatterns, terminate, the inner **for**-loop (Lines 29-33) will iterate only for a finite number of times.

Thus, the procedure will terminate when either it has found a witness rf, $\mathsf{vis}_{\mathsf{wk}}^{\mathsf{stb}}$ and $\mathsf{vis}_{\mathsf{st}}^{\mathsf{stb}}$ for the correctness of the hybrid history, or when it has iterated over all the finitely many *reads-from* relation. □

Lemma 169 (Correctness of MinVisOne)**.** *Let vis_ℓ be a visiblity relation over the history H_ℓ. Let α_ℓ be a consistency criteria. Let $\mathsf{vis} := \mathsf{MinVisOne}(H_\ell, \mathsf{vis}_\ell, \alpha_\ell)$. Then $H_\ell, \mathsf{vis} \models VisBasic(\alpha_\ell)$.*

Proof. We will denote the value of vis_n at the end of the i^{th} iteration of the outer while-loop as $\mathsf{vis}_\mathsf{n}^i$. We shall denote the value of vis_n at the end of the the j^{th} iteration in the i^{th} iteration of the inner for loop as $\mathsf{vis}_\mathsf{n}^{i,j}$.

Let vis be the value returned by MinVisOne at the end of the k^{th} iteration of the outer **while**-loop. Then, $\mathsf{vis} = \mathsf{vis}_\mathsf{n}^k$.

Note that vis_o is the value of vis_n at the end of the previous iteration of while loop. Thus $\mathsf{vis}_\mathsf{o}^k = \mathsf{vis}_\mathsf{n}^{k-1}$. Further since $\mathsf{vis}_\mathsf{o} = \mathsf{vis}_\mathsf{n}$ for the function to return, we have $\mathsf{vis}_\mathsf{n}^k = \mathsf{vis}_\mathsf{o}^k = \mathsf{vis}_\mathsf{n}^{k-1}$. Let $\mathsf{vis}_\mathsf{n}^{k,0}$ denote the value of vis_n at the beginning of the inner for-loop. Then $\mathsf{vis}_\mathsf{n}^{k,0} = \mathsf{vis}_\mathsf{n}^{k-1}$. Suppose $RelTerms(\alpha)$ has N terms where the n^{th} term is denoted by τ_n, then, we can see that for $j \in [1, \ldots, N]$, $\mathsf{vis}_\mathsf{n}^{k,j} = \mathsf{vis}_\mathsf{n}^{k,j-1} \cup \tau_j[\mathsf{so}_\ell, \mathsf{vis}_\mathsf{n}^{k,j-1}]$. Thus, we can conclude that $\tau_j[\mathsf{so}_\ell, \mathsf{vis}_\mathsf{n}^{k,j-1}] \subseteq \mathsf{vis}_\mathsf{n}^{k,j}$.

Also, we can note that $\mathsf{vis}_\mathsf{n}^{k-1} = \mathsf{vis}_\mathsf{n}^{k,0} \subseteq \mathsf{vis}_\mathsf{n}^{k,1} \subseteq \cdots \subseteq \mathsf{vis}_\mathsf{n}^{k,N} = \mathsf{vis}_\mathsf{n}^k$. Since, $\mathsf{vis}_\mathsf{n}^{k-1} = \mathsf{vis}_\mathsf{n}^i$, this implies that for each $j \in [0, \ldots, N]$, $\mathsf{vis}_\mathsf{n}^{k,j} = \mathsf{vis}_\mathsf{n}^k = \mathsf{vis}$.

Thus, for each $j \in [1, \ldots, N]$, we have $\tau_j[\mathsf{so}_\ell, \mathsf{vis}] \subseteq \mathsf{vis}$. Hence,
$\mathsf{so}_\ell, \mathsf{vis} \models \bigwedge_{\tau_j \in RelTerms(\alpha_\ell)} (\tau_j \subseteq vis)$. But by definition,
$\bigwedge_{\tau_j \in RelTerms(\alpha_\ell)} (\tau_j \subseteq vis) = VisBasic(\alpha)$. Hence $\mathsf{so}_\ell, \mathsf{vis} \models VisBasic(\alpha)$ which implies that $H_\ell, \mathsf{vis} \models VisBasic(\alpha)$.

□

Lemma 170 (Monotonicity of *RelTerms*)**.** *Let $H = (\mathcal{O}, \mathsf{so})$ be a history and let vis and vis' be two visibility relation over H such that $\mathsf{vis} \subseteq \mathsf{vis}'$. Then for any term $\tau \in RelTerms$, $\tau[so := \mathsf{so}, vis := \mathsf{vis}] \subseteq \tau[so := \mathsf{so}, vis := \mathsf{vis}']$.*

Proof. We shall write $\tau[\mathsf{so}, \mathsf{vis}]$ to mean $\tau[so := \mathsf{so}, vis := \mathsf{vis}]$ and $\tau[so, \mathsf{vis}']$ to mean $\tau[so := \mathsf{so}, vis := \mathsf{vis}']$.

We will prove this by induction over the number of compositions in the term τ. The base case is when there are no compositions. We have two cases $\tau = so$ and $\tau = vis$.

In the former case, the result trivially follows. In the latter case, the result follows since it is given that $\mathsf{vis} \subseteq \mathsf{vis}'$.

Suppose the result holds for all τ with fewer than n compositions. We now consider a $\tau = \tau'; \tau''$ where both τ' and τ'' have at most $n-1$ compositions. Now $\tau[\mathsf{so}, \mathsf{vis}] = \tau'[\mathsf{so}, \mathsf{vis}]; \tau''[\mathsf{so}, \mathsf{vis}]$. By induction hypothesis, $\tau'[\mathsf{so}, vis] \subseteq \tau'[\mathsf{so}, \mathsf{vis}']$ and $\tau''[\mathsf{so}, vis] \subseteq \tau''[\mathsf{so}, \mathsf{vis}']$. Since $A \subseteq A'$ and $B \subseteq B'$ implies $A; B \subseteq A'; B'$ we can conclude that $\tau'[\mathsf{so}, \mathsf{vis}]; \tau''[\mathsf{so}, \mathsf{vis}] \subseteq \tau'[\mathsf{so}, \mathsf{vis}']; \tau''[\mathsf{so}, \mathsf{vis}'] = \tau[\mathsf{so}, \mathsf{vis}']$. Thus the result is true for a τ with n compositions.

Hence, the result is true for all $\tau \in RelTerms$ □

Lemma 171 (Minimality of MinVisOne)**.** *Let vis_ℓ be a visibility relation over the history H_ℓ. Let α_ℓ be axioms defining the consistency criteria. Let vis' be a visibility relation over H_ℓ such that $\mathsf{vis}_\ell \subseteq \mathsf{vis}'$ and $H_\ell, \mathsf{vis}' \models VisBasic(\alpha_\ell)$.*

Then if, $\mathsf{vis} := \mathsf{MinVisOne}(H_\ell, \mathsf{vis}_\ell, \alpha_\ell)$, we have $\mathsf{vis} \subseteq \mathsf{vis}'$.

Proof. As before, we will denote the value at the end of the i^{th} iteration of the outer while-loop as $\mathsf{vis}_\mathsf{n}^i$. We shall denote the value of vis_n at the end of the the j^{th} iteration in the i^{th} iteration of the inner for loop as $\mathsf{vis}_\mathsf{n}^{i,j}$. We set $\mathsf{vis}_\mathsf{n}^0 = \mathsf{vis}_\mathsf{n}^{0,0} = \mathsf{vis}_\ell$.

Let $|RelTerms(\alpha)| = n$ and let τ_j denote the j^{th} member of $RelTerms(\alpha)$.

We will first show that for $j \in [1, \ldots, n]$, If $\mathsf{vis}_\mathsf{n}^{i,j-1} \subseteq \mathsf{vis}'$ then $\mathsf{vis}_\mathsf{n}^{i,j} \subseteq \mathsf{vis}'$. Note that $\mathsf{vis}_\mathsf{n}^{i,j} = \mathsf{vis}_\mathsf{n}^{i,j-1} \cup \tau_j[\mathsf{so}_\ell, \mathsf{vis}_\mathsf{n}^{i,j-1}]$. By assumption, $\mathsf{vis}_\mathsf{n}^{i,j-1} \subseteq \mathsf{vis}'$. By lemma 170, $\tau_j[\mathsf{so}_\ell, \mathsf{vis}_\mathsf{n}^{i,j-1}] \subseteq \tau_j[\mathsf{so}_\ell, \mathsf{vis}']$. Thus, we can conclude that $\mathsf{vis}_\mathsf{n}^{i,j} \subseteq \mathsf{vis}'$.

Since for any i, $\mathsf{vis}_\mathsf{n}^{i,0} \subseteq \mathsf{vis}_\mathsf{n}^{i,1} \subseteq \cdots \subseteq \mathsf{vis}_\mathsf{n}^{i,n} = \mathsf{vis}_\mathsf{n}^i$, we can conclude that if $\mathsf{vis}_\mathsf{n}^{i,0} \subseteq \mathsf{vis}'$ then, $\mathsf{vis}_\mathsf{n}^i \subseteq \mathsf{vis}'$. Finally note that $\mathsf{vis}_\mathsf{n}^i = \mathsf{vis}_\mathsf{n}^{i+1,0}$. Thus, if $\mathsf{vis}_\mathsf{n}^i \subseteq \mathsf{vis}'$ then $\mathsf{vis}_\mathsf{n}^{i+1} \subseteq \mathsf{vis}'$. Finally we note that $\mathsf{vis}_\mathsf{n}^{0,0} = \mathsf{vis}_\ell \subseteq \mathsf{vis}'$. Thus for all $i > 0$, $\mathsf{vis}_\mathsf{n}^i \subseteq \mathsf{vis}'$. Since the value vis returned by MinVisOne is the value of vis_n at the end of some iteration i, it follows that $\mathsf{vis} \subseteq \mathsf{vis}'$. □

Lemma 172 (Correctness of MinVisMulti)**.** *Let $H = (\mathcal{O}, \mathsf{so})$ be a hybrid history and let vis_wk and vis_st respectively be visibility relations over H_wk and H_st. Let ψ be a subformula in φ.*

Let $(\mathsf{vis}_\mathsf{wk}^{Ret}, \mathsf{vis}_\mathsf{st}^{Ret}) = \mathsf{MinVisMulti}(\mathcal{O}, \mathsf{so}, \mathsf{vis}_\mathsf{wk}, \mathsf{vis}_\mathsf{st}, \psi)$.

Then, $\mathsf{vis}_\mathsf{wk} \subseteq \mathsf{vis}_\mathsf{wk}^{Ret}$, $\mathsf{vis}_\mathsf{st} \subseteq \mathsf{vis}_\mathsf{st}^{Ret}$ and $H, \mathsf{vis}_\mathsf{wk}^{Ret}, \mathsf{vis}_\mathsf{st}^{Ret} \models \psi$.

Proof. Note that if $\psi \in \{\psi_{back}^{write}, \psi_{thru}^{read}\}$, we set $\mathsf{vis}_\mathsf{wk}^\mathsf{Ret} = \mathsf{vis}_\mathsf{wk}$ and $\mathsf{vis}_\mathsf{st}^\mathsf{Ret} = \mathsf{vis}_\mathsf{st}$. Since in this case $\psi = \top$, since trivially $H, \mathsf{vis}_\mathsf{wk}^\mathsf{Ret}, \mathsf{vis}_\mathsf{st}^\mathsf{Ret} \models \top$, the lemma is proved for these cases.

We now prove the result for the cases when $\psi = \psi_{thru}^{write}$.

For $\psi = \psi_{thru}^{write}$, we note that $\ell = \mathsf{st}$ and $\ell' = \mathsf{wk}$ and $\mathsf{vis}_\mathsf{st}^\mathsf{o} = \mathsf{vis}_\mathsf{st}$ and $\mathsf{vis}_\mathsf{wk}^\mathsf{o} = \mathsf{vis}_\mathsf{wk}$.

Now $\mathsf{vis}_\mathsf{st}^\mathsf{n} = \mathsf{vis}_\mathsf{st}^\mathsf{o} \cup (\mathsf{vis}_\mathsf{wk}^\mathsf{o}; \mathsf{so}) \restriction_\mathsf{st}$ and $\mathsf{vis}_\mathsf{wk}^\mathsf{n} = \mathsf{vis}_\mathsf{wk}^\mathsf{o}$. Thus, we can rewrite this as $\mathsf{vis}_\mathsf{st}^\mathsf{n} = \mathsf{vis}_\mathsf{st}^\mathsf{o} \cup (\mathsf{vis}_\mathsf{wk}^\mathsf{n}; \mathsf{so}) \restriction_\mathsf{st}$. Thus, $(\mathsf{vis}_\mathsf{wk}^\mathsf{n}; \mathsf{so}) \restriction_\mathsf{st} \subseteq \mathsf{vis}_\mathsf{st}^\mathsf{n}$. Hence, we can write that $\mathsf{so}, \mathsf{vis}_\mathsf{wk}^\mathsf{n}, \mathsf{vis}_\mathsf{st}^\mathsf{n} \models \psi_{thru}^{write}$.

The case where $\psi = \psi^{read}_{back}$ is proved with similar reasoning by interchanging wk and st.

□

Lemma 173 (Minimality of MinVisMulti)**.** *Let $H = (\mathcal{O}, \mathsf{so})$ be a hybrid history and let $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ be visibility relations over histories H_{wk} and H_{st}. Let ψ be a subformula in the hybrid constraint φ.*

Suppose there exists $\mathsf{vis}'_{\mathsf{wk}}$ and $\mathsf{vis}'_{\mathsf{st}}$ over H_{wk} and H_{st} respectively such that

- $\mathsf{vis}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$
- $\mathsf{vis}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$
- $H, \mathsf{vis}'_{\mathsf{wk}}, \mathsf{vis}'_{\mathsf{st}} \models \psi$.

Then, if $(\mathsf{vis}^{Ret}_{\mathsf{wk}}, \mathsf{vis}^{Ret}_{\mathsf{st}}) = \mathsf{MinVisMulti}(\mathcal{O}, \mathsf{so}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}}, \psi)$, it is the case that $\mathsf{vis}^{Ret}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$ and $\mathsf{vis}^{Ret}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$

Proof. When $\psi \in \{\psi^{write}_{back}, \psi^{read}_{thru}\}$, since $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{wk}} = \mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{st}} = \mathsf{vis}_{\mathsf{st}}$, it follows that $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$.

We will now prove the result for the case when $\psi = \psi^{write}_{thru}$.

Suppose ψ is ψ^{write}_{thru}. We have $\mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}} = \mathsf{vis}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{o}}_{\mathsf{st}} = \mathsf{vis}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$

Since $\mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$, we have $\mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}}; \mathsf{so} \subseteq \mathsf{vis}'_{\mathsf{wk}}; \mathsf{so}$. From this, we have $(\mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}}; \mathsf{so}) \restriction_{\mathsf{st}} \subseteq (\mathsf{vis}'_{\mathsf{wk}}; \mathsf{so}) \restriction_{\mathsf{st}}$. This implies $\mathsf{vis}^{\mathsf{o}}_{\mathsf{st}} \cup (\mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}}; \mathsf{so}) \restriction_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}} \cup (\mathsf{vis}'_{\mathsf{wk}}; \mathsf{so}) \restriction_{\mathsf{st}}$ since $\mathsf{vis}^{\mathsf{o}}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$. Since $H, \mathsf{vis}'_{\mathsf{wk}}, \mathsf{vis}'_{\mathsf{st}} \models \psi^{write}_{thru}$ implies $(\mathsf{vis}'_{\mathsf{wk}}; \mathsf{so}) \restriction_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$ we can conclude that
$\mathsf{vis}^{\mathsf{o}}_{\mathsf{st}} \cup (\mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}}; \mathsf{so}) \restriction_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$. However $\mathsf{vis}^{\mathsf{o}}_{\mathsf{st}} \cup (\mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}}; \mathsf{so}) \restriction_{\mathsf{st}} = \mathsf{vis}^{\mathsf{n}}_{\mathsf{st}}$. Thus $\mathsf{vis}^{\mathsf{n}}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$. Since $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{st}} = \mathsf{vis}^{\mathsf{n}}_{\mathsf{st}}$ and $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{wk}} = \mathsf{vis}_{\mathsf{wk}}$, it follows that $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$ and $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$. Hence this case is proved.

The proof for the case $\psi = \psi^{read}_{back}$ follows via similar reasoning by interchanging wk and st.

□

Lemma 174 (Correctness of ComputeStableExt)**.** *Let H be a hybrid history and let $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ respectively be a visibility relations over H_{wk} and H_{st}. Let α_w and α_s respectively be the weak and strong consistency criteria and let φ be the multilevel constraints. Let $(\mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}})$ be the return value obtained from $\mathsf{ComputeStableExt}(\mathcal{O}, \mathsf{so}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}}, \alpha_w, \alpha_s, \varphi)$. Then*

- $H_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}} \models \mathit{VisBasic}(\alpha_w)$
- $H_{\mathsf{st}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \models \mathit{VisBasic}(\alpha_s)$
- $H, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \models \varphi$

Proof. We note that the value returned by $\mathsf{ComputeStableExt}$ $(\mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}})$ are respectively the values of variables $\mathsf{vis}^{\mathsf{n}}_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{n}}_{\mathsf{st}}$ at the end of the outer while loop, when they respectively match the values $\mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{o}}_{\mathsf{st}}$. Furthermore, $\mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{o}}_{\mathsf{st}}$ were the values of $\mathsf{vis}^{\mathsf{n}}_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{n}}_{\mathsf{st}}$ at the end of the previous iteration of the outer **while**-loop.

We will replay the iteration of the outer-while loop which returned the value. Here, we note that $\mathsf{vis}^{\mathsf{p}}_{\mathsf{wk}} = \mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{p}}_{\mathsf{st}} = \mathsf{vis}^{\mathsf{o}}_{\mathsf{st}}$.

Let the value computed in line 7 by invoking the method $\mathsf{MinVisOne}$ be denoted as $\mathsf{vis}^{1}_{\mathsf{wk}}$. Now $\mathsf{vis}^{1}_{\mathsf{wk}} \subseteq \mathsf{vis}^{\mathsf{p}}_{\mathsf{wk}} = \mathsf{vis}^{\mathsf{o}}_{\mathsf{wk}}$. Further, by Lemma 169 $H_{\mathsf{wk}}, \mathsf{vis}^{1}_{\mathsf{wk}} \models \mathit{VisBasic}(\alpha_w)$.

Let the value computed in line 9 by invoking the method $\mathsf{MinVisOne}$ be denoted as $\mathsf{vis}^{1}_{\mathsf{st}}$. Now $\mathsf{vis}^{1}_{\mathsf{st}} \subseteq \mathsf{vis}^{\mathsf{p}}_{\mathsf{st}} = \mathsf{vis}^{\mathsf{o}}_{\mathsf{st}}$. Further, $H_{\mathsf{st}}, \mathsf{vis}^{1}_{\mathsf{st}} \models \mathit{VisBasic}(\alpha_s)$.

Let $\varphi = \psi_2 \wedge \psi_3$

We let $(\mathsf{vis}^{i}_{\mathsf{wk}}, \mathsf{vis}^{i}_{\mathsf{st}}) = \mathsf{MinVisMulti}(\mathcal{O}, \mathsf{so}, \mathsf{vis}^{i-1}_{\mathsf{wk}}, \mathsf{vis}^{i-1}_{\mathsf{st}}, \psi_i)$ for $i \in [2, 3]$

By lemma 172, for $i \in [2, 3]$, we have

- $\mathsf{vis}^{i-1}_{\mathsf{wk}} \subseteq \mathsf{vis}^{i}_{\mathsf{wk}}$
- $\mathsf{vis}^{i-1}_{\mathsf{st}} \subseteq \mathsf{vis}^{i}_{\mathsf{st}}$
- $H, \mathsf{vis}^{i}_{\mathsf{wk}}, \mathsf{vis}^{i}_{\mathsf{st}} \models \psi_i$

And $\mathsf{vis}^{3}_{\mathsf{wk}} = \mathsf{vis}^{\mathsf{n}}_{\mathsf{wk}}$, $\mathsf{vis}^{3}_{\mathsf{st}} = \mathsf{vis}^{\mathsf{n}}_{\mathsf{st}}$

Thus, for $\ell \in \{\mathsf{wk}, \mathsf{st}\}$ we have $\mathsf{vis}^{\mathsf{o}}_{\ell} \subseteq \mathsf{vis}^{1}_{\ell} \subseteq \mathsf{vis}^{2}_{\ell} \subseteq \ldots \mathsf{vis}^{\mathsf{n}}_{\ell} = \mathsf{vis}^{\mathsf{o}}_{\ell}$.

From this we can conclude that $\mathsf{vis}^{i}_{\ell} = \mathsf{vis}^{\mathsf{n}}_{\ell} = \mathsf{vis}^{\mathsf{stb}}_{\ell}$ for $i \in [1, 2, 3]$.

Since

- $H_{\mathsf{wk}}, \mathsf{vis}^{1}_{\mathsf{wk}} \models \mathit{VisBasic}(\alpha_w)$.
- $H_{\mathsf{st}}, \mathsf{vis}^{1}_{\mathsf{st}} \models \mathit{VisBasic}(\alpha_s)$.
- $H, \mathsf{vis}^{2}_{\mathsf{wk}}, \mathsf{vis}^{2}_{\mathsf{st}} \models \psi_2$
- $H, \mathsf{vis}^{3}_{\mathsf{wk}}, \mathsf{vis}^{3}_{\mathsf{st}} \models \psi_3$

we can conclude that

- $H_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}} \models \mathit{VisBasic}(\alpha_w)$.
- $H_{\mathsf{st}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \models \mathit{VisBasic}(\alpha_s)$.
- $H, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \models \psi_2 \wedge \psi_3 = \varphi$

This proves the result. □

Lemma 175 (Minimality of ComputeStableExt)**.** *Let H be a hybrid history and let $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ respectively be a visibility relations over H_{wk} and H_{st}. Let α_w, α_s be weak and strong consistency criteria and let φ be multilevel constraints.Let*
$(\mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}}) := \mathsf{ComputeStableExt}(\mathcal{O}, \mathsf{so}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}}, \alpha_w, \alpha_s, \varphi)$*. If there exists visibility relations $\mathsf{vis}'_{\mathsf{wk}}$ and $\mathsf{vis}'_{\mathsf{st}}$ over H_{wk} and H_{st} respectively such that*

- $\mathsf{vis}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$
- $\mathsf{vis}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$
- $H_{\mathsf{wk}}, \mathsf{vis}'_{\mathsf{wk}} \models \mathit{VisBasic}(\alpha_w)$
- $H_{\mathsf{st}}, \mathsf{vis}'_{\mathsf{st}} \models \mathit{VisBasic}(\alpha_s)$
- $H, \mathsf{vis}'_{\mathsf{wk}}, \mathsf{vis}'_{\mathsf{st}} \models \varphi$

Then $\mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$

Proof. The proof for this follows the line of argument showing the minimality of MinVisOne. We note that at each step we compute extensions of the weak and strong visibility relations via invoking MinVisOne and MinVisMulti.

From Lemmas 171 and 173, the output produced by these procedures $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{st}}$ from inputs $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$ respectively will satisfy $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{Ret}}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$ whenever it is the case that $\mathsf{vis}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$.

Thus, even the final output $(\mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}})$ will satisfy the containment. □

We shall prove another interesting result pertaining to the conflict relations of two visibility relations extending the same *reads-from* relations, with one visibility relation contained inside another.

Lemma 176. *Let rf_ℓ be a* reads-from *relation over the history H_ℓ and let vis_ℓ and vis'_ℓ be two visibility relations over H_ℓ, both extending rf_ℓ such that $\mathsf{vis}_\ell \subseteq \mathsf{vis}'_\ell$. Then, $\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell) \subseteq (\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}'_\ell) \cup (\mathsf{vis}'_\ell \restriction_{\mathsf{Write}})^+$*

Proof. Suppose $(o'', o') \in \mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}_\ell)$. That implies that there exists a read operation o such that $o'', o' \in \mathit{MaxRelWrites}_{\mathsf{vis}_\ell}(o)$ and $o' = \mathsf{rf}_\ell^{-1}(o)$.

Since $\mathsf{vis}_\ell \subseteq \mathsf{vis}'_\ell$, it implies that $o'', o' \in \mathit{RelWrites}_{\mathsf{vis}'_\ell}(o)$.

We consider two cases.

Suppose $o'' \in \mathit{MaxRelWrites}_{\mathsf{vis}'_\ell}(o)$, then by definition, $(o'', o') \in \mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}'_\ell)$. Therefore, in this case $(o'', o') \in (\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}'_\ell) \cup (\mathsf{vis}'_\ell \restriction_{\mathsf{Write}})^+$.

Suppose $o'' \notin \mathit{MaxRelWrites}_{\mathsf{vis}'_\ell}(o)$. Then, this implies that o'' is not a maximal write in the vis'_ℓ view of o restricted to its related writes. Thus, either $o'' \xrightarrow{\mathsf{vis}'_\ell \restriction_{\mathsf{Write}}} o'$ or there exists

a path from $o'' \xrightarrow{\mathsf{vis}'_\ell \restriction \mathsf{Write}} o_1 \xrightarrow{\mathsf{vis}'_\ell \restriction \mathsf{Write}} \ldots \xrightarrow{\mathsf{vis}'_\ell \restriction \mathsf{Write}} o_k \xrightarrow{\mathsf{vis}'_\ell \restriction \mathsf{Write}} o'''$ where $o''' \in \mathit{MaxRelWrites}_{\mathsf{vis}'_\ell}(o)$ and each of
$o_1, \ldots, o_k \in \mathit{RelWrites}_{\mathsf{vis}'_\ell}(o)$. In this case too, either $o''' = o'$ or $(o''', o') \in \mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}'_\ell)$. Thus even in this case $(o'', o') \in (\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}'_\ell) \cup \mathsf{vis}'_\ell \restriction_{\mathsf{Write}})^+$. □

With this we can now prove the correctness of Theorem 166. We need to prove the following:

For a hybrid read-write history $H = (\mathcal{O}, \mathsf{so})$, weak and strong consistency criteria α_w, α_s and multilevel constraints φ, the procedure
TestMultiCorrect returns a witness $(\mathsf{rf}, \mathsf{vis}_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{st}})$ over H iff H is multi-level correct with respect to α_w, α_s and φ.

Proof. Suppose the hybrid history H is multi-level correct with respect to the consistency criteria α_w, α_s, and multilevel constraints φ. Then, by theorem 164, there exists a *reads-from* relation rf and visibility relations $\mathsf{vis}'_{\mathsf{wk}}$ and $\mathsf{vis}'_{\mathsf{st}}$ over H_{wk} and H_{st} extending $\mathsf{rf}_{\mathsf{wk}}$ and $\mathsf{rf}_{\mathsf{st}}$ respectively such that

- $H_{\mathsf{wk}}, \mathsf{vis}'_{\mathsf{wk}} \models \alpha_w$
- $H_{\mathsf{st}}, \mathsf{vis}'_{\mathsf{st}} \models \alpha_s$
- $H, \mathsf{vis}'_{\mathsf{wk}}, \mathsf{vis}'_{\mathsf{st}} \models \varphi$

Since the procedure, iterates through all possible *reads-from* relation, if it returns before encountering the rf mentioned earlier, then we have nothing to prove. Suppose it does not return. Then, we will consider the iteration with the *reads-from* relation being rf.

Note that since $\mathsf{vis}^{\min}_{\mathsf{wk}}$ and $\mathsf{vis}^{\min}_{\mathsf{st}}$ are extensions of $\mathsf{rf}_{\mathsf{wk}}$ and $\mathsf{rf}_{\mathsf{st}}$ via the procedure MinVisOne, by Lemma 171, we have $\mathsf{vis}^{\min}_{\mathsf{wk}} \subseteq \mathsf{vis}'_{\mathsf{wk}}$ and $\mathsf{vis}^{\min}_{\mathsf{st}} \subseteq \mathsf{vis}'_{\mathsf{st}}$.

Now, suppose for $\mathsf{total}(\mathit{vis})$ is a subformula in α_w. Then $\mathsf{vis}'_{\mathsf{wk}}$ is a total order. Similarly if $\mathsf{total}(\mathit{vis})$ is a subformula in α_s, then $\mathsf{vis}'_{\mathsf{st}}$ is a total order.

For $\ell \in \{\mathsf{wk}, \mathsf{st}\}$,since we iterate through all the total orders extending $\mathsf{vis}^{\min}_\ell$, if the procedure returns before the iteration reaches vis'_ℓ, then, there is nothing to prove. Suppose, the procedure returns with none of the prior total orders extending $\mathsf{vis}^{\min}_\ell$. Then we consider the case where the iterating variable vis_ℓ is the total order vis'_ℓ.

On the other hand, if $\mathsf{total}(\mathit{vis})$ is not a subformula in α_w or α_s, then we would set the corresponding vis_ℓ to $\mathsf{vis}^{\min}_\ell$. In both these cases, we can notice that $\mathsf{vis}_\ell \subseteq \mathsf{vis}'_\ell$.

Now, we obtain $(\mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}})$ by invoking ComputeStableExt with $\mathsf{vis}_{\mathsf{wk}}$ and $\mathsf{vis}_{\mathsf{st}}$. By Lemma 174, $H_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}} \models \mathit{VisBasic}(\alpha_w)$ $H_{\mathsf{st}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \models \mathit{VisBasic}(\alpha_s)$ and $H, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \models \varphi$.

Further, by Lemma 175, for $\ell \in \{\mathsf{wk}, \mathsf{st}\}$, $\mathsf{vis}^{\mathsf{stb}}_\ell \subseteq \mathsf{vis}'_\ell$. Which implies that if $\mathsf{total}(\mathit{vis})$ is a subformula in the ℓ-consistency criteria then, $\mathsf{vis}^{\mathsf{stb}}_\ell$ is a total order as vis'_ℓ is.

From this, we can conclude that

- $H_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}} \models \alpha_w$,
- $H_{\mathsf{st}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \models \alpha_s$
- $H, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \models \varphi$.

Now we check $H, \mathsf{rf}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}}$ for bad patterns.

Note that, $(H, \mathsf{rf}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}})$ cannot have BADVISIBILITY, THINAIR, BADINITREAD or BADREAD bad patterns, since that would imply the existence of those bad patterns in $(H, \mathsf{rf}, \mathsf{vis}'_{\mathsf{wk}}, \mathsf{vis}')$ since $\mathsf{vis}^{\mathsf{stb}}_{\ell}$ is contained within vis'_{ℓ} for $\ell \in \{\mathsf{wk}, \mathsf{st}\}$.

We will show by contradiction that BADARB bad pattern doesn't exist for $(H, \mathsf{rf}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}})$ doesn't exist. Suppose this bad pattern did exist. Then, there is a cycle $C = o_1 \xrightarrow{\sigma_1} o_2 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_n} o_1$ where each σ_i is one of $\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}^{\mathsf{stb}}_{\ell})$ or $\mathsf{vis}^{\mathsf{stb}}_{\ell} \restriction_{\mathsf{Write}}$ for $\ell \in \{\mathsf{wk}, \mathsf{st}\}$

Note that since $\mathsf{vis}^{\mathsf{stb}}_{\ell} \subseteq \mathsf{vis}'_{\ell}$ we have $\mathsf{vis}^{\mathsf{stb}}_{\ell} \restriction_{\mathsf{Write}} \subseteq \mathsf{vis}'_{\ell} \restriction_{\mathsf{Write}}$. Hence in the Cycle C above, we can rewrite the edge $o_i \xrightarrow{\mathsf{vis}^{\mathsf{stb}}_{\ell} \restriction_{\mathsf{Write}}} o_{i+1}$ by $o_i \xrightarrow{\mathsf{vis}'_{\ell} \restriction_{\mathsf{Write}}} o_{i+1}$.

Further from Lemma 176, we have $\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}^{\mathsf{stb}}_{\ell}) \subseteq (\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}'_{\ell}) \cup (\mathsf{vis}'_{\ell} \restriction_{\mathsf{Write}})^+$. Which means that the any edge $o_i \xrightarrow{\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}^{\mathsf{stb}}_{\ell})} o_{i+1}$ in the cycle C can be replaced by a path $o_i \xrightarrow{\sigma'_1} \ldots \xrightarrow{\sigma'_{n'}} o_{i+1}$ where each σ'_k is either $\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}'_{\ell})$ or $\mathsf{vis}'_{\ell} \restriction_{\mathsf{Write}}$. Thus, we get a cycle C' from C whose edges comprise of $\mathsf{CF}(\mathsf{rf}_\ell, \mathsf{vis}'_{\ell})$ and $\mathsf{vis}'_{\ell} \restriction_{\mathsf{Write}}$ for $\ell \in \{\mathsf{wk}, \mathsf{st}\}$. Thus, BADARB bad pattern exists for $(H, \mathsf{rf}, \mathsf{vis}'_{\mathsf{wk}}, \mathsf{vis}'_{\mathsf{st}})$, which is a contradiction. Thus, if H is correct, then we have proved that the procedure TestMultiCorrect produces a satisfying witness.

Conversely we will show that if TestMultiCorrect produces a satisfying witness $(\mathsf{rf}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}})$ then the hybrid history H is multi-level correct.

Suppose $(\mathsf{rf}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}})$ is the witness. Then, $\mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}$ and $\mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}}$ are the visibility relations returned by the procedure ComputeStableExt. Further, none of the bad patterns exist for $(H, \mathsf{rf}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}})$.

By lemma174, we know that

- $H_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}} \models \mathit{VisBasic}(\alpha_w)$
- $H_{\mathsf{st}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \models \mathit{VisBasic}(\alpha_s)$
- $H, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{wk}}, \mathsf{vis}^{\mathsf{stb}}_{\mathsf{st}} \models \varphi$.

For $\ell \in \{\mathsf{wk}, \mathsf{st}\}$, in order to show that $H_\ell, \mathsf{vis}_\ell \models \alpha_\ell$, we need to show that if $\mathsf{total}(\mathit{vis})$ is a subformula of α_ℓ, then, $\mathsf{vis}^{\mathsf{stb}}_{\ell}$ is a total order

Note that if $\mathsf{total}(\mathit{vis})$ is a subformula of α_ℓ, then the iterating variable vis_ℓ would have been a total order (line 71). By lemma 174, we know that $\mathsf{vis}_\ell \subseteq \mathsf{vis}^{\mathsf{stb}}_{\ell}$. Suppose $\mathsf{vis}_\ell \subsetneq \mathsf{vis}^{\mathsf{stb}}_{\ell}$, it implies that $\mathsf{vis}^{\mathsf{stb}}_{\ell}$ has at least one additional edges between the operations of $\mathcal{O}_\ell$ over what is present in vis_ℓ. However, since vis_ℓ is a total order, it implies that any additional edges introduce a cycle in $\mathsf{vis}^{\mathsf{stb}}_{\ell}$. But this is not the case since that would imply BADVISIBILITY for

$\mathsf{vis}_\ell^{\mathsf{stb}}$. Hence it has to be the case that $\mathsf{vis}_\ell^{\mathsf{stb}} = \mathsf{vis}_\ell$. Thus, $\mathsf{vis}_\ell^{\mathsf{stb}}$ is a total order. This proves that if total is a subformula in the consistency criteria for level ℓ, then, $H_\ell, \mathsf{vis}_\ell^{\mathsf{stb}} \models \mathsf{total}(vis)$. Hence $H_\ell, \mathsf{vis}_\ell^{\mathsf{stb}} \models \alpha_\ell$.

Thus, we can conclude that there exists a *reads-from* relation rf and weak and strong visibility relations $\mathsf{vis}_{\mathsf{wk}}^{\mathsf{stb}}$ and $\mathsf{vis}_{\mathsf{st}}^{\mathsf{stb}}$ extending $\mathsf{rf}_{\mathsf{wk}}$ and $\mathsf{rf}_{\mathsf{st}}$ respectively such that $H_{\mathsf{wk}}, \mathsf{vis}_{\mathsf{wk}}^{\mathsf{stb}} \models \alpha_w$, $H_{\mathsf{st}}, \mathsf{vis}_{\mathsf{st}}^{\mathsf{stb}} \models \alpha_s$, $H, \mathsf{vis}_{\mathsf{wk}}^{\mathsf{stb}}, \mathsf{vis}_{\mathsf{st}}^{\mathsf{stb}} \models \varphi$, and none of the bad patterns exist for $(H, \mathsf{rf}, \mathsf{vis}_{\mathsf{wk}}^{\mathsf{stb}}, \mathsf{vis}_{\mathsf{st}}^{\mathsf{stb}})$. By theorem 164, this implies that the hybrid history H is multi-level correct with respect to $\alpha_w, \alpha_s, \varphi$. □

7.6.1 Complexity

Suppose $H = (\mathcal{O}, \mathsf{so})$ is history with $|\mathcal{O}| = N$.

We note that in the procedure $\mathsf{ComputeStableExt}$, at the end of every iteration of the outer **while**-loop, the values of $\mathsf{vis}_{\mathsf{wk}}^{\mathsf{n}}$ and $\mathsf{vis}_{\mathsf{st}}^{\mathsf{n}}$ monotonically increase from the end of the previous iteration. Since they are binary relations over finite history $H = (\mathcal{O}, \mathsf{so})$ their size is upper bounded by $O(N^2)$. The time taken to evaluate each term in $RelTerms(\alpha_\ell)$ is again polynomial in N. Hence, the time-complexity of $\mathsf{ComputeStableExt}$ is polynomial in N, say $f(N)$.

We can observe from the procedure $\mathsf{TestMultiCorrect}$ that the main part that adds to the complexity is iterating through all the *reads-from* relation, as well as the total orders if α_w or α_s contain the subformula $\mathsf{total}(vis)$. Suppose the number of read operations are k. Then the number of write operations is $N - k$, and there are $O((N-k)^k)$ *reads-from* relations. Since $k = O(N)$, this can be bound by $O(2^{N \log N})$. Furthermore, for a given rf, if any of the levels $\ell \in \{\mathsf{wk}, \mathsf{st}\}$ require that the visibility relation be a total order, then we iterate over all the total-orders containing the minimal visibility relation extending rf. Iterating through this requires time bounded by $O(2^{N \log N})$. Thus the worst case time complexity of the procedure is $O(f(N).2^{N \log N})$.

In general, the problem of testing the correctness of a hybrid history is in NP. We need to guess the *reads-from* relation, and then, extend it to obtain the minimal visibility relations satisfying the visibility constraints of the wk and the st consistency criteria. If the visibility relation is required to be a total order, we can guess the order. Extending this to derive fixed-point minimal visibility relations that satisfy all the visibility constraints via $\mathsf{ComputeStableExt}$ requires polynomial time. Subsequently checking for each of the bad-patterns requires polynomial time.

Note that we can reduce the testing of the correctness of a regular history (that contains only a single level of Read and Write operations) with respect to consistency criterion α to this procedure by defining the level of all the read operations to st. We set α_s to α, α_w to $\top$, and φ to $\psi_{back}^{write} \wedge \psi_{thru}^{read}$. For any *reads-from* relation rf, $\mathsf{rf}_{\mathsf{wk}} = \emptyset$. Thus $\mathsf{vis}_{\mathsf{wk}} = \emptyset$, trivially satisfying α_w as well as φ. Thus, the lower bound for testing the correctness of the hybrid

history H is the complexity of testing the correctness of the H_{wk} and H_{st} with respect to their respective consistency criteria. It has been shown in [Furbach et al., 2014] that testing the correctness of a read-write history with respect to sequential consistency is NP-COMPLETE. In [Bouajjani et al., 2017], the authors use the same reduction to show that testing the correctness with respect to causal consistency is NP-COMPLETE. However, it can be shown that the reduction works for any consistency criterion stronger than FIFO consistency, and checking correctness with respect to such a consistency criterion is NP-COMPLETE. Thus, in general, though testing the multi-level correctness of a hybrid history is a hard problem, the hardness is not due to the multilevel constraints but due to the constraints of the individual consistency criteria and the read-write specification.

In [Bouajjani et al., 2017], the authors identify the class of read-write data-stores known as *data-independent* data-stores whose behaviour is not dependent on the exact values written to the keys. Thus, for such stores, if there is a bad history, there is an equivalent bad *differentiated history* where a particular value is written to a particular memory location at most once. Thus, we can restrict our testing to only the correctness of differentiated histories. The authors show that the problem of testing the correctness of differentiated-histories with respect to causal consistency is solvable in polynomial time.

Note that for differentiated histories, there is exactly one *reads-from* relation which associates every Read operation with at most one Write operation which has written that value to the memory location read by the Read operation. Thus, if neither of α_w or α_s contain the subformula $\mathsf{total}(vis)$, the procedure TestMultiCorrect terminates in polynomial time. Thus, our procedure generalizes the result from [Bouajjani et al., 2017] to all the consistency criteria defined in terms of the set of formulas involving only visibility, but not totality constraints. Our procedure checks the multi-level correctness of hybrid histories where the individual consistency levels do not require the visibility relation to be a total order, in polynomial time.

On the other hand, if one of α_w or α_s contains $\mathsf{total}(vis)$, then the worst case complexity remains $O(2^{N \log N})$. Once again, this does not come as a surprise, since the problem of testing the correctness of a differentiated history w.r.t. sequential consistency is not known to have a polynomial time solution.

7.7 Related Work

There is prior work that illustrates the need for multiple levels of consistency provided by the distributed data-stores to provide a trade off between consistency and availability or latency [Guerraoui et al., 2016; Kraska et al., 2009; Bailis et al., 2013; Li et al., 2012].

The work by Kraska et al. [Kraska et al., 2009] provides a transactional paradigm that allows applications to define the consistency level on data instead of transactions, and also

allows the application to switch consistency guarantees at runtime. In the work by Guerraoui et al. [Guerraoui et al., 2016], the authors provide a generic library that allows applications to request multiple responses to the same query, where the response that comes later in time is *more-correct* than the prior responses. Thus, later responses are supposed to have more knowledge of the state of the system compared to earlier responses. In our work, we have defined multilevel constraints, which can model the requirement of incremental consistency guarantees by requiring that subsequent strong responses see the effects observed by prior weak responses.

Burckhardt [Burckhardt, 2014] provides a generic methodology for formalizing the specification of distributed data-stores in terms of histories, visibility and arbitration orders and provides an axiomatic characterization for consistency criteria. In our work, we have derived the specification for read-write stores based on this formalism. We have adapted this characterization to define consistency criteria as a conjunction of individual formulas. Our work extends [Burckhardt, 2014] in terms of the definition of hybrid histories and provides a definition of multi-level correctness for read-write stores.

There is prior work on verifying the correctness of a behaviour with respect to individual consistency criteria. Examples include [Bouajjani et al., 2014b], which deals with verifying the correctness with respect to eventual consistency, [Bouajjani and Emmi, 2013], which investigates the feasibility of checking a concurrent implementation with respect to a consistency criterion that has a sequential specification, including sequential consistency, linearizability and conflict-serializability and [Bouajjani et al., 2017], which focusses on correctness with respect to causal consistency. Our work provides a generic procedure for checking the correctness of read-write histories for all these individual consistency criteria. Further, [Bouajjani et al., 2017] show that verification of correctness of a history with respect to causal consistency is NP-COMPLETE. However, for differentiated histories, the problem is solvable in polynomial time. In our work, we generalize the technique of computing the minimal visibility relation and checking for the absence of bad patterns for all the consistency criteria defined using our syntax. In [Dongol and Hierons, 2016], the authors models quiescent consistency using Mazurkiewicz Trace Theory to model the notion of independence between the events prior to the quiescent point. They work shows that the testing problem (which they call the membership problem) for a history is NP-COMPLETE. We cannot model quiescent consistency in our framework since we cannot model quiescent point. In [Furbach et al., 2014], the authors present a detailed complexity analysis of the problem of testing the correctness of a history with respect to various consistency criteria. Our findings are consistent with the results from [Furbach et al., 2014] with respect to hardness of testing consistency criteria that require the visiblity relation to be a total order. In a recent work [Emmi and Enea, 2018], the authors provide a technique for testing the correctness of a history of a data-store with respect to a weak consistency criterion. That work also characterizes correctness in terms of minimal visibility relation extending the session

order (called program-order there) and the happened-before relation (called returns-before relation in [Burckhardt, 2014]). Our work applies this concept to read-write stores, where we observe that correctness with respect to visibility constraints can be satisfied by constructing a minimal visibility relation while the correctness with respect to read-write specifications and arbitration constraints can be reduced to checking for absence of certain bad patterns. In particular, our characterization of the arbitration relation in terms of the conflict relation saves the step of searching through all possible arbitration relations which is used in [Emmi and Enea, 2018].

[Gotsman et al., 2016] deals with verification of *red-blue* consistency where, in a history, a subset of operations are labelled *red* while the remaining are labelled *blue*. The *blue* operations are expected to satisfy a weaker consistency criterion, while the *red* operations are supposed to satisfy a stronger consistency criterion. The effects of the strong operations and weak operations are visible to each other. We can model this by setting $\varphi = \psi_{thru}^{write} \wedge \psi_{back}^{read}$.

Our work should also be contrasted with [Biswas et al., 2019], which addresses the problem of checking the consistency of CRDTs against their specifications, and covers a wide range of CRDTs including replicated sets, flags, counters, registers, etc. The relevant data structure in our case is registers, where the results are comparable (checking w.r.t. the weaker consistency criterion is tractable). However, we also consider registers with multiple consistency criteria in this chapter, which is not considered there.

Another related work is [Biswas and Enea, 2019], which uses the reads-from relation (called the *write-read* relation there) to show that testing the correctness of an execution (containing transactions) with respect to various consistency criteria like Read Committed (RC), Read Atomic (RA), Causal Consistency (CC), Prefix Consistency, and Snapshot Isolation. The key difference in the current work is that we consider histories having multiple consistency levels simultaneously while [Biswas and Enea, 2019] considers executions consisting of transactions, under a single consistency criterion.

8

Summary, Future Work and Conclusion

8.1 Summary

We started this thesis with an introduction to the widely studied class of distributed data types namely the *Conflict-free Replicated Data Types (CRDTs)*. We introduced the terminology and provided new rigorous proofs for the sufficient conditions for these data types to satisfy *Strong Eventual Consistency (SEC)* in Chapter 1.

We then studied a particular replicated data type, namely the *Observed Remove Set (OR Set)* where we presented two existing implementations and our novel optimization implementation that retained the best properties of the existing implementations in Chapter 3. We also provided a precise formal specification for the OR-Set which would capture its essence without relying on the network guarantees on delivery of updates. In this chapter, we introduced a novel object named *Interval Version Vector* which can be used to succinctly keep track of concurrent updates in a network where updates can be delivered out of order.

In the next chapter, we described a framework for providing declarative specifications of replicated data types using the concepts from labelled partial orders introduced originally in [Mazurkiewicz, 1987] on *Theory of Traces*. Using this framework, we provided the declarative specification for many well known CRDTs. We also showed a principled approach towards constructing a reference implementation for any replicated data type given its declarative specification. We then explored bounded reference implementations which can be used to formally verify the correctness of a given implementation. We showed two novel bounded reference implementations.

The first was was a distributed reference implementation, which can be constructed for replicated data types with bounded specifications. To achieve this, we generalize the *gossip problem* from [Mukund and Sohoni, 1997; Mukund et al., 2003]. We showed that the generalized gossip problem has a bounded solution when the runs *B-Concurrent*. At the heart of the bounded distributed reference implementation of a replicated data type

lies the bounded solution to the generalized gossip problem. The second bounded reference implementation was a *global reference implementation* where the state of every replica is represented as a *Later Appearance Record*, a concept introduced in [Gurevich and Harrington, 1982]. We show that for replicated data types with bounded specifications, when the runs are guaranteed to be *B-Bounded* by the underlying network, this global reference implementation is a bounded implementation.

In the last part of the thesis, we studied read-write data stores which provide the option for the users to tag Read operations with a particular consistency criteria, thus, allowing behaviours where multiple consistency criteria co-exist. We showed how such behaviours can be modelled, and we defined a correctness criteria for such behaviours with multiple consistency levels. Following that, we provided a bad-pattern characterization for read-write stores, which is a generalization of the bad-pattern characterization first introduced in [Bouajjani et al., 2017]. We also provide an algorithm to derive the minimal visibility relation extending a *reads-from* relation, which satisfy all the constraints of the consistency criteria offered by the data stores and also the multilevel constraint. This algorithm then decides whether a given behaviour of such data stores is correct with respect to the individual consistency criteria as well as the multilevel constraint.

8.2 Future Work

In the process of addressing some of the challenges mentioned in this thesis, we have come across new open problems which can be explored further as a part of the future work. We define three interesting problems below.

8.2.1 Bounded Interval Version Vectors

Our robust optimized implementation of the OR-Set uses *interval version vectors* to keep track of the elements that have already been seen. It is known that regular version vectors have a bounded representation when the replicas communicate using pairwise synchronization [Almeida et al., 2004]. An alternative proof of this in [Mukund et al., 2020] is based on the solution to the *gossip problem* for synchronous communication [Mukund and Sohoni, 1997], which has also been generalized to message-passing systems [Mukund et al., 2003]. In the case of regular version vectors, their purpose was to understand, if one replica had a more up-to-date information compared to another. Thus given a version vector of V_r and $V_{r'}$ for replicas r and r', we say that r is more up-to-date than r' if for every other replica r'',

$$V_r[i] \geq V_{r'}[i]$$

Since this was the only question of interest, it did not matter what the exact values of

$V_r[i]$ and $V_{r'}[i]$ were, but merely how are they related to each other. Hence, it was possible to reuse the labels and provide a bounded representation.

In the case of *interval version vectors* there are two questions that we are interested in. Given an interval version vector V at replica r, we want to know if r has seen an update associated with the integer c from replica r''. This is equivalent to checking if $c \in V[r'']$. The other question is, given a pair of interval version vectors V and V', if one of them is more up-to-date than the other. We say V is more up-to-date than V' iff

$$\forall r'' : V'[r''] \subseteq V[r'']$$

.

In the first case, it is difficult to imagine a bounded representation, since the integer c whose membership we want to check in $V[r'']$ itself could be arbitrarily large. However, in the second case, since we are not interested in the individual values but only the relative state of interval sequences from two interval version vectors, there is a possibility of exploring a bounded implementation. Even in this case, we recognize that since the number of undelivered updates could be unbounded in an arbitrarily long run, the number of intervals in each Interval sequence $V[r'']$ could be unbounded as well. However, if the underlying network guarantees that there won't be more than B undelivered messages in the network at any point in time, the number of intervals in the interval sequence $V[r'']$ are guaranteed to be bounded by $B + 1$. Thus in this scenario, it would be interesting to see if the ideas of reusing the timestamps using some form of generalized gossip can be applied to arrive at a bounded implementation of *interval version vectors*.

8.2.2 Complexity of testing the correctness of differentiated histories

In Chapter 7, we discussed the problem of testing the correctness of a history of a *read-write* Data store with respect to a given consistency criteria. In general this is a hard problem. It has been formally proven that testing the correctness a history with respect to causal consistency and sequential consistency is NP-COMPLETE. However, there is a class of histories known as the differentiated history, where a particular value is written to a variable x at most once. In our work we have shown that the problem of testing the correctness of a differentiated history is in PTIME if the visibility relation is not required to be a total order. However, the exact complexity of testing the correctness of such differentiated histories against stronger consistency criteria, such as *sequential consistency* or *linearizability* is not yet known. We believe that it cannot be done in PTIME but we do not know if this problem is NP-COMPLETE. The reason for this is that all prior known reductions, which reduce the SAT problem to the problem of testing the correctness of the *read-write* History, rely on the fact that the same value is written to the variable my multiple sessions. This property is not

available in the case of differentiated histories. Thus, in this problem is NP-COMPLETE, we need a novel reduction of some well known problem to the problem of testing the correctness of a differentiated history where the visibility relation is a total order. To our knowledge, no straightforward reduction exists as of now.

8.2.3 Generalizing Bad-Patterns and minimum visibility relation to other data types

In Chapter 7, we have provided a systematic methodology for deriving bad patterns characterizing a wide range of consistency criteria. However, except for the BADVISIBILITY pattern, which flags out visibility relations which have a cycle, the other four of the five bad patterns are specific to *read-write* data stores. In some sense these four bad patterns capture all the possible violations of the specification of the *read-write* stores. It would be worthwhile to explore if a similar finite bad-pattern characterization would be available for other replicated data type such as counters, sets and graphs.

Furthermore, in case of the *read-write* stores, we defined the *reads-from* relation, which is in some sense the smallest visibility relation over the *read-write* history, such that the history is correct as per the specification in the absence of any constraints imposed by the required consistency criteria. We then applied some fixed point computation to inflate this minimal visibility relation so that it would satisfy the constraints of the individual consistency criteria and their combinations via the multilevel constraints. It was on this inflated visibility relation, which we proved was a minimal visibility relation satisfying all the constraints, that we tested for the absence of bad patterns. It would be an interesting exercise to explore the existence of minimal visibility relations akin to the *reads-from* relation for other data types. For example in the case of OR-Sets, if $\mathsf{contains}(x)$ operation in the history returns True, then we know that there has to be an $\mathsf{add}(x)$ operation from the history which should be visible to that $\mathsf{contains}(x)$. On the other hand, if $\mathsf{contains}(x)$ returns False, then there are two options : Either no $\mathsf{add}(x)$ or $\mathsf{delete}(x)$ operation is visible to that $\mathsf{contains}(x)$, or only a $\mathsf{delete}(x)$ is visible to $\mathsf{contains}(x)$. On such a minimal visibility relation, we perform the fixed point computation to inflate it into a visibility relation that would satisfy all the constraints of the consistency criteria. On this inflated minimal visibility relation, we check for the presence of bad patterns corresponding to the specification of OR-Sets.

8.3 Conclusion

As mentioned in Chapter 1, one of the themes in this thesis has been to explore the applicability of concepts and tools from traditional trace theory and automata theory, and wherever required, adapt them and extend them in the study and analysis of relatively modern repli-

cated data types such as the CRDTs. We believe we have been successful in this effort. Some of the novel contributions in this thesis include

- *Interval Version Vectors*, which allow us to robustly keep track of concurrent updates in the absence of causal delivery. We use these to provide an optimized implementation of OR-set in the absence of causal-delivery.
- Formulation of a generalization of the *Gossip Problem* [Mukund and Sohoni, 1997; Mukund et al., 2003] and a *bounded solution* to this *Generalized Gossip Problem* that provides a principled approach towards synthesizing distributed reference implementations of CRDTs from their specifications.
- A technique for synthesizing a simple global reference implementation of CRDTs from their specification using the concept of *Later Appearance Records (LAR)* introduced in [McNaughton, 1965].
- A formal framework for defining the correctness of behaviours of read-write data-stores which provide multiple consistency levels.
- A *systematic methodology* for deriving bad patterns characterizing a wide range of consistency models and combinations thereof.
- An *effective algorithmic framework* for testing the behaviours of modern data-stores that providing multiple levels of consistency.

References

Almeida, J. B., Almeida, P. S., and Baquero, C. (2004). Bounded version vectors. In *DISC*, pages 102–116.

Almeida, P. S., Shoker, A., and Baquero, C. (2015). Efficient state-based crdts by delta-mutation. *ArXiv*, abs/1410.2803.

Attiya, H., Burckhardt, S., Gotsman, A., Morrison, A., Yang, H., and Zawirski, M. (2016). Specification and complexity of collaborative text editing. In *ACM Symposium on Principles of Distributed Computing*, PODC 2016, pages 259–268. ACM.

Auvolat, A. and Taïani, F. (2019). Merkle search trees: Efficient state-based crdts in open networks. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 221–22109.

Bailis, P., Ghodsi, A., Hellerstein, J. M., and Stoica, I. (2013). Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA. ACM.

Bieniusa, A., Zawirski, M., Preguiça, N. M., Shapiro, M., Baquero, C., Balegas, V., and Duarte, S. (2012). An optimized conflict-free replicated set. *CoRR*, abs/1210.3368.

Biswas, R., Emmi, M., and Enea, C. (2019). On the complexity of checking consistency for replicated data types. In Dillig, I. and Tasiran, S., editors, *Computer Aided Verification*, pages 324–343, Cham. Springer International Publishing.

Biswas, R. and Enea, C. (2019). On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.*, 3(OOPSLA):165:1–165:28.

Blau, T. (2020). Verifying strong eventual consistency in δ-crdts. *CoRR*, abs/2006.09823.

Bouajjani, A. and Emmi, M. (2013). Analysis of recursively parallel programs. *ACM Trans. Program. Lang. Syst.*, 35(3):10:1–10:49.

Bouajjani, A., Enea, C., Guerraoui, R., and Hamza, J. (2017). On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 626–638, New York, NY, USA. ACM.

Bouajjani, A., Enea, C., and Hamza, J. (2014a). Verifying eventual consistency of optimistic replication systems. *SIGPLAN Not.*, 49(1):285–296.

Bouajjani, A., Enea, C., and Hamza, J. (2014b). Verifying eventual consistency of optimistic replication systems. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 285–296.

Bouajjani, A., Enea, C., Mukund, M., R., G. S., and Suresh, S. P. (2020). Formalizing and checking multilevel consistency. In Beyer, D. and Zufferey, D., editors, *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, volume 11990 of *Lecture Notes in Computer Science*, pages 379–400. Springer.

Brewer, E. A. (2000). Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*.

Briot, L., Urso, P., and Shapiro, M. (2016). High responsiveness for group editing CRDTs. In *19th International Conference on Supporting Group Work*, GROUP 2016, pages 51–60. ACM.

Brocco, A. (2021). Delta-state JSON CRDT: Putting collaboration on solid ground. In *23rd International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS 2021, pages 474–478. Springer LNCS volume 13046.

Burckhardt, S. (2014). Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150.

Burkhardt, S., Gotsman, A., Yang, H., and Zawirski, M. (2014). Replicated data types: specification, verification, optimality. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 271–284.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794.

CS551 (2001). 551: Distributed operating systems. `http://www.cs.colostate.edu/~cs551/CourseNotes/Consistency/TypesConsistency.html`. Accessed: 2018-03-31.

Damien. (2017 (Accessed Nov 16, 2018)). *DynamoDB vs Cassandra*.

De Porre, K., Myter, F., Scholliers, C., and Gonzalez Boix, E. (2020). CScript: A distributed programming language for building mixed-consistency applications. *Journal of Parallel and Distributed Computing volume 144*, pages 109–123.

Documentation, A. (2018 (Accessed Sep 26, 2019)). *DAX and DynamoDB consistency Models*.

Dongol, B. and Hierons, R. M. (2016). Decidability and complexity for quiescent consistency. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 116–125, New York, NY, USA. ACM.

Emmi, M. and Enea, C. (2018). Monitoring weak consistency. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 487–506.

Enes, V., Almeida, P. S., Baquero, C., and Leitão, J. (2019). Efficient synchronization of state-based crdts. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 148–159.

Furbach, F., Meyer, R., Schneider, K., and Senftleben, M. (2014). Memory model-aware testing - A unified complexity analysis. In *14th International Conference on Application of Concurrency to System Design, ACSD 2014, Tunis La Marsa, Tunisia, June 23-27, 2014*, pages 92–101.

Gilbert, S. and Lynch, N. A. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59.

Gomes, V. B. F., Kleppmann, M., Mulligan, D. P., and Beresford, A. R. (2017). Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA).

Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., and Shapiro, M. (2016). 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 371–384.

Grishchenko, V. and Patrakeev, M. (2020). Chronofold: A data structure for versioned text. In *7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC 2020. ACM.

Grosch, P., Krafft, R., Wölki, M., and Bieniusa, A. (2020). *AutoCouch: A JSON CRDT Framework*. Association for Computing Machinery, New York, NY, USA.

Guerraoui, R., Pavlovic, M., and Seredinschi, D.-A. (2016). Incremental consistency guarantees for replicated objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 169–184, Berkeley, CA, USA. USENIX Association.

Gurevich, Y. and Harrington, L. (1982). Trees, automata, and games. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 60–65, New York, NY, USA. ACM.

Herlihy, M. (2008). *Linearizability*, pages 450–453. Springer US, Boston, MA.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492.

Kleppmann, M. and Beresford, A. R. (2017). A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746.

Kraska, T., Hentschel, M., Alonso, G., and Kossmann, D. (2009). Consistency rationing in the cloud: Pay only when it matters. *PVLDB*, 2(1):253–264.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691.

Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. (2012). Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA. USENIX Association.

Malkhi, D. and Terry, D. B. (2007). Concise version vectors in WinFS. *Distributed Computing*, 20(3):209–219.

Mazurkiewicz, A. (1987). Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 278–324. Springer.

McNaughton, R. (1965). Finite-state infinite games. *Project MAC Rep.*

Meiklejohn, C. and Van Roy, P. (2015). Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, PPDP '15, page 184–195, New York, NY, USA. Association for Computing Machinery.

Mukund, M., Narayan Kumar, K., and Sohoni, M. A. (2003). Bounded time-stamping in message-passing systems. *Theor. Comput. Sci.*, 290(1):221–239.

Mukund, M., Shenoy, G. R., and Suresh, S. P. (2014). Optimized or-sets without ordering constraints. In Chatterjee, M., Cao, J.-N., Kothapalli, K., and Rajsbaum, S., editors, *ICDCN*, volume 8314 of *Lecture Notes in Computer Science*, pages 227–241. Springer.

Mukund, M., Shenoy R, G., and Suresh, S. P. (2015a). Bounded implementations of replicated data types. In *Proceedings of VMCAI 2015*, volume 8931 of *LNCS*, pages 355–372.

Mukund, M., Shenoy R., G., and Suresh, S. P. (2015b). Effective verification of replicated data types using later appearance records (LAR). In *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*, pages 293–308.

Mukund, M., Shenoy R., G., and Suresh, S. P. (2020). Bounded version vectors using mazurkiewicz traces. In Chaki, R., Cortesi, A., Saeed, K., and Chaki, N., editors, *Advanced Computing and Systems for Security - Volume Eleven, 7th International Doctoral Symposium on Applied Computation and Security Systems, ACSS 2020, Kolkata, India, February 28-29, 2020*, volume 1178 of *Advances in Intelligent Systems and Computing*, pages 31–42. Springer.

Mukund, M. and Sohoni, M. A. (1997). Keeping track of the latest gossip in a distributed system. *Distributed Computing*, 10(3):137–148.

Nicolas, M., Oster, G., and Perrin, O. (2020). Efficient renaming in sequence CRDTs. In *7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC 2020. ACM.

Owen, M. (2015). Using erlang, riak and the orswot crdt at bet365 for scalability and performance. `https://www.erlang-factory.com/static/upload/media/1434558446558020erlanguserconference2015bet365michaelowen.pdf`. Accessed : 2022-03-06.

Ozkan, B. K., Majumdar, R., Niksic, F., Befrouei, M. T., and Weissenbacher, G. (2018). Randomized testing of distributed systems with probabilistic guarantees. *PACMPL*, 2(OOPSLA):160:1–160:28.

Perrin, M., Mostefaoui, A., and Jard, C. (2016). Causal consistency: Beyond memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 26:1–26:12, New York, NY, USA. ACM.

Pratt, V. (1986). Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71.

Redis (2020). Diving into crdts. https://redis.com/blog/diving-into-crdts/. Accessed : 2022-03-06.

Riak (2015). Riak concept data types. https://docs.riak.com/riak/kv/2.2.3/learn/concepts/crdts/index.html. Accessed : 2022-03-06.

Saito, Y. and Shapiro, M. (2005). Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81.

Shapiro, M. and Kemme, B. (2009). Eventual consistency. In *Encyclopedia of Database Systems*, pages 1071–1072.

Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011a). A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA. http://hal.inria.fr/inria-00555588/PDF/techreport.pdf.

Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. (2011b). Conflict-free replicated data types. In *SSS*, pages 386–400.

Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M., Theimer, M., and Welch, B. B. (1994). Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994*, pages 140–149. IEEE Computer Society.

Terry, D. B., Theimer, M., Petersen, K., Demers, A. J., Spreitzer, M., and Hauser, C. (1995). Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 172–183.

Vogels, W. (2008). Eventually consistent. *ACM Queue*, 6(6):14–19.

Weidner, M., Miller, H., and Meiklejohn, C. (2020). Composing and decomposing op-based crdts with semidirect products. *Proc. ACM Program. Lang.*, 4(ICFP).

Willcock, C., Deiß, T., Tobies, S., Keil, S., Engler, F., and Schulz, S. (2005). *An Introduction to TTCN-3*. Wiley.

Wolper, P. (1986). Expressing interesting properties of programs in propositional temporal logic. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pages 184–193.

Yu, W., Elvinger, V., and Ignat, C.-L. (2020). A Generic Undo Support for State-Based CRDTs. In Felber, P., Friedman, R., Gilbert, S., and Miller, A., editors, *23rd International*

Conference on Principles of Distributed Systems (OPODIS 2019), volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:17, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Zeller, P., Bieniusa, A., and Poetzsch-Heffter, A. (2014). Formal specification and verification of CRDTs. In *34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems*, FORTE 2014, pages 33–48. Springer LNCS volume 8461.