Verifying Strong Eventual Consistency in Distributed Systems

VICTOR B. F. GOMES, University of Cambridge, UK
MARTIN KLEPPMANN, University of Cambridge, UK
DOMINIC P. MULLIGAN, University of Cambridge, UK
ALASTAIR R. BERESFORD, University of Cambridge, UK

Data replication is used in distributed systems to maintain up-to-date copies of shared data across multiple computers in a network. However, despite decades of research, algorithms for achieving consistency in replicated systems are still poorly understood. Indeed, many published algorithms have later been shown to be incorrect, even some that were accompanied by supposed mechanised proofs of correctness. In this work, we focus on the correctness of Conflict-free Replicated Data Types (CRDTs), a class of algorithm that provides strong eventual consistency guarantees for replicated data. We develop a modular and reusable framework in the Isabelle/HOL interactive proof assistant for verifying the correctness of CRDT algorithms. We avoid correctness issues that have dogged previous mechanised proofs in this area by including a network model in our formalisation, and proving that our theorems hold in all possible network behaviours. Our axiomatic network model is a standard abstraction that accurately reflects the behaviour of real-world computer networks. Moreover, we identify an abstract convergence theorem, a property of order relations, which provides a formal definition of strong eventual consistency. We then obtain the first machine-checked correctness theorems for three concrete CRDTs: the Replicated Growable Array, the Observed-Remove Set, and an Increment-Decrement Counter. We find that our framework is highly reusable, developing proofs of correctness for the latter two CRDTs in a few hours and with relatively little CRDT-specific code.

CCS Concepts: • **Networks** → **Protocol testing and verification**; *Formal specifications*; • **Computer systems organization** → **Peer-to-peer architectures**; • **Theory of computation** → *Distributed algorithms*; *Program verification*; • **Software and its engineering** → *Formal software verification*;

Additional Key Words and Phrases: strong eventual consistency, verification, distributed systems, replication, convergence, CRDTs, automated theorem proving

ACM Reference Format:
Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (October 2017), 28 pages. https://doi.org/10.1145/3133933

1 INTRODUCTION

A data replication algorithm is executed by a set of computers—or *nodes*—in a distributed system, and ensures that all nodes eventually obtain an identical copy of some shared state. Whilst vital

Authors' addresses: Victor B. F. Gomes, Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK, vb358@cam.ac.uk; Martin Kleppmann, Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK, mk428@cam.ac.uk; Dominic P. Mulligan, Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK, dpm36@cam.ac.uk; Alastair R. Beresford, Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK, arb33@cam.ac.uk.

2475-1421/2017/10-ART109
https://doi.org/10.1145/3133933

for overall systems correctness, implementing a replication algorithm is a challenging task, as any such algorithm must operate across computer networks that may arbitrarily delay, drop, or reorder messages, experience temporary partitions of the nodes, or even suffer outright node failure. Reflecting the importance of this task, a number of replication algorithms exist, with different algorithms exploring the inherent trade-offs between the strength of data consistency guarantees, and operational characteristics such as scalability and performance. Accordingly, replication algorithms can be divided into classes—*strong consistency*, *eventual consistency*, and *strong eventual consistency*—based on the consistency guarantees that they provide.

Strong consistency can be understood as *linearisability*, *serialisability*, or a combination of the two (one-copy serialisability). Informally, the goal of strong consistency is to make a system behave like a single sequentially executing node, even when it is replicated and concurrent. Most systems implement strong consistency by designating a single node as the *leader*, which decides on a total order of operations and prevents concurrent access from causing conflicts. Many relational databases, such as PostgreSQL, use this model.

However, strong consistency may be unwarranted or unnecessary depending on the application: it may impose an unacceptable performance degradation on the system, or it may simply be unfeasible to implement, especially in large distributed systems. Relying on a single leader or central server limits the use and deployment of these systems: the server may become a bottleneck that limits scalability, and it makes the system vulnerable to disruption by network outages, denial-of-service attacks, censorship, and server failures. Clients must constantly communicate with the leader in order to perform operations; if a node cannot reach the leader due to a network fault, its execution is stalled. This fact makes strong consistency unsuitable for mobile devices, such as laptops and smartphones, that have intermittent network connectivity and must work offline. It also rules out approaches that bypass the central server by using a local network for replication.

By contrast, decentralised or peer-to-peer architectures with weaker consistency models are able to provide better performance, fault-tolerance, and scalability characteristics. One widely-implemented model is *eventual consistency*, which guarantees that if no new updates are made to the shared state, all nodes will eventually have the same data [Bailis and Ghodsi 2013; Burckhardt 2014; Terry et al. 1994; Vogels 2009]. Since this model allows conflicting updates to be made concurrently, it requires a mechanism for resolving such conflicts. For example, version control systems such as Git or Mercurial require the user to resolve merge conflicts manually; and some "NoSQL" distributed database systems such as Cassandra adopt a *last-writer-wins* policy, under which one update is chosen as the winner, and concurrent updates are discarded [Kingsbury 2013]. Eventual consistency offers weak guarantees: it does not constrain the system behaviour when updates never cease, or the values that read operations may return prior to convergence.

Strong eventual consistency (SEC) is a model that strikes a compromise between strong and eventual consistency [Shapiro et al. 2011b]. Informally, it guarantees that whenever two nodes have received the same set of updates—possibly in a different order—their view of the shared state is identical, and any conflicting updates are merged automatically. Large-scale deployments of SEC algorithms include datacentre-based applications using Riak [Brown et al. 2014], and collaborative editing applications such as Google Docs [Day-Richter 2010].

Unlike strong consistency models, it is possible to implement SEC in decentralised settings without any central server or leader, and it allows local execution at each node to proceed without waiting for communication with other nodes. However, algorithms for achieving decentralised SEC are currently poorly understood: several such algorithms, published in peer-reviewed venues, were subsequently shown to violate their supposed guarantees [Imine et al. 2003, 2006; Oster et al. 2005]. As we show in Section 8, informal reasoning has repeatedly produced plausible-looking but incorrect algorithms, and there have even been examples of mechanised formal proofs of SEC

algorithm correctness later being shown to be flawed [Oster et al. 2005]. These mechanised proofs failed because, in formalising the algorithm, they made false assumptions about the execution environment.

In this work we use the Isabelle/HOL proof assistant [Wenzel et al. 2008] to create a framework for reliably reasoning about the correctness of a particular class of decentralised replication algorithms. We do this by formalising not only the replication algorithms, but also the network in which they execute, allowing us to prove that the algorithm's assumptions hold in all possible network behaviours. We model the network using the axioms of *asynchronous unreliable causal broadcast*, a well-understood abstraction that is commonly implemented by network protocols, and which can run on almost any computer network, including large-scale networks that delay, reorder, or drop messages, and in which nodes may fail.

We then use this framework to produce machine-checked proofs of correctness for three Conflict-Free Replicated Data Types (CRDTs), a class of replication algorithms that ensure strong eventual consistency [Shapiro et al. 2011a,b]. These algorithms are suitable for use on mobile devices, which are not always connected to the Internet, but which may have a local connection (e.g. via Bluetooth) to other nodes carrying copies of the shared state. We have used these algorithms to build a collaborative text editing application, and we plan to encapsulate them in a library that will allow developers to easily build applications that require data synchronisation, such as collaboratively editable spreadsheets, shared calendars, address books, and note-taking tools.

Our contributions in this paper are as follows:

- We establish a framework for proving the strong eventual consistency (SEC) property of replication algorithms. Our approach is "foundational" in the sense that we start with a general-purpose model of asynchronous unreliable causal broadcast networks—a communication abstraction that is compatible with virtually all network technologies today—and build up composable layers towards a full proof of correctness for a particular algorithm. To our knowledge, this is the first machine-checked verification of SEC algorithms that explicitly models the network and reasons about all possible network behaviours. The framework is modular and reusable, making it easy to formulate proofs for new algorithms.
- We provide the first mechanised proofs of correctness for the Replicated Growable Array (RGA), the operation-based Observed-Remove Set, and the operation-based counter CRDT. RGA is an especially subtle algorithm: Attiya et al. [2016] wrote, "the reason why RGA actually works has been a bit of a mystery", making its formal verification of interest, whilst the ORSet is supported as a primitive by the Lasp language [Meiklejohn and Roy 2015] for synchronisation-free programming, with an implemention also exported by the Akka framework [Akka 2017]. These proofs demonstrate that our framework is highly reusable: we were able to quickly develop proofs of convergence for the set and counter CRDTs with little CRDT-specific code, using a fixed proof pattern that applies to all of our CRDTs. All of our CRDT implementations are "executable" in the sense that functioning OCaml (or Scala, SML, and Haskell) code can be obtained from our definitions using Isabelle's code generation mechanism, and in experiments we have used one extracted implementation, sitting above a simple TCP network of n nodes, to show that our implementations are usable in practice.
- As part of our proof framework, we identify an abstract convergence theorem, a property of order relations, from which we can deduce correctness theorems for concrete SEC algorithms. Intuitively, this theorem can be viewed as the "essence" of why strong eventual consistency algorithms converge. The convergence theorems for our three concrete CRDTs are obtained as direct corollaries of this theorem.

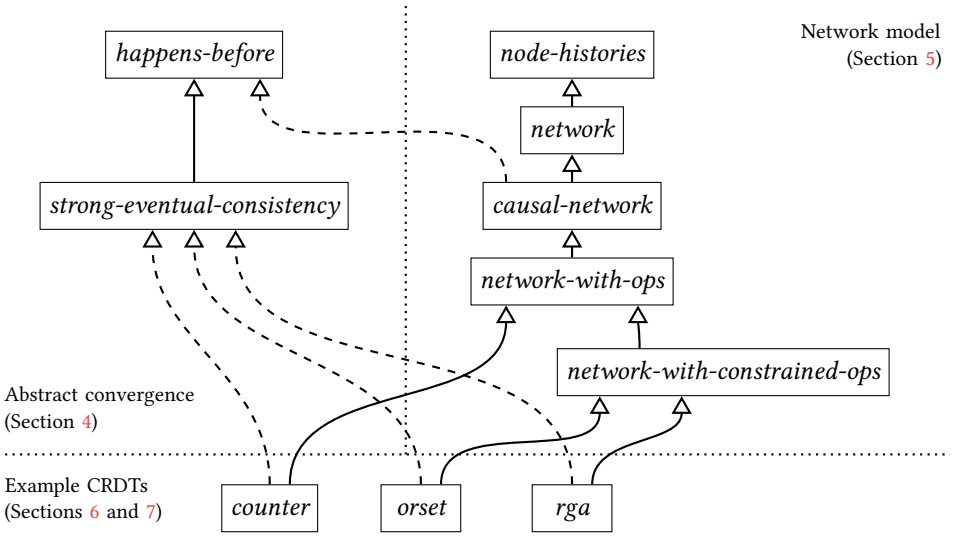

Fig. 1. The main locales (modules) of our proof, and the relationships between them. Solid arrows indicate a more specialised locale that extends a more general locale (like extending interfaces in OOP). Dashed arrows indicate a sublocale that satisfies the assumptions of the superlocale (like implementing an interface in OOP).

Our Isabelle theory files are open source[1] and included in the Archive of Formal Proofs [Gomes et al. 2017], enabling others to build upon our proof framework.

2 HIGH-LEVEL PROOF STRATEGY

Since our formalisation of distributed algorithms goes into greater depth than prior work on strong eventual consistency, it is important to have a structure that keeps the proofs manageable. Our approach breaks the proof into simple modules with cleanly defined properties—called *locales*, a standard sectioning mechanism of Isabelle/HOL that will be described in Section 3 below—and composes them in order to describe more complex objects. This locale structure is illustrated in Figure 1 and explained below.

By lines of code, more than half of our proof is used to construct a general-purpose model of consistency in distributed systems, described in Section 4, and an axiomatic model of a computer network, described in Section 5, with both modules independent of any particular replication algorithm. The remainder describes a formalisation of three CRDTs and their proofs of correctness, described in Sections 6 and 7. By keeping the general-purpose modules abstract and implementation-independent, we construct a reusable library of specifications and theorems.

We describe our formalisation of strong eventual consistency in Section 4. In particular, we define what we mean by convergence, and prove an *abstract convergence theorem*, which shows that the state of nodes converges if concurrent operations commute. We are able to prove this fact without mentioning networks or any particular CRDT, but merely by reasoning about the ordering and properties of operations. This definition constitutes a formal specification of what we mean by strong eventual consistency.

In Section 5 we describe an axiomatic model of asynchronous networks. The definition of the network is important because it allows us to prove that the desired properties hold in *all* possible

[1] https://github.com/trvedata/crdt-isabelle

network behaviours, and that we are not making any dangerous assumptions that might be violated—an aspect that has dogged previous verification efforts for related algorithms (see Section 8.2). The network is the only part of our proof in which we make any axiomatic assumptions, and we show in Section 5 that our assumptions are realistic, reflecting both standard conventions for modelling distributed systems, and the practical realities of network protocols today. We then prove that our network satisfies the ordering properties required by the abstract convergence theorem of Section 4, and thus deduce a convergence theorem for our network model.

We use the general-purpose theorems and definitions from Sections 4 and 5 to prove the strong eventual consistency properties of concrete algorithms. In Section 6 we describe our formalisation of the Replicated Growable Array (RGA), a CRDT for ordered lists. We first show how to implement the RGA's insert and delete operations, with proofs that each operation commutes with itself, and that all operations commute with each other. Insertion and deletion only commute under various conditions, so we prove that these conditions are satisfied in all possible network behaviours, and thus we obtain a concrete convergence theorem for our RGA implementation. Next, in Section 7, we demonstrate the generality of our proof framework with definitions of two simple CRDTs: a Counter and an Observed-Remove Set.

As illustrated in Figure 1, the *counter*, *orset*, and *rga* locales can use the definitions and lemmas of the network model because they extend that model. We then prove that all three locales satisfy the abstract specification *strong-eventual-consistency*, and therefore show that these algorithms provide strong eventual consistency.

3 AN INTRODUCTION TO ISABELLE

We now provide a brief introduction to the key concepts and syntax of Isabelle/HOL. Familiar readers may skip to Section 4. A more detailed introduction can be found in the standard tutorial material [Nipkow and Klein 2014].

Syntax of expressions. Isabelle/HOL is a logic with a strict, polymorphic, inferred type system. *Function types* are written $\tau_1 \Rightarrow \tau_2$, and are inhabited by *total* functions, mapping elements of τ_1 to elements of τ_2. We write $\tau_1 \times \tau_2$ for the *product type* of τ_1 and τ_2, inhabited by pairs of elements of type τ_1 and τ_2, respectively. In a similar fashion to Standard ML and OCaml, *type operators* are applied to arguments in reverse order, and therefore τ *list* denotes the type of lists of elements of type τ, and τ *set* denotes the type of mathematical (i.e., potentially infinite) sets of type τ. Type variables are written in lowercase, and preceded with a prime: *'a* $\Rightarrow$ *'a* denotes the type of a polymorphic identity function, for example. *Tagged union* types are introduced with the **datatype** keyword, with constructors of these types usually written with an initial upper case letter.

In Isabelle/HOL's term language we write $t :: \tau$ for a *type ascription*, constraining the type of the term t to the type τ. We write $\lambda x.\ t$ for an anonymous function mapping an argument x to *t(x)*, and write the application of term t with function type to an argument u as $t\ u$, as usual. Terms of list type are introduced using one of two constructors: the empty list [ ] or 'nil', and the infix operator # which is pronounced "cons", and which prepends an element to an existing list. We use $[t_1, \ldots, t_n]$ as syntactic sugar for a list literal, and xs @ ys to express the concatenation (appending) of two lists xs and ys. We write { } for the empty set, and use usual mathematical notation for set union, disjunction, membership tests, and so on: $t \cup u$, $t \cap u$, and $x \in t$. We write $t \longrightarrow s$ for logical implication between formulae (terms of type *bool*). Strictly speaking Isabelle is a logical framework, providing a weak meta-logic within which object logics are embedded, including the Isabelle/HOL object logic that we use in this work. Accordingly, the implication arrow of Isabelle's meta-logic, $t \Longrightarrow u$, is required in certain contexts over the object-logic implication arrow, $t \longrightarrow s$, already introduced. However, for purposes of an intuitive understanding, the two forms of

implication can be regarded as equivalent by the reader, with the requirement to use one over the other merely being an implementation detail of Isabelle itself. We will sometimes use the shorthand $[\![H_1; \ldots; H_n]\!] \Longrightarrow C$ instead of iterated meta-logic implications, i.e., $H_1 \Longrightarrow \ldots \Longrightarrow H_n \Longrightarrow C$.

Definitions and theorems. New non-recursive definitions are entered into Isabelle's global context using the **definition** keyword. Recursive functions are defined using the **fun** keyword, and support pattern matching on their arguments. All functions are total, and therefore every recursive function must be provably terminating. The termination proofs in this work are generated automatically by Isabelle itself.

Inductive relations are defined with the **inductive** keyword. For example, the definition

inductive *only-fives* :: *nat list* ⇒ *bool* **where**
only-fives [] |
[[*only-fives xs*]] ⟹ *only-fives* (*5#xs*)

introduces a new constant *only-fives* of type *nat list* ⇒ *bool*. The two clauses in the body of the definition enumerate the conditions under which *only-fives xs* is true, for arbitrary *xs*: firstly, *only-fives* is true for the empty list; and secondly, if you know that *only-fives xs* is true for some *xs*, then you can deduce that *only-fives* (5#*xs*) (i.e., *xs* prefixed with the number 5) is also true. Moreover, *only-fives xs* is true in no other circumstances—it is the *smallest* relation closed under the rules defining it. In short, the clauses above state that *only-fives xs* holds exactly in the case where *xs* is a (potentially empty) list containing only repeated copies of the natural number 5.

Lemmas, theorems, and corollaries can be asserted using the **lemma**, **theorem**, and **corollary** keywords, respectively. There is no semantic difference between these keywords in Isabelle. For example,

theorem *only-fives-concat*:
assumes *only-fives xs* **and** *only-fives ys*
shows *only-fives (xs @ ys)*

conjectures that if *xs* and *ys* are both lists of fives, then their concatenation *xs* @ *ys* is also a list of fives. Isabelle then requires that this claim be proved by using one of its proof methods, for example by induction. Some proofs can be automated, whilst others require the user to provide explicit reasoning steps. The theorem is assigned a name, here *only-fives-concat*, so that it may be referenced in later proofs.

Locales. Lastly, we use *locales*—or local theories [Haftmann and Wenzel 2008; Kammüller et al. 1999]—extensively to structure the proof, as shown in Figure 1. In programming terms, Isabelle's locales may be thought of as an interface with associated laws that implementations must obey. In particular, a declaration of the form

locale *semigroup* =
fixes *f* :: *'a* ⇒ *'a* ⇒ *'a*
assumes *f x (f y z)* = *f (f x y) z*

introduces a locale, with a fixed, typed constant *f*, and a law asserting that *f* is associative. Functions and constants may now be defined, and theorems conjectured and proved, within the context of the *semigoup* locale, i.e. definitions may be made "generic" in a semigroup. This is indicated syntactically by writing (**in** *semigroup*) before the name of the constant being defined, or the theorem being conjectured, at the point of definition or conjecture. Any function, constant, or theorem, marked in this way may make reference to *f*, or the fact that *f* is associative. *Interpreting* a locale—such as *semigroup* above—involves providing a concrete implementation of *f* coupled with a proof that the concrete implementation satisfies the associated law, and is akin to implementing an interface.

Once interpreted, all functions, definitions, and theorems made within the *semigroup* locale become available to use for that concrete implementation. Like interfaces, locales may be extended with new functionality, and may be specialised, by other "sublocales", forming a hierarchy.

4 ABSTRACT CONVERGENCE

Strong eventual consistency (SEC) requires *convergence* of all copies of the shared state: whenever two nodes have received the same set of updates, they must be in the same state. This definition constrains the values that read operations may return at any time, making SEC a stronger property than eventual consistency. By accessing only their local copy of the shared state, nodes can execute read and write operations without waiting for network communication. Nodes exchange updates asynchronously when a network connection is available.

We now use Isabelle to formalise the notion of strong eventual consistency. In this section we do not make any assumptions about networks or data structures; instead, we use an abstract model of operations that may be reordered, and we reason about the properties that those operations must satisfy. We then provide concrete implementations of that abstract model in later sections.

4.1 The Happens-before Relation and Causality

The simplest way of achieving convergence is to require all operations to be commutative, but this definition is too strong to be useful for many datatypes. For example, in a set, an element may first be added and then subsequently removed again. Although it is possible to make such additions and removals unconditionally commutative, doing so yields counter-intuitive semantics [Bieniusa et al. 2012a,b]. Instead, a better approach is to require only *concurrent* operations to commute with each other. Two operations are concurrent if neither "knew about" the other at the time when they were generated. If one operation happened before another—for example, if the removal of an element from a set knew about the prior addition of that element from the set—then it is reasonable to assume that all nodes will apply the operations in that order (first the addition, then the removal).

The *happens-before* relation, as introduced by Lamport [1978], captures such causal dependencies between operations. It can be defined in terms of sending and receiving messages on a network, and we give such a definition in Section 5. However, for now, we keep it abstract, writing $x \prec y$ to indicate that operation x happened before y, where $\prec$ is a predicate of type $'oper \Rightarrow 'oper \Rightarrow bool$. In words, $\prec$ can be applied to two operations of some abstract type $'oper$, returning either *True* or *False*.[2] Our only restriction on the happens-before relation $\prec$ is that it must be a *strict partial order*, that is, it must be irreflexive and transitive, which implies that it is also antisymmetric. We say that two operations x and y are *concurrent*, written $x \parallel y$, whenever one does not happen before the other: $\neg(x \prec y)$ and $\neg(y \prec x)$. Thus, given any two operations x and y, there are three mutually exclusive ways in which they can be related: either $x \prec y$, or $y \prec x$, or $x \parallel y$.

As discussed above, the purpose of the happens-before relation is to require that some operations must be applied in a particular order, while allowing concurrent operations to be reordered with respect to each other. We assume that each node applies operations in some sequential order (a standard assumption for distributed algorithms), and so we can model the execution history of a node as a list of operations. We can then inductively define a list of operations as being *consistent with the happens-before relation*, or simply *hb-consistent*, as follows:

inductive *hb-consistent* :: $'oper\ list \Rightarrow bool$ **where**
hb-consistent [] |
$[\![$ *hb-consistent xs*; $\forall x \in set\ xs.\ \neg\ y \prec x\ ]\!] \Longrightarrow$ *hb-consistent* $(xs\ @\ [y])$

[2]Note that in the distributed systems literature it is conventional to write the happens-before relation as $x \rightarrow y$, but we reserve the arrow operator to denote logical implication.

In words: the empty list is hb-consistent; furthermore, given an hb-consistent list *xs*, we can append an operation y to the end of the list to obtain another hb-consistent list, provided that y does not happen-before any existing operation x in *xs*. As a result, whenever two operations x and y appear in a hb-consistent list, and $x \prec y$, then x must appear before y in the list. However, if $x \parallel y$, the operations can appear in the list in either order.

4.2 Interpretation of Operations

We describe the state of a node using an abstract type variable *'state*. To model state changes, we assume the existence of an *interpretation* function of type *interp* :: *'oper* ⇒ *'state* ⇒ *'state option*, which lifts an operation into a *state transformer*—a function that either maps an old state to a new state, or fails by returning *None*. If x is an operation, we also write $\langle x \rangle$ for the state transformer obtained by applying x to the interpretation function.

Concretely, these definitions are captured in Isabelle with the following locale declaration:

```
locale happens-before = preorder hb-weak hb
  for   hb-weak :: 'oper ⇒ 'oper ⇒ bool
  and   hb      :: 'oper ⇒ 'oper ⇒ bool +
  fixes interp  :: 'oper ⇒ 'state ⇒ 'state option
```

The *happens-before* locale extends the *preorder* locale, which is part of Isabelle's standard library and includes various useful lemmas. It fixes two constants: a preorder that we call *hb-weak* or $\preceq$, and a strict partial order that we call *hb* or $\prec$. We are only interested in the strict partial order and define $x \preceq y$ to be $x \prec y \vee x = y$. Moreover, the locale fixes the interpretation function *interp* as described above, which means that we assume the existence of a function with the given type signature without specifying an implementation.

Given two operations x and y, we can now define the composition of state transformers: we write $\langle x \rangle \rhd \langle y \rangle$ to denote the state transformer that first applies the effect of x to some state, and then applies the effect of y to the result. If either $\langle x \rangle$ or $\langle y \rangle$ fails, the combined state transformer also fails. The operator $\rhd$ is a specialised form of the *Kleisli arrow composition*, which we define as:

```
definition kleisli :: ('a ⇒ 'a option) ⇒ ('a ⇒ 'a option) ⇒ ('a ⇒ 'a option) where
  f ▷ g ≡ λx. f x >=> (λy. g y)
```

Here, $>\!\!=\!\!>$ is the *monadic bind* operation, defined on the option type that we are using to implement partial functions. We can now define a function *apply-operations* that composes an arbitrary list of operations into a state transformer. We first map *interp* across the list to obtain a state transformer for each operation, and then collectively compose them using the Kleisli arrow composition combinator:

```
definition apply-operations :: 'oper list ⇒ 'state ⇒ 'state option where
  apply-operations ops ≡ foldl (op ▷) Some (map interp ops)
```

The result is a state transformer that applies the interpretation of each of the operations in the list, in left-to-right order, to some initial state. If any of the operations fails, the entire composition returns *None*.

4.3 Commutativity and Convergence

We say that two operations x and y *commute* whenever $\langle x \rangle \rhd \langle y \rangle = \langle y \rangle \rhd \langle x \rangle$, i.e. when we can swap the order of the composition of their interpretations without changing the resulting state transformer. For our purposes, requiring that this property holds for *all* pairs of operations is too strong. Rather, the commutation property is only required to hold for operations that are concurrent, as captured in the next definition:

definition *concurrent-ops-commute* :: *'oper list* ⇒ *bool* **where**
concurrent-ops-commute xs ≡ ∀ *x y*. {*x*, *y*} ⊆ *set xs* ⟶ *x* ∥ *y* ⟶ ⟨*x*⟩▷⟨*y*⟩ = ⟨*y*⟩▷⟨*x*⟩

Given this definition, we can now state and prove our main theorem, *convergence*. This theorem states that two hb-consistent lists of distinct operations, which are permutations of each other and in which concurrent operations commute, have the same interpretation:

theorem *convergence*:
assumes *set xs* = *set ys* **and** *concurrent-ops-commute xs* **and** *concurrent-ops-commute ys*
and *distinct xs* **and** *distinct ys* **and** *hb-consistent xs* **and** *hb-consistent ys*
shows *apply-operations xs* = *apply-operations ys*

A fully mechanised proof of this theorem can be found in our submission to the Archive of Formal Proofs [Gomes et al. 2017]. Although this theorem may seem "obvious" at first glance—commutativity allows the operation order to be permuted—it is more subtle than it seems. The difficulty arises because operations may succeed when applied to some state, but fail when applied to another state (for example, attempting to delete an element that does not exist in the state). We find it interesting that it is nevertheless sufficient for the definition of *concurrent-ops-commute* to be expressed only in terms of the Kleisli arrow composition, and without explicitly referring to the state.

4.4 Formalising Strong Eventual Consistency

Besides convergence, another required property of SEC is *progress*: if one node issues a valid operation, and another node applies that operation, then it must not become stuck in an error state. Although the type signature of the interpretation function allows operations to fail, we need to prove that such a failure never occurs in any *hb-consistent* network behaviour. We capture this requirement in the *strong-eventual-consistency* locale:

locale *strong-eventual-consistency* = *happens-before* +
fixes *op-history* :: *'oper list* ⇒ *bool* **and** *initial-state* :: *'state*
assumes *causality*: ⟦ *op-history xs* ⟧ ⟹ *hb-consistent xs*
and *distinctness*: ⟦ *op-history xs* ⟧ ⟹ *distinct xs*
and *trunc-history*: ⟦ *op-history*(*xs*@[*x*]) ⟧ ⟹ *op-history xs*
and *commutativity*: ⟦ *op-history xs* ⟧ ⟹ *concurrent-ops-commute xs*
and *no-failure*: ⟦ *op-history*(*xs*@[*x*]); *apply-operations xs initial-state* = *Some state* ⟧ ⟹ ⟨*x*⟩ *state* ≠ *None*

Here, *op-history* is an abstract predicate describing any valid operation history of some replication algorithm, encapsulating the assumptions of the *convergence* theorem (*concurrent-ops-commute*, *distinct*, and *hb-consistent*). This locale serves as a concise summary of the properties that we require in order to achieve SEC, and from these assumptions and the theorem above we easily obtain the two safety properties of SEC as theorems:

theorem *sec-convergence*:
assumes *set xs* = *set ys* **and** *op-history xs* **and** *op-history ys*
shows *apply-operations xs* = *apply-operations ys*

theorem *sec-progress*:
assumes *op-history xs*
shows *apply-operations xs initial-state* ≠ *None*

Thus, in order to prove SEC for some replication algorithm, we only need to show that the five assumptions of the *strong-eventual-consistency* locale are satisfied. As we shall see in Section 5, the first three assumptions are satisfied by our network model, and do not require any algorithm-specific proofs. For individual algorithms we only need to prove the *commutativity* and *no-failure* properties, and we show how to do this in Sections 6 and 7.

Note that the *trunc-history* assumption requires that every prefix of a valid operation history is also valid. This means that the convergence theorem holds at every step of the execution, not only at some unspecified time in the future ("eventually"), making SEC stronger than eventual consistency.

5 AN AXIOMATIC NETWORK MODEL

In this section we develop a formal definition of an *asynchronous unreliable causal broadcast network*. We choose this model because it satisfies the causal delivery requirements of many operation-based CRDTs [Almeida et al. 2015; Baquero et al. 2014]. Moreover, it is suitable for use in decentralised settings, as motivated in the introduction, since it does not require waiting for communication with a central server or a quorum of nodes. Stronger consistency models do not have this property [Attiya et al. 2015; Davidson et al. 1985].

The *causal* and *broadcast* aspects of the model are explained in Sections 5.2 and 5.3. The *asynchronous* aspect means that we make no timing assumptions: messages sent over the network may suffer unbounded delays before they are delivered, nodes may pause their execution for unbounded periods of time, and we require no clock synchronisation. *Unreliable* means that messages may never arrive at all, and nodes may fail permanently without warning. Networks are known to exhibit these behaviours in practice [Bailis and Kingsbury 2014], and replication algorithms must tolerate such failures.

This model provides a realistic setting in which we can embed various replication algorithms, and prove that they guarantee SEC in all possible behaviours of the network. But it is also abstract enough to be able to model a wide range of scenarios: for example, if a user makes updates while offline, and the device re-synchronises when it is next online, we can simply model that interaction as very large network delay. Our network model is defined using only six axioms, all of which are standard assumptions when modelling distributed systems, and which are satisfied by many systems in practice. All theorems in this paper are derived from those axioms; in particular, we show that the causal delivery abstraction satisfies the strict partial ordering assumption of *hb-consistent* (Section 4.1), allowing us to use the convergence theorem in any locales that extend the network.

5.1 Modelling a Distributed System

We model a distributed system as an unbounded number of communicating nodes. We assume nothing about the communication pattern of nodes—we assume only that each node is uniquely identified by a natural number, and that the flow of execution at each node consists of a finite, totally ordered sequence of execution steps (events). We call that sequence of events at node i the *history* of that node. For convenience, we assume that every event or execution step is unique within a node's history; this assumption is standard when modelling distributed systems [Cachin et al. 2011] and can easily be implemented by attaching a sequence number, timestamp, or other unique identifier to each event. This system model can be expressed in Isabelle as follows:

locale *node-histories* =
  fixes *history* :: *nat* ⇒ *'a list*
  assumes *histories-distinct*: *distinct* (*history i*)

Here, the history of a node i is obtained by using a function fixed by the locale, *history*. The history is simply a list of events, and each event is modelled as an abstract type variable—here we use *'a*. The *distinct* predicate is an Isabelle/HOL library function that asserts that a list contains no duplicate elements. Note that we make no assumption about the number of nodes in the system, which allows us to model systems in which nodes join and leave the network over time. A node that does not exist is simply modelled as an empty list of events.

A node's history is finite, and at the end of a node's history we assume that a node has either failed or successfully terminated. We treat node failure as permanent, and model it by the absence of any further events in its history. This *crash-stop* abstraction is commonly used by distributed algorithms [Cachin et al. 2011].

In the *node-histories* locale we may write $x \sqsubset^i y$, which means that event x *comes before* event y in the history of node i. More formally, $x \sqsubset^i y$ if and only if there exist lists xs, ys, and zs such that $xs \mathbin{@} [x] \mathbin{@} ys \mathbin{@} [y] \mathbin{@} zs = history\ i$.

5.2 An Asynchronous Broadcast Network

We now extend the *node-histories* locale by defining how nodes can communicate. We specialise *'a* to be one of two kinds of event: either *broadcast* or *deliver*. (In the conventional distributed systems terminology, a *deliver* event indicates that a message was received from the network and delivered to the application.) Each event contains a message of some abstract type *'msg*:

datatype *'msg event* = *Broadcast 'msg* | *Deliver 'msg*

Intuitively, a node can be regarded as a deterministic state machine where each state transition corresponds to a broadcast or deliver event. We assume that users may query the state of any node at any time, and such queries need not be reflected as events, since they neither modify the node state nor send or receive any messages.

A broadcast abstraction is the standard network model for operation-based CRDTs because it best fits the replication pattern: any node can accept writes, and propagate them to the other nodes through broadcast. In practical systems, broadcast abstractions are often implemented as overlay networks on top of unicast TCP links, for example as a fully connected graph (each node is connected to every other node), using a spanning tree protocol, a gossip protocol, or some other network topology. Such protocols have already been studied extensively, for example by Leitão et al. [2007], so we leave the implementation of the overlay network out of the scope of this paper.

To formally specify the properties of a broadcast network, we define a new locale *network* containing three axioms that define how broadcast and deliver events may interact. Since *network* is an extension of *node-histories*, the aforementioned definitions of *history* and $\sqsubset^i$ are available for use in the *network* axioms:

locale *network* = *node-histories history*
  for *history* :: *nat* ⇒ *'msg event list* +
  fixes *msg-id* :: *'msg* ⇒ *'msgid*
  assumes *delivery-has-a-cause*:
    ⟦ *Deliver m* ∈ *set* (*history i*) ⟧ ⟹ ∃ *j*. *Broadcast m* ∈ *set* (*history j*)
  and *deliver-locally*: ⟦ *Broadcast m* ∈ *set* (*history i*) ⟧ ⟹ *Broadcast m* $\sqsubset^i$ *Deliver m*
  and *msg-id-unique*: ⟦ *Broadcast m1* ∈ *set* (*history i*);
    Broadcast m2 ∈ *set* (*history j*);
    msg-id m1 = *msg-id m2* ⟧ ⟹ *i* = *j* ∧ *m1* = *m2*

The axioms can be understood as follows:

delivery-has-a-cause: If some message *m* was delivered at some node, then there exists some node on which *m* was broadcast. With this axiom, we assert that messages are not created "out of thin air" by the network itself, and that the only source of messages are the nodes.

deliver-locally: If a node broadcasts some message *m*, then the same node must subsequently also deliver *m* to itself. Since *m* does not actually travel over the network, this local delivery is always possible, even if the network is interrupted. Local delivery may seem redundant, since its effect could also occur in the broadcast event, but it is convenient for algorithms that use the broadcast abstraction [Cachin et al. 2011].

msg-id-unique: We do not require the message type *'msg* to have any particular structure; we only assume the existence of a function *msg-id* :: *'msg* ⇒ *'msgid* that maps every message to some globally unique identifier of type *'msgid*. We assert this uniqueness by stating that if *m1* and *m2* are any two messages broadcast by any two nodes, and their *msg-id*s are the same, then they were in fact broadcast by the same node and the two messages are identical. In practice, these globally unique IDs can by implemented using unique node identifiers, sequence numbers or timestamps.

The *network* locale also inherits the *histories-distinct* axiom from its parent locale *node-histories*. Many other properties that we require can be deduced as lemmas from these axioms. For example, we can prove that for every message that is delivered by some node, there is exactly one broadcast event (on the same or some other node) that created the message. Also, due to the *histories-distinct* axiom we know that the same message is not delivered more than once to each node—an aspect that can be implemented in practical systems by having each node keep track of message IDs it has received, and suppressing any duplicates.

Note that we make no assumptions about the reliability or the ordering of messages. If one node broadcasts a message, it *may* be delivered by other nodes, but we do not state if or when that will happen. Messages may be arbitrarily delayed, reordered, or even lost entirely. It is even acceptable for a node to never deliver any messages besides those it broadcasts itself, modelling a node that is permanently disconnected from the network.

5.3 Causally Ordered Delivery

As discussed in Section 4.1, some replication algorithms require that some operations be applied in a particular order because the later operation has a causal dependency on the earlier one. We previously characterised these dependencies using the *happens-before* relation ≺, which we required to be a strict partial order, but otherwise kept abstract. In Section 4 we reasoned about the order of *operations*, but in a network we work with *messages*. We will connect operations and messages in Section 5.4; for now we will define a particular instance of the ordering relation ≺ on messages, and prove that it satisfies the requirements of a strict partial order.

We do not use physical time (such as UTC) to define the order of messages, since reliance on physical time is often problematic in distributed systems [Sheehy 2015]. Instead, we say that a message *m1* happens before another message *m2* if the node that generated *m2* "knew about" *m1* at the time *m2* was generated. More precisely, based on the well-known definition by Lamport [1978], we say that *m1* ≺ *m2* if any of the following is true:

(1) *m1* and *m2* were broadcast by the same node, and *m1* was broadcast before *m2*.
(2) The node that broadcast *m2* had delivered *m1* before it broadcast *m2*.
(3) There exists some operation *m3* such that *m1* ≺ *m3* and *m3* ≺ *m2*.

This verbal definition translates directly into Isabelle syntax:

inductive *hb* :: *'msg* ⇒ *'msg* ⇒ *bool* **where**
⟦ *Broadcast m1* $\sqsubset^i$ *Broadcast m2* ⟧ ⟹ *m1* ≺ *m2* |
⟦ *Deliver m1* $\sqsubset^i$ *Broadcast m2* ⟧ ⟹ *m1* ≺ *m2* |
⟦ *m1* ≺ *m2*; *m2* ≺ *m3* ⟧ ⟹ *m1* ≺ *m3*

Given this definition, we define a restricted variant of our broadcast network model by extending the *network* locale. In addition to the existing *network* axioms, we require that if there are any happens-before dependencies between messages, they must be delivered in that order. Concurrent messages may be delivered in any order.

locale *causal-network* = *network* +
assumes *causal-delivery*:
⟦ *Deliver m2* ∈ *set* (*history i*); *m1* ≺ *m2* ⟧ ⟹ *Deliver m1* $\sqsubset^i$ *Deliver m2*

The *causal-delivery* axiom does not strengthen the reliability assumptions of the network: only in the case where some message *m2* is delivered, it requires that any causally preceding messages are delivered first. It is still possible for some message never to be delivered. Causal delivery is typically implemented in network protocols using vector timestamps [Fidge 1988; Raynal and Singhal 1996; Schwarz and Mattern 1994]. As these protocols are widely known and well understood, we elide any further discussion.

5.4 Using Operations in the Network

We can now include the convergence theorem into our network model by further extending the *causal-network* locale. In the new locale *network-with-ops* we do not assume any additional axioms; we only specialise the type variable of messages *'msg* to be a pair of *'msgid*×*'oper*, and we instantiate the *msg-id* function fixed by the *network* locale to be *fst*, i.e., to return the first component *'msgid* of the pair. We also assume the existence of an interpretation function (see Section 4.2) and a fixed initial node state:

locale *network-with-ops* = *causal-network history fst*
for *history* :: *nat* ⇒ (*'msgid* × *'oper*) *event list* +
fixes *interp* :: *'oper* ⇒ *'state* ⇒ *'state option*
and *initial-state* :: *'state*

We have proved that the happens-before relation ≺ defined in the network is a strict partial order, so it meets the requirements of the *happens-before* locale. The lemmas and definitions of this locale are therefore available to use with the happens-before relation ≺, and we indicate these specialised theorems and definitions by prefixing their names with *hb*. Moreover, we can prove that the sequence of message deliveries at any node is consistent with ≺, that is, it satisfies the definition of *hb-consistent* given in Section 4.1 (note *hb-consistent* is now prefixed):

theorem *hb.hb-consistent* (*node-deliver-messages* (*history i*))

where *node-deliver-messages* is a function that filters the history of events at some node to return only messages that were delivered, in the order they were delivered. Now, whenever a message is delivered at some node, we can take the operation *'oper* from the message, and use its interpretation to update the state at that node. Broadcast events do not change the state, but since every message must be delivered locally at the node where it was broadcast, the state change nevertheless takes effect locally. We can then define the state of some node by using our definition of *apply-operations* from Section 4.2:

definition *apply-operations* :: (*'msgid* × *'oper*) *event list* ⇒ *'state option* **where**
apply-operations es ≡ *hb.apply-operations* (*node-deliver-messages es*) *initial-state*

So far we have no restriction on the operations that may be broadcast, except that they must be of some type *'oper*. This suffices for some replication algorithms, but many have additional requirements regarding the contents of messages that cannot be expressed in Isabelle's type system. As a general-purpose means of describing such requirements, the locale *network-with-constrained-ops* allows a replication algorithm to define a predicate *valid-msg* to specify whether a node is allowed to broadcast some message when in a particular state:

locale *network-with-constrained-ops* = *network-with-ops* +
  fixes *valid-msg* :: *'state* ⇒ (*'msgid* × *'oper*) ⇒ *bool*
  assumes *broadcast-only-valid-msgs*:
    ∃ *suf* . *pre* @ [*Broadcast m*] @ *suf* = *history i* ⟹
    ∃ *state*. *apply-operations pre* = *Some state* ∧ *valid-msg state m*

broadcast-only-valid-msgs is our final axiom, and it simply requires that if a node broadcasts some message, it must be valid according to the *valid-msg* predicate. Since the choice of messages to broadcast is under the control of the replication algorithm, and the algorithm defines this predicate, this assumption is reasonable.

Although these six axioms are simple and uncontroversial, we believe that the set of axioms could be reduced further by defining some of the aforementioned algorithms (such as vector timestamps for causal delivery, or sequence numbers for message uniqueness) within Isabelle, and proving that the algorithms guarantee the required properties within some weaker network model. However, doing so would lead us too far astray from the goal of proving the strong eventual consistency of CRDTs, so we leave it for future work.

The axioms of *network-with-constrained-ops* and its superlocales are consistent (in the sense that that we are unable to prove *False* by assuming the axioms). We demonstrate this fact by building a trivial model of *network-with-constrained-ops* within Isabelle and showing that it satisfies all of the locale's axioms. We elide these models here.

6 REPLICATED GROWABLE ARRAY

The RGA, introduced by Roh et al. [2011], is a replicated ordered list (sequence) datatype that supports *insert* and *delete* operations. It can be used for collaborative editing of text by representing a string as an ordered list of characters.

The convergence of RGA has been proved by hand in previous work (see Section 8.2); we now present the first (to our knowledge) mechanised proof that RGA satisfies the specification of SEC from Section 4. We perform this proof within the causal broadcast model defined in Section 5, and without making any assumptions beyond the six aforementioned network axioms. Since the axioms of our network model are easily justified, we have confidence in the correctness of our formalisation. Our proof makes extensive use of the general-purpose framework that we have established in the last two sections.

6.1 Specifying Insertion and Deletion

In an ordered list, each insertion and deletion operation must identify the position at which the modification should take place. In a non-replicated setting, the position is commonly expressed as an index into the list. However, the index of a list element may change if other elements are concurrently inserted or deleted earlier in the list; this is the problem at the heart of Operational Transformation (see Section 8.1). Instead of using indexes, the RGA algorithm assigns a unique, immutable identifier to each list element.

Insertion operations place the new element *after* an existing list element with a given ID, or at the head of the list if no ID is given. Deletion operations refer to the ID of the list element

that is to be deleted. However, it is not safe for a deletion operation to completely remove a list element, because then a concurrent insertion after the deleted element would not be able to locate the insertion position. Instead, the list retains *tombstones*: a deletion operation merely sets a flag on a list element to mark it as deleted, but the element actually remains in the list. A garbage collection process can be used to purge tombstones [Roh et al. 2011], but we do not consider it here.

The RGA state at each node is a list of elements. Each element is a triple consisting of the unique ID of the list element (of some type *'id*), the value inserted by the application (of some type *'v*), and a flag that indicates whether the element has been marked as deleted (of type *bool*):

```
type-synonym ('id, 'v) elt = 'id × 'v × bool
```

The *insert* function takes three parameters: the previous state of the list, the new element to insert, and optionally the ID of an existing element after which the new element should be inserted. It returns the list with the new element inserted at the appropriate position, or *None* on failure, which occurs if there was no existing element with the given ID. The function iterates over the list, and for each list element *x*, it compares the ID (the first component of the *'id* × *'v* × *bool* triple, written *fst x*) to the requested insertion position:

```
fun insert :: ('id::{linorder}, 'v) elt list ⇒ ('id, 'v) elt ⇒ 'id option ⇒ ('id, 'v) elt list option
where
  insert xs       e None     = Some (insert-body xs e) |
  insert []       e (Some i) = None |
  insert (x#xs) e (Some i) = (if fst x = i then Some (x#insert-body xs e)
                              else insert xs e (Some i) ⤜ (λt. Some (x#t)))
```

When the insertion position is found (or, in the case of insertion at the head of the list, immediately), the function *insert-body* is invoked to perform the actual insertion:

```
fun insert-body :: ('id::{linorder}, 'v) elt list ⇒ ('id, 'v) elt ⇒ ('id, 'v) elt list where
  insert-body []       e = [e] |
  insert-body (x#xs) e = (if fst x < fst e then e#x#xs else x#insert-body xs e)
```

In a non-replicated datatype it would be sufficient to insert the new element directly at the position found by the *insert* function. However, a replicated setting is more difficult, because several nodes may concurrently insert new elements at the same position, and those insertion operations may be processed in a different order by different nodes. In order to ensure that all nodes converge towards the same state (that is, the same order of list elements), we sort any concurrent insertions at the same position in descending order of the inserted elements' IDs. This sorting is implemented in *insert-body* by skipping over any elements with an ID that is greater than that of the newly inserted element (the *fst x* > *fst e* case), and then placing the new element before the first existing element with a lesser ID (the *fst x* < *fst e* case).

Note that the type of IDs is specified as *'id*::{*linorder*}, which means that we require the type *'id* to have an associated total (linear) order. *linorder* is the name of a type class supplied by the Isabelle/HOL library. This annotation is required in order to be able to perform the comparison *fst x* < *fst e* on IDs. To be precise, RGA requires the total order of IDs to be consistent with causality, which can easily be achieved using the logical timestamps defined by Lamport [1978].

The delete operation searches for the element with a given ID, and sets its flag to *True* to mark it as deleted:

fun *delete* :: (*'id*::{*linorder*}, *'v*) *elt list* ⇒ *'id* ⇒ (*'id*, *'v*) *elt list option* **where**
delete [] *i* = *None* |
delete ((*i'*, *v*, *flag*)#*xs*) *i* = (*if i'* = *i then Some* ((*i'*, *v*, *True*)#*xs*)
else delete xs i ⤜ (λ*t*. *Some* ((*i'*,*v*,*flag*)#*t*)))

Note that the operations presented here are deliberately inefficient in order to make them easier to reason about. One can see our implementations of *insert-body*, *insert*, and *delete* as functional specifications for RGAs, which could be optimised into more efficient algorithms using data refinement, if desired.

6.2 Commutativity of Insertion and Deletion

Recall from Section 4.3 that in order to prove the convergence theorem we need to show that for the datatype in question, all its concurrent operations commute. It is straightforward to demonstrate that *delete* always commutes with itself, on concurrent and non-concurrent operations alike:

lemma *delete-commutes*:
delete xs i1 ⤜ (λ*ys*. *delete ys i2*) = *delete xs i2* ⤜ (λ*ys*. *delete ys i1*)

It is a little more complex to demonstrate that two *insert* operations commute. Let *e1* and *e2* be the two new list elements being inserted, each of which is a *'id* × *'v* × *bool* triple. Further, let *i1* :: *'id option* be the position after which *e1* should be inserted (either *None* for the head of the list, or *Some i* where *i* is the ID of an existing list element), and similarly let *i2* be the position after which *e2* should be inserted. Then the two insertions commute only under certain assumptions:

lemma *insert-commutes*:
assumes *fst e1* ≠ *fst e2*
and *i1* = *None* ∨ *i1* ≠ *Some* (*fst e2*)
and *i2* = *None* ∨ *i2* ≠ *Some* (*fst e1*)
shows *insert xs e1 i1* ⤜ (λ*ys*. *insert ys e2 i2*) = *insert xs e2 i2* ⤜ (λ*ys*. *insert ys e1 i1*)

That is, *i1* cannot refer to the ID of *e2* and vice versa, and the IDs of the two insertions must be distinct. We prove later that these assumptions are indeed satisfied for all concurrent operations. Finally, *delete* commutes with *insert* whenever the element to be deleted is not the same as the element to be inserted:

lemma *insert-delete-commute*:
assumes *i2* ≠ *fst e*
shows *insert xs e i1* ⤜ (λ*ys*. *delete ys i2*) = *delete xs i2* ⤜ (λ*ys*. *insert ys e i1*)

6.3 Embedding RGA in the Network Model

In order to obtain a proof of the strong eventual consistency of RGA, we embed the insertion and deletion operations in the network model of Section 5. We first define a datatype for operations (which are sent across the network in messages), and an interpretation function as introduced in Section 4.2:

datatype (*'id*, *'v*) *operation* = *Insert* (*'id*, *'v*) *elt 'id option* | *Delete 'id*

fun *interpret-opers* :: (*'id*::*linorder*, *'v*) *operation* ⇒ (*'id*, *'v*) *elt list* ⇒ (*'id*, *'v*) *elt list option*
where
interpret-opers (*Insert e n*) *xs* = *insert xs e n* |
interpret-opers (*Delete n*) *xs* = *delete xs n*

As discussed above, the validity of operations depends on some assumptions: IDs of insertion operations must be unique, and whenever an insertion or deletion operation refers to an existing list element, that element must exist. As introduced in Section 5.4, we can describe these requirements by using a predicate to specify what messages a node is allowed to broadcast when in a particular state:

definition *valid-rga-msg* :: $('id, 'v)$ *elt list* $\Rightarrow$ $'id \times ('id::linorder, 'v)$ *operation* $\Rightarrow$ *bool* **where**
valid-rga-msg list msg $\equiv$ *case msg of*
$(i, Insert\ e\ None\ ) \Rightarrow fst\ e = i\ |$
$(i, Insert\ e\ (Some\ pos)) \Rightarrow fst\ e = i \wedge pos \in set\ (map\ fst\ list)\ |$
$(i, Delete\ pos\ ) \Rightarrow pos \in set\ (map\ fst\ list)$

We can now define RGA by extending *network-with-constrained-ops*. The interpretation function is instantiated with *interpret-opers*, the initial state with the empty list [ ], and the validity predicate with *valid-rga-msg*:

locale *rga* = *network-with-constrained-ops* - *interpret-opers* [] *valid-rga-msg*

Within this locale, we prove that whenever an insertion or deletion operation *op2* references an existing list element, there is always a prior insertion operation *op1* that created the element being referenced:

lemma *allowed-insert*:
assumes *Broadcast* $(Insert\ e\ n) \in set\ (history\ i)$
shows $n = None \vee (\exists e'\ n'.\ n = Some\ (fst\ e') \wedge$
$Deliver\ (Insert\ e'\ n') \sqsubset^i Broadcast\ (Insert\ e\ n))$

lemma *allowed-delete*:
assumes *Broadcast* $(Delete\ x) \in set\ (history\ i)$
shows $\exists n'\ v\ b.\ Deliver\ (Insert\ (x, v, b)\ n') \sqsubset^i Broadcast\ (Delete\ x)$

Since the network ensures causally ordered delivery, all nodes must deliver the insertion *op1* before the dependent operation *op2*. Hence we show that in all cases where operations do not commute, one operation happens before another. Conversely, whenever operations are concurrent, we show that they commute:

theorem *concurrent-operations-commute*:
shows *hb.concurrent-ops-commute* (*node-deliver-messages* (*history i*))

Furthermore, although the type signature of the interpretation function allows an operation to fail by returning *None*, we can prove that this failure case is never reached in any execution of the network:

theorem *apply-operations-never-fails*:
shows *hb.apply-operations* (*node-deliver-messages* (*history i*)) $\neq$ *None*

It is now easy to show that the *rga* locale satisfies all of the requirements of the abstract specification *strong-eventual-consistency* (Section 4.4), which demonstrates formally that RGA provides SEC.

7 TWO OTHER CRDTS: COUNTER AND SET

To demonstrate that our proof framework provides reusable components that significantly simplify SEC proofs for new algorithms, we show proofs for two other well-known operation-based CRDTs: the Observed-Remove Set (ORSet) and the Increment-Decrement Counter as described by Shapiro et al. [2011a]. These proofs build upon the abstract convergence theorem and the network model

of Sections 4 and 5, and reuse some of the proof techniques developed in the formalisation of RGA in Section 6.

As these proofs leverage the framework's machinery and proof techniques, we were able to develop them very quickly: the counter was proved correct in a matter of minutes, and the specification and correctness proof of the ORSet was done in about four hours by one of the authors, an Isabelle novice who had never used any proof assistant software prior to the start of this project. Although these anecdotes do not constitute a formal evaluation of ease of use, we take them as being an encouraging sign.

7.1 Increment-Decrement Counter

The Increment-Decrement Counter is perhaps the simplest CRDT, and a paradigmatic example of a replicated data structure with commutative operations. As the name suggests, the data structure supports two operations: *increment* and *decrement* which respectively increment and decrement a shared integer counter:

datatype *operation* = *Increment* | *Decrement*

The interpretation function for these two operations is straightforward:

fun *counter-op* :: *operation* ⇒ *int* ⇒ *int option* **where**
counter-op Increment x = *Some* (*x* + *1*) |
counter-op Decrement x = *Some* (*x* − *1*)

Note that the operations do not fail on under- or overflow, as they are defined on a type of unbounded (mathematical) integers. We could also have implemented the counter using fixed-size integers—e.g. signed 32- or 64-bit machine words—with wrap-around on overflow, which would not have impacted the proofs. Showing commutativity of the operations is an easy exercise in applying Isabelle's proof automation:

lemma *counter-op x* ▷ *counter-op y* = *counter-op y* ▷ *counter-op x*

Unlike more complex CRDTs such as RGA, the operations of the increment-decrement counter commute unconditionally. As a result, this CRDT converges in any asynchronous broadcast network, without requiring causally ordered delivery. For simplicity, we define *counter* as a simple extension of our existing *network-with-ops* locale. We need only specify the interpretation function and the initial state 0:

locale *counter* = *network-with-ops* - *counter-op 0*

It is then straightforward to prove that *counter* is a sublocale of *strong-eventual-consistency* (see Section 4.4), from which we obtain concrete convergence and progress theorems for the counter CRDT.

7.2 Observed-Remove Set

The Observed-Remove Set (ORSet) is a well-known CRDT for implementing replicated sets, supporting two operations: *adding* and *removing* arbitrary elements in the set. It has mostly been studied in its state-based formulation [Bieniusa et al. 2012a,b; Brown et al. 2014; Zeller et al. 2014], but here we use the operation-based formulation as described by Shapiro et al. [2011a]. The name derives from the fact that the algorithm "observes" the state of a node when removing an element from the set, as explained below.

We start by defining the two possible operations of the datatype:

datatype (*'id*, *'a*) *operation* = *Add 'id 'a* | *Rem* (*'id set*) *'a*

Here, *'id* is an abstract type of message identifiers, and the type variable *'a* represents the type of values that the application wishes to add to the set. When an element *e* is added to the set, the operation *Add i e* is tagged with a unique identifier *i* in order to distinguish it from other operations that may concurrently add the same element *e* to the set. When an element *e* is removed from the set, the operation *Rem is e* contains a set of identifiers *is*, identifying all of the additions of that element that causally happened-before the removal.

The state maintained at each node is a function that maps each element *'a* to the set of identifiers of operations that have added that element:

```
type-synonym ('id, 'a) state = 'a ⇒ 'id set
```

We consider an element *'a* to be a member of the ORSet if the set of addition identifiers is non-empty. The initial state of a node—the empty ORSet—is then simply $\lambda x.\ \{\}$, i.e. the function that maps every possible element *'a* to the empty set of identifiers {}.

When interpreting an *Add* operation, we must add the identifier of that operation to the node state. When interpreting a *Rem* operation, we must update the node state to remove all causally prior *Add* identifiers. If there are no concurrent additions of the same element, this has the effect of making the set of identifiers for that element empty, and thus considering the element as no longer being in the set. We express this as follows:

```
definition op-elem :: ('id, 'a) operation ⇒ 'a where
  op-elem oper ≡ case oper of Add i e ⇒ e | Rem is e ⇒ e

definition interpret-op :: ('id, 'a) operation ⇒ ('id, 'a) state ⇒ ('id, 'a) state option where
  interpret-op oper state ≡
    let before = state (op-elem oper);
        after  = case oper of Add i  e ⇒ before ∪ {i} |
                              Rem is e ⇒ before − is
    in Some (state ((op-elem oper) := after))
```

Here, *state*((*op-elem oper*) := *after*) is Isabelle's syntax for pointwise function update. A remove operation effectively undoes the prior additions of that element of the set, while leaving any concurrent or later additions of the same element unaffected. When an element *e* is concurrently added and removed, the identifier of the addition operation will not be in the identifier set of the removal operation. As a result, the final state after interpreting these two operations will contain the element *e*.

As the last part of specifying ORSet, we must require that *Add* and *Rem* use identifiers correctly. We require the identifier of *Add* operations to be globally unique, which we can express by making it equal to the unique ID of the message containing the operation (Section 5.2). A *Rem* operation must contain the set of addition identifiers in the node state at the moment when the *Rem* operation was issued. We express these constraints using the following *valid-behaviours* predicate:

```
definition valid-behaviours :: ('id, 'a) state ⇒ 'id × ('id, 'a) operation ⇒ bool where
  valid-behaviours state msg ≡
    case msg of (i, Add j  e) ⇒ i = j |
                (i, Rem is e) ⇒ is = state e
```

To prove that ORSet satisfies the specification of strong eventual consistency, we follow the same pattern as before. We first define a locale *orset* that extends *network-with-constrained-ops*:

```
locale orset = network-with-constrained-ops - interpret-op (λx. {}) valid-behaviours
```

Recall the requirements of the *strong-eventual-consistency* specification (Section 4.4). Firstly, we must show that *apply-operations* never fails, which is easy in this case, since the interpretation function never returns *None*:

theorem *apply-operations-never-fails*:
  shows *hb.apply-operations* (*node-deliver-messages* (*history i*)) ≠ *None*

Secondly, we must show that concurrent operations commute. Isabelle's proof automation can easily verify that two addition operations commute unconditionally, as do two removal operations:

lemma *add-add-commute*:
  shows ⟨*Add i1 e1*⟩ ▷ ⟨*Add i2 e2*⟩ = ⟨*Add i2 e2*⟩ ▷ ⟨*Add i1 e1*⟩

lemma *rem-rem-commute*:
  shows ⟨*Rem i1 e1*⟩ ▷ ⟨*Rem i2 e2*⟩ = ⟨*Rem i2 e2*⟩ ▷ ⟨*Rem i1 e1*⟩

However, add and remove operations commute only if the identifier of the addition is not one of the identifiers affected by the removal:

lemma *add-rem-commute*:
  assumes *i* ∉ *is*
  shows ⟨*Add i e1*⟩ ▷ ⟨*Rem is e2*⟩ = ⟨*Rem is e2*⟩ ▷ ⟨*Add i e1*⟩

Proving that the assumption *i* ∉ *is* holds for all concurrent *Add* and *Rem* operations is a bit more laborious. We define *added-ids* to be the identifiers of all *Add* operations in a list of delivery events, even if those elements are subsequently removed. Then we prove that the set of identifiers in the node state is a subset of *added-ids* (since *Add* operations only ever add identifiers to the node state, and *Rem* operations only ever remove identifiers):

lemma *apply-operations-added-ids*:
  assumes ∃ *suf* . *pre* @ *suf* = *history i*
    and *apply-operations pre* = *Some state*
  shows *state e* ⊆ *set* (*added-ids pre e*)

From this lemma, we deduce that when an *Add* and a *Rem* operation are concurrent, the identifier of the *Add* cannot be in the set of identifiers removed by *Rem*:

lemma *concurrent-add-remove-independent*:
  assumes (*Add i e1*) ∥ (*Rem is e2*)
    and *Add i e1* ∈ *set* (*node-deliver-messages* (*history j*))
    and *Rem is e2* ∈ *set* (*node-deliver-messages* (*history j*))
  shows *i* ∉ *is*

Now that we have proved that the assumption of *add-rem-commute* holds for all concurrent operations, we can deduce that all concurrent operations commute:

theorem *concurrent-operations-commute*:
  shows *hb.concurrent-ops-commute* (*node-deliver-messages* (*history i*))

Having proved *apply-operations-never-fails* and *concurrent-operations-commute*, we can now immediately prove that *orset* is a sublocale of *strong-eventual-consistency*, using the familiar proof pattern from the other CRDTs. This proof produces concrete convergence and progress theorems for the ORSet.

8 RELATED WORK

In a system where different nodes may concurrently perform updates without coordinating with each other, strong eventual consistency requires a conflict resolution algorithm to reconcile concurrent updates. In some cases, a trivial algorithm is used, for example:

User-defined conflict resolution: Some systems store all conflicting versions of the data, and either leave it for manual resolution by a user, or invoke a user-defined merge function. However, manual resolution is an unacceptable burden for the user in many applications, and defining merge functions in application code is error-prone; for example, DeCandia et al. [2007] describe a shopping cart anomaly at Amazon that arose due to poor conflict resolution.

Last write wins (LWW): Each version of the data structure is assigned a unique timestamp. When there is a conflict, the system picks the version with the highest timestamp and discards other versions. Apache Cassandra takes this approach, for example [Kingsbury 2013]. Although LWW achieves convergence, it does so at the cost of losing user input, which is often unacceptable.

However, there are also algorithms that achieve convergence automatically, without discarding updates. In Section 8.1 we summarise two main lines of work, CRDTs and OT, which have the same fundamental goal of conflict resolution and convergence, but which take different approaches towards achieving it. In Section 8.2 we discuss existing work on formal verification of those algorithms.

8.1 Operational Transformation and Conflict-free Replicated Data Types

Algorithms for achieving strong eventual consistency have been studied extensively in the context of collaborative editing and groupware. The *operational transformation* (OT) approach was developed to allow several users to concurrently modify a document, applying edits immediately to their local copy, propagating them asynchronously to other users, and automatically resolving any conflicts such that all nodes converge towards the same state.

OT algorithms for text documents include dOPT [Ellis and Gibbs 1989], Jupiter [Nichols et al. 1995], adOPTed [Ressel et al. 1996], GOT [Sun et al. 1998], GOTO [Sun and Ellis 1998], SOCT2 [Suleiman et al. 1997, 1998], SOCT3/4 [Vidot et al. 2000], IMOR [Imine et al. 2003], SDT [Li and Li 2004, 2008], and TTF [Oster et al. 2006a]. The approach has also been generalised to other data structures such as XML trees [Davis et al. 2002; Ignat and Norrie 2003; Jungnickel and Herb 2015] and vector graphics documents [Sun and Chen 2002].

Many OT algorithms assume that operations are sequenced through a central server and delivered to all clients in the same order. This design was originally pioneered by the Jupiter system [Nichols et al. 1995] and is now used by all widely-deployed OT-based collaboration systems, including Google Docs [Day-Richter 2010], Microsoft Word Online, Etherpad [AppJet, Inc. 2011], Google Wave/Apache Wave [Wang et al. 2015], and Novell Vibe [Spiewak 2010].

OT algorithms track the version of the document in which each operation applies, and if an operation needs to be applied to a later document version (because another, concurrent operation has already been applied), the operation must be transformed. Ressel et al. [1996] introduced two properties that the OT transformation function must satisfy, which are known as TP_1 and TP_2.

Given two concurrent operations x and y that modify the same initial state, TP_1 requires that y can be transformed into an operation y' that performs an equivalent modification on a state where x has already been applied, and vice versa, such that $x \circ y' = y \circ x'$. Systems that sequence operations through a central server need only satisfy TP_1 because each client only needs to reorder its operations with respect to the server's operation sequence.

However, as discussed in the introduction, we are interested in replication algorithms for decentralised systems without any central server. If there are three concurrent operations x, y, and z that modify the same initial state, and those operations can be applied in any order, TP_1 does not suffice, and the TP_2 property must also be satisfied. TP_2 requires that if transformations of x and y are applied in either order, the same transformation of z can be applied to the result: $x \circ y' \circ z' = y \circ x' \circ z'$. Since transformed operations may be different from original operations, this property demands much more than just commutativity, making it difficult to implement correctly. We show in Section 8.2 how almost all OT algorithms have failed to satisfy TP_2.

Instead, *conflict-free replicated data types* (CRDTs) have been developed to achieve SEC in decentralised systems. As we have noted, CRDTs make operations commutative by design by attaching additional metadata to the data structure. To propagate changes between nodes, a CRDT either captures every update as an operation and broadcasts it to other nodes (an *operation-based* CRDT), or periodically broadcasts its entire node state (a *state-based* CRDT). Operation-based CRDTs require operations to commute; state-based CRDTs require a merge function over a join-semilattice, allowing two states to be combined such that the result reflects changes made in both nodes [Shapiro et al. 2011a,b]. State-based CRDTs have been deployed commercially in the Riak database [Brown et al. 2014], but in this work we focus on operation-based algorithms, because all known CRDTs for text editing and ordered lists are operation-based.

As with OT, several CRDTs for text documents have been developed, including RGA [Roh et al. 2011], Treedoc [Preguiça et al. 2009], WOOT [Oster et al. 2006b], Logoot [Weiss et al. 2010], and LSEQ [Nédelec et al. 2016, 2013]. Other datatypes include registers and counters [Shapiro et al. 2011a,b], maps [Baquero et al. 2016], sets [Bieniusa et al. 2012a,b], XML [Martin et al. 2010], and JSON trees [Kleppmann and Beresford 2017]. Cloud types [Burckhardt et al. 2012] have similarities to CRDTs, using a relational data model.

8.2 Formal Verification

The history of algorithms for achieving convergence in a distributed setting has been fraught with difficulty. Informal reasoning has repeatedly produced approaches that fail to converge in certain scenarios, and even several formal "proofs" later turned out to be false, as explained below. For OT, as described in Section 8.1, convergence in this setting requires satisfying the TP_1 and TP_2 properties. While TP_1 has proved to be readily achievable in practice, and all the aforementioned widely-deployed OT systems rely on it, the TP_2 property has been a significant source of problems.

The original peer-reviewed publications of dOPT, adOPTed, IMOR, SOCT2, and SDT all claimed that their transformation functions satisfied TP_2, but those claims were subsequently shown to be false by giving counter-examples [Imine et al. 2003, 2006; Oster et al. 2005]. In the case of dOPT and adOPTed, the TP_2 claim had originally been asserted without proof. In the case of SOCT2 and SDT, there were hand-written "proofs" that later turned out to be incorrect. For IMOR and SOCT2, there had even been machine-checked "proofs" [Imine et al. 2003], but Oster et al. [2005] showed that they were also invalid because they had made incorrect assumptions.

Randolph et al. [2015] have even shown that in the classic formulation of OT it is impossible to achieve TP_2. To our knowledge, TTF is at present the only TP_2-claiming OT algorithm for which no counter-example is known, and it circumvents the impossibility result of Randolph et al. [2015] by using a different formulation of the transformation [Levien 2016; Oster et al. 2006a].

Formal proofs of the TP_1 property have been more successful: Sinchuk et al. [2016] use Coq to verify that their algorithm satisfies TP_1, and Jungnickel and Herb [2015] use Isabelle/HOL for the same purpose. For CRDTs, the only machine-checked verification of which we are aware is an Isabelle formalisation of state-based sets, registers, and counters by Zeller et al. [2014]; this work does not consider any list datatypes or any operation-based CRDTs.

The convergence of the RGA CRDT for ordered lists, which we study in this paper, has previously been demonstrated in handwritten proofs [Attiya et al. 2016; Kleppmann and Beresford 2017; Roh et al. 2009]. Although we have no reason to doubt the correctness of those proofs, the historic experience with TP_2 makes us wary of claims whose assumptions and reasoning process have not been checked rigorously. Other authors have also pointed out that handwritten proofs are laborious and difficult to check by hand [Li and Li 2008, 2005].

To our knowledge, our work is the first mechanised proof of operation-based CRDTs in general, and of any ordered list CRDT in particular. As Oster et al. [2005] have demonstrated, machine-checked proofs are not immune to errors that are due to false assumptions. To avoid this trap, we prove not only the commutativity of operations (which is subject to certain assumptions), but also that those assumptions are guaranteed to hold in all behaviours of our network model. The network model in turn is specified by a small set of axioms that are not specific to any particular CRDT, and whose correctness can be robustly defended (see Section 5).

Burckhardt et al. [2014] present a framework for specifying and reasoning about replicated datatypes, but do not support mechanised proofs at present, and use different techniques to those described in this paper.

More generally, applying verification techniques to distributed systems is an active area of research. Interactive theorem provers [Charron-Bost et al. 2011; Debrat and Merz 2012; Wilcox et al. 2015], model checkers [Azmy et al. 2016; Johnson et al. 2004], and formal specification tools [Andriamiarina et al. 2014; Tounsi et al. 2013, 2016] have all been adopted for the verification and specification of distributed systems, algorithms, and protocols. Interestingly, recent empirical work [Fonseca et al. 2017] has found that several verified distributed systems contain critical bugs that can cause runtime crashes or the return of incorrect results to clients—violating the supposed guarantees offered by their correctness theorems. A common cause of these bugs is a mismatch between the assumptions made when verifying the system and the guarantees offered by the underlying network, libraries, and operating system infrastructure upon which they are built. We see this as compelling evidence that verifying distributed systems starting from a model of the network and building up, as we do in this work, is a robust approach to distributed systems verification.

9 DISCUSSION

The convergence proofs for all of our CRDT implementations follow the same structure. First we define the type of local state at each node, and the types of operations that may be invoked to modify the state. When one node invokes an operation, it is broadcast to other nodes using our network model, implemented as a specialisation of the *network-with-(constrained-)ops* locale. An interpretation function is called whenever a message containing an operation is delivered to a node, and it transforms that node's local state to incorporate the operation. To demonstrate convergence, we must show that all operations commute with themselves and with each other, subject to certain assumptions. Next, we must prove that those assumptions are always satisfied by any concurrent operations in the network. Finally, a CRDT must demonstrate that applying an operation never fails, provided that the operation was constructed according to the definition of the algorithm.

When these proof obligations have been met, we are able to conclude that the algorithm satisfies our abstract specification *strong-eventual-consistency*, from which we obtain convergence and progress theorems for the replicated datatype. The abstract specification is independent of any particular network model or replication algorithm, and we assert that it constitutes a general but precise definition of strong eventual consistency. As this recurring pattern demonstrates, we have not only isolated reusable lemmas and models of networks, but also a proof strategy that algorithm designers can use to obtain a convergence theorem for their operation-based CRDT.

Over half of our development—the network model, convergence proof, and lemmas—is independent of any particular CRDT and is reusable in future proofs. In particular, we use: around 620 lines for our network model, around 380 lines for the abstract convergence proof, 775 lines for the RGA proof, around 270 lines for the ORSet proof, and around 55 lines for the Counter proof. Additional shared code consists of around 170 lines of source. Definitions and proofs of correctness for our three CRDT implementations are pleasingly short: all three are shown to be convergent in fewer than 800 lines of source, using the proof strategy described above.

Lastly, all three of our CRDT implementations are "executable" in the sense that we can use Isabelle's code generation mechanism to obtain OCaml (or Scala, SML, and Haskell) implementations from our definitions [Haftmann and Nipkow 2010]. We have run an extraction of one of our CRDTs—the counter—on a simple network of n nodes, communicating over TCP links between machines. The purpose of this extraction is to demonstrate that we have not used any uncomputable functions in our Isabelle definitions. We leave a detailed empirical evaluation of the algorithms, such as tests of their performance and fault tolerance, for future work.

10 CONCLUSION

In this paper we adopted a "foundational" approach to proving the correctness of a class of SEC algorithms: Conflict-free Replicated Datatypes (CRDTs). In our work, we made no axiomatic assumptions related to any individual algorithm; instead, we constructed a formal, realistic model of a computer network that may delay, drop, or reorder messages sent between computers—a model well known within the distributed systems community, with defensible axioms. In addition, we isolated a formal specification of SEC, and showed that any algorithm that meets the preconditions of this specification must converge. Our network model, the SEC specification, and a library of lemmas form a framework with which one can prove convergence for concrete CRDT implementations.

As a case study in applying our framework, we formalised three operation-based CRDTs—the Replicated Growable Array, the Observed-Remove Set, and an increment-decrement Counter. For each algorithm we proved that its assumptions are satisfied in all possible network behaviours, and that it satisfies the preconditions of our abstract SEC specification, obtaining a guarantee that each of our three CRDT implementations converge. Moreover, these convergence theorems were obtained using only a thin layer of CRDT-specific code, using a fixed pattern of proof in all three cases. Since informal reasoning about convergent replication algorithms has been shown to difficult and error-prone, as exemplified by various failed proofs (Section 8.2) and the "a bit of a mystery" RGA [Attiya et al. 2016], we believe that this formal verification is particularly important.

Our work has been motivated by the desire to support a wider range of strong eventual consistency algorithms in distributed systems. Operational Transformation algorithms based on the TP_1 property alone are widely used in systems such as Google Docs (see Section 8.2), but they require clients to communicate all changes via a single central server. On the other hand, state-based CRDTs are used in systems such as Riak. Both of these approaches are limiting. In the case of Operational Transformation, the requirement for a central server increases the risk of faults, and rules out direct communication between co-located devices. In the case of state-based CRDTs, efficient algorithms to support complex data structures such as ordered lists are yet to be found. Consequently, our approach provides the groundwork for a new generation of robust, collaborative applications that operate on complex data structures in a peer-to-peer setting, which are likely to become increasingly important as mobile devices become ubiquitous.

Although we have focussed on operation-based algorithms in this work, we speculate that our framework is also amenable to formalising Operational Transformation algorithms and state-based CRDTs. We also speculate that our framework could be used to demonstrate the equivalence of classes of strong eventual consistency algorithms. We leave this to future work.

ACKNOWLEDGMENTS

The authors wish to acknowledge the support of the EPSRC "REMS: Rigorous Engineering for Mainstream Systems" programme grant (EP/K008528), the EPSRC "Interdisciplinary Centre for Finding, Understanding and Countering Crime in the Cloud" grant (EP/M020320), and The Boeing Company. We thank Peter Sewell, Alan Mycroft, and the anonymous reviewers for their helpful feedback on this paper.

REFERENCES

Akka 2017. The Akka actor framework for the Java Virtual Machine. (2017). http://www.akka.io/ Accessed April 2017.

Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient State-Based CRDTs by Delta-Mutation. In *International Conference on Networked Systems (NETYS)*. https://doi.org/10.1007/978-3-319-26850-7_5

Manamiary Bruno Andriamiarina, Dominique Méry, and Neeraj Kumar Singh. 2014. Analysis of Self-★ and P2P Systems Using Refinement. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*. 117–123. https://doi.org/10.1007/978-3-662-43652-3_9

AppJet, Inc. 2011. Etherpad and EasySync Technical Manual. (March 2011). https://github.com/ether/etherpad-lite/blob/e2ce9dc/doc/easysync/easysync-full-description.pdf

Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. 2016. Specification and Complexity of Collaborative Text Editing. In *ACM Symposium on Principles of Distributed Computing (PODC)*. 259–268. https://doi.org/10.1145/2933057.2933090

Hagit Attiya, Faith Ellen, and Adam Morrison. 2015. Limitations of Highly-Available Eventually-Consistent Data Stores. In *ACM Symposium on Principles of Distributed Computing (PODC)*. https://doi.org/10.1145/2767386.2767419

Noran Azmy, Stephan Merz, and Christoph Weidenbach. 2016. A Rigorous Correctness Proof for Pastry. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings*. 86–101. https://doi.org/10.1007/978-3-319-33600-8_5

Peter Bailis and Ali Ghodsi. 2013. Eventual Consistency Today: Limitations, Extensions, and Beyond. *ACM Queue* 11, 3 (March 2013). https://doi.org/10.1145/2460276.2462076

Peter Bailis and Kyle Kingsbury. 2014. The Network is Reliable. *ACM Queue* 12, 7 (July 2014). https://doi.org/10.1145/2639988.2639988

Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. 2016. The problem with embedded CRDT counters and a solution. In *2nd Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC)*. https://doi.org/10.1145/2911151.2911159

Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making Operation-based CRDTs Operation-based. In *14th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*. 126–140. https://doi.org/10.1007/978-3-662-43352-2_11

Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012a. Brief Announcement: Semantics of Eventually Consistent Replicated Sets. In *26th International Symposium on Distributed Computing (DISC)*. https://doi.org/10.1007/978-3-642-33651-5_48

Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012b. *An Optimized Conflict-free Replicated Set*. Technical Report RR-8083. http://arxiv.org/abs/1210.3368

Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. 2014. Riak DT map: a composable, convergent replicated dictionary. In *1st Workshop on Principles and Practice of Eventual Consistency (PaPEC)*. https://doi.org/10.1145/2596631.2596633

Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Foundations and Trends in Programming Languages* 1, 1-2 (Oct. 2014), 1–150. https://doi.org/10.1561/2500000011

Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. 2012. Cloud Types for Eventual Consistency. In *26th European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-642-31057-7_14

Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. 2014. Replicated Data Types: Specification, Verification, Optimality. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 271–284. https://doi.org/10.1145/2535838.2535848

Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (second ed.). Springer. https://doi.org/10.1007/978-3-642-15260-3

Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. 2011. Formal Verification of Consensus Algorithms Tolerating Malicious Faults. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. 120–134. https://doi.org/10.1007/978-3-642-24550-3_11

Susan B Davidson, Hector Garcia-Molina, and Dale Skeen. 1985. Consistency in Partitioned Networks. *Comput. Surveys* 17,

3 (Sept. 1985), 341–370. https://doi.org/10.1145/5505.5508

Aguido Horatio Davis, Chengzheng Sun, and Junwei Lu. 2002. Generalizing Operational Transformation to the Standard General Markup Language. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 58–67. https://doi.org/10.1145/587078.587088

John Day-Richter. 2010. What's different about the new Google Docs: Making collaboration fast. (Sept. 2010). https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html

Henri Debrat and Stephan Merz. 2012. Verifying Fault-Tolerant Distributed Algorithms in the Heard-Of Model. *Archive of Formal Proofs* 2012 (2012). https://www.isa-afp.org/entries/Heard_Of.shtml

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *21st ACM Symposium on Operating Systems Principles (SOSP)*. 205–220. https://doi.org/10.1145/1294261.1294281

Clarence Ellis and S J Gibbs. 1989. Concurrency Control in Groupware Systems. In *ACM International Conference on Management of Data (SIGMOD)*. 399–407. https://doi.org/10.1145/67544.66963

Colin J Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*. 56–66.

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. 328–343. https://doi.org/10.1145/3064176.3064183

Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. A framework for establishing Strong Eventual Consistency for Conflict-free Replicated Datatypes. *Archive of Formal Proofs* (July 2017). http://isa-afp.org/entries/CRDT.shtml, Formal proof development.

Florian Haftmann and Tobias Nipkow. 2010. Code Generation via Higher-Order Rewrite Systems. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*. 103–117. https://doi.org/10.1007/978-3-642-12251-4_9

Florian Haftmann and Makarius Wenzel. 2008. Local Theory Specifications in Isabelle/Isar. In *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*. 153–168. https://doi.org/10.1007/978-3-642-02444-3_10

Claudia-Lavinia Ignat and Moira C Norrie. 2003. Customizable Collaborative Editor Relying on treeOPT Algorithm. In *8th European Conference on Computer-Supported Cooperative Work (ECSCW)*. 315–334. https://doi.org/10.1007/978-94-010-0068-0_17

Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. 2003. Proving Correctness of Transformation Functions in Real-Time Groupware. In *8th European Conference on Computer-Supported Cooperative Work (ECSCW)*. 277–293. https://doi.org/10.1007/978-94-010-0068-0_15

Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. 2006. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science* 351, 2 (Feb. 2006), 167–183. https://doi.org/10.1016/j.tcs.2005.09.066

James E. Johnson, David E. Langworthy, Leslie Lamport, and Friedrich H. Vogt. 2004. Formal Specification of a Web Services Protocol. *Electr. Notes Theor. Comput. Sci.* 105 (2004), 147–158. https://doi.org/10.1016/j.entcs.2004.02.022

Tim Jungnickel and Tobias Herb. 2015. TP1-valid Transformation Functions for Operations on ordered n-ary Trees. arXiv:1512.05949. (Dec. 2015). https://arxiv.org/abs/1512.05949

Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. 1999. Locales - A Sectioning Concept for Isabelle. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*. 149–166. https://doi.org/10.1007/3-540-48256-3_11

Kyle Kingsbury. 2013. Jepsen: Cassandra. (Sept. 2013). https://aphyr.com/posts/294-jepsen-cassandra Accessed April 2017.

Martin Kleppmann and Alastair R Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* (April 2017). https://doi.org/10.1109/TPDS.2017.2697382

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

João Leitão, José Pereira, and Luís Rodrigues. 2007. HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 419–429. https://doi.org/10.1109/DSN.2007.56

Raph Levien. 2016. Towards a unified theory of Operational Transformation and CRDT. (July 2016). https://medium.com/@raphlinus/towards-a-unified-theory-of-operational-transformation-and-crdt-70485876f72f

Du Li and Rui Li. 2004. Preserving operation effects relation in group editors. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 457–466. https://doi.org/10.1145/1031607.1031683

Du Li and Rui Li. 2008. An Approach to Ensuring Consistency in Peer-to-Peer Real-Time Group Editors. *Computer Supported Cooperative Work (CSCW)* 17, 5 (Dec. 2008), 553–611. https://doi.org/10.1007/s10606-005-9009-5

Rui Li and Du Li. 2005. A landmark-based transformation approach to concurrency control in group editors. In *International ACM SIGGROUP Conference on Supporting Group Work*. 284–293. https://doi.org/10.1145/1099203.1099252

Stéphane Martin, Pascal Urso, and Stéphane Weiss. 2010. Scalable XML Collaborative Editing with Undo. In *On the Move to Meaningful Internet Systems (OTM)*. 507–514. https://doi.org/10.1007/978-3-642-16934-2_37

Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: a language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*. 184–195. https://doi.org/10.1145/2790449.2790525

Brice Nédelec, Pascal Molli, and Achour Mostefaoui. 2016. CRATE: Writing Stories Together with our Browsers. In *25th International World Wide Web Conference (WWW)*. 231–234. https://doi.org/10.1145/2872518.2890539

Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In *13th ACM Symposium on Document Engineering (DocEng)*. 37–46. https://doi.org/10.1145/2494266.2494278

David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. 1995. High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System. In *8th Annual ACM Symposium on User Interface Software and Technology (UIST)*. 111–120. https://doi.org/10.1145/215585.215706

Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics - With Isabelle/HOL*. Springer. https://doi.org/10.1007/978-3-319-10542-0

Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. 2006a. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In *2nd International Conference on Collaborative Computing (CollaborateCom)*. https://doi.org/10.1109/COLCOM.2006.361867

Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2005. *Proving correctness of transformation functions in collaborative editing systems*. Technical Report RR-5795. https://hal.inria.fr/inria-00071213/

Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006b. Data Consistency for P2P Collaborative Editing. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. https://doi.org/10.1145/1180875.1180916

Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. 2009. A commutative replicated data type for cooperative editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS)*. https://doi.org/10.1109/ICDCS.2009.20

Aurel Randolph, Hanifa Boucheneb, Abdessamad Imine, and Alejandro Quintero. 2015. On Synthesizing a Consistent Operational Transformation Approach. *IEEE Trans. Comput.* 64, 4 (April 2015), 1074–1089. https://doi.org/10.1109/TC.2014.2308203

Michel Raynal and Mukesh Singhal. 1996. Logical time: capturing causality in distributed systems. *IEEE Computer* 29, 2 (Feb. 1996), 49–56. https://doi.org/10.1109/2.485846

Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuer. 1996. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 288–297. https://doi.org/10.1145/240080.240305

Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel and Distrib. Comput.* 71, 3 (2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

Hyun-Gul Roh, Jin-Soo Kim, Joonwon Lee, and Seungryoul Maeng. 2009. *Optimistic Operations for Replicated Abstract Data Types*. Technical Report CS/TR-2009-318. KAIST.

Reinhard Schwarz and Friedemann Mattern. 1994. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing* 7, 3 (March 1994), 149–174. https://doi.org/10.1007/BF02277859

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011a. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Technical Report 7506. INRIA.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011b. Conflict-free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

Justin Sheehy. 2015. There is No Now: Problems with simultaneity in distributed systems. *ACM Queue* 13, 3 (March 2015). https://doi.org/10.1145/2733108

Sergey Sinchuk, Pavel Chuprikov, and Konstantin Solomatov. 2016. Verified Operational Transformation for Trees. In *7th International Conference on Interactive Theorem Proving (ITP)*. https://doi.org/10.1007/978-3-319-43144-4_22

Daniel Spiewak. 2010. Understanding and Applying Operational Transformation. (May 2010). http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation

Maher Suleiman, Michèle Cart, and Jean Ferrié. 1997. Serialization of concurrent operations in a distributed collaborative environment. In *International Conference on Supporting Group Work (GROUP)*. 435–445. https://doi.org/10.1145/266838.267369

Maher Suleiman, Michèle Cart, and Jean Ferrié. 1998. Concurrent operations in a distributed and mobile collaborative environment. In *14th International Conference on Data Engineering (ICDE)*. 36–45. https://doi.org/10.1109/ICDE.1998.

655755

Chengzheng Sun and David Chen. 2002. Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems. *ACM Transactions on Computer-Human Interaction (TOCHI)* 9, 1 (March 2002), 1–41. https://doi.org/10.1145/505151.505152

Chengzheng Sun and Clarence Ellis. 1998. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 59–68. https://doi.org/10.1145/289444.289469

Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. 1998. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction (TOCHI)* 5, 1 (1998), 63–108. https://doi.org/10.1145/274444.274447

Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *3rd International Conference on Parallel and Distributed Information Systems (PDIS)*. 140–149. https://doi.org/10.1109/PDIS.1994.331722

Mohamed Tounsi, Mohamed Mosbah, and Dominique Méry. 2013. From Event-B Specifications to Programs for Distributed Algorithms. In *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Hammamet, Tunisia, June 17-20, 2013*. 104–109. https://doi.org/10.1109/WETICE.2013.44

Mohamed Tounsi, Mohamed Mosbah, and Dominique Méry. 2016. From Event-B specifications to programs for distributed algorithms. *IJAACS* 9, 3/4 (2016), 223–242. https://doi.org/10.1504/IJAACS.2016.079623

Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. 2000. Copies convergence in a distributed real-time collaborative environment. In *ACM Conference on Computer Supported Cooperative Work (CSCW)*. 171–180. https://doi.org/10.1145/358916.358988

Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (Jan. 2009), 40–44. https://doi.org/10.1145/1435417.1435432

David Wang, Alex Mah, Soren Lassen, and Sam Thorogood. 2015. Apache Wave (incubating) Protocol Documentation, Release 0.4. Apache Software Foundation. (Aug. 2015). https://people.apache.org/~al/wave_docs/ApacheWaveProtocol-0.4.pdf

Stéphane Weiss, Pascal Urso, and Pascal Molli. 2010. Logoot-Undo: Distributed Collaborative Editing System on P2P networks. *IEEE Transactions on Parallel and Distributed Systems* 21, 8 (Jan. 2010), 1162–1174. https://doi.org/10.1109/TPDS.2009.173

Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. 2008. The Isabelle Framework. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. 33–38. https://doi.org/10.1007/978-3-540-71067-7_7

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 357–368. https://doi.org/10.1145/2737924.2737958

Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. 2014. Formal Specification and Verification of CRDTs. In *34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems (FORTE)*. https://doi.org/10.1007/978-3-662-43613-4_3