VeriFx: Correct Replicated Data Types for the Masses

Kevin De Porre
Vrije Universiteit Brussel, Belgium

Carla Ferreira
NOVA School of Science and Technology, Caparica, Portugal

Elisa Gonzalez Boix
Vrije Universiteit Brussel, Belgium

Abstract

Distributed systems adopt weak consistency to ensure high availability and low latency, but state convergence is hard to guarantee due to conflicts. Experts carefully design replicated data types (RDTs) that resemble sequential data types and embed conflict resolution mechanisms that ensure convergence. Designing RDTs is challenging as their correctness depends on subtleties such as the ordering of concurrent operations. Currently, researchers manually verify RDTs, either by paper proofs or using proof assistants. Unfortunately, paper proofs are subject to reasoning flaws and mechanized proofs verify a formalization instead of a real-world implementation. Furthermore, writing mechanized proofs is reserved for verification experts and is extremely time-consuming. To simplify the design, implementation, and verification of RDTs, we propose VeriFx, a specialized programming language for RDTs with *automated* proof capabilities. VeriFx lets programmers implement RDTs atop functional collections and express correctness properties that are verified automatically. Verified RDTs can be transpiled to mainstream languages (currently Scala and JavaScript). VeriFx provides libraries for implementing and verifying Conflict-free Replicated Data Types (CRDTs) and Operational Transformation (OT) functions. These libraries implement the general execution model of those approaches and define their correctness properties. We use the libraries to implement and verify an extensive portfolio of 51 CRDTs, 16 of which are used in industrial databases, and reproduce a study on the correctness of OT functions.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Computing methodologies → Distributed programming languages; Theory of computation → Distributed algorithms

Keywords and phrases distributed systems, eventual consistency, replicated data types, verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.9

Related Version *Previous Version*: `https://arxiv.org/abs/2207.02502`

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.19`

Funding *Kevin De Porre*: Funded by the Research Foundation - Flanders. Project number 1S98519N. *Carla Ferreira*: Partly funded by EU Horizon Europe under Grant Agreement no. 101093006 (TaRDIS), and FCT-Portugal under grants UIDB/04516/2020 and PTDC/CCI-INF/32081/2017.

Acknowledgements The authors would like to thank Nuno Preguiça, Carlos Baquero, and Imine Abdessamad for their early feedback on this work.

1 Introduction

Replication is essential to modern distributed systems as it enables fast access times and improves the system's overall scalability, availability, and fault tolerance. When data is replicated across machines, replicas must be kept consistent to some extent. When facing network partitions, replicas cannot remain consistent while also accepting reads and writes, a consequence of the CAP theorem [17, 18, 39]. Programmers thus face a trade-off between

consistency and availability. Keeping replicas strongly consistent induces high latencies, poor scalability, and reduced availability since updates must be coordinated, e.g. using a consensus algorithm. By relaxing the consistency guarantees, latencies can be reduced and the overall availability improved, but users may observe temporary inconsistencies between replicas.

Distributed systems increasingly adopt weak consistency models. However, concurrent operations may lead to conflicts which must be solved in order to guarantee state convergence. Consider the case of collaborative text editors. When a user edits a document, the operation is immediately applied on the local replica and propagated asynchronously to other replicas. Since concurrent edits are applied in different orders at different replicas, states can diverge.

To ensure convergence, Ellis and Gibbs [25] proposed a technique called Operational Transformation (OT) that modifies incoming operations against previously executed concurrent operations such that the modified operation preserves the intended effect. Much work focused on designing OT functions for collaborative text editing [25, 34, 64, 69, 72], but most tombstone-free transformation functions (some with mechanized proofs) are wrong [34, 49, 62].

Since conflict resolution is hard [2, 41, 68], researchers now focus on designing replicated data types (RDTs) that serve as building blocks for the development of highly available distributed systems. Such RDTs resemble sequential data types (e.g. counters, sets) but include conflict resolution strategies that guarantee convergence.

Conflict-free Replicated Data Types (CRDTs) [68] are a widely adopted family of RDTs that leverage mathematical properties (such as commutative operations) to avoid conflicts by design. However, designing new RDTs is difficult [42] and even seasoned researchers miss subtle corner cases for basic data structures such as maps [40]. Currently, researchers and practitioners propose new or improved RDT designs [2, 8, 14, 19, 38, 41, 66–68] and include a formal specification or pseudo code of the RDT with a manual proof of convergence, mostly paper proofs. Unfortunately, paper proofs are subject to reasoning flaws.

To avoid the pitfalls of paper proofs, Zeller et al. [79], Gomes et al. [27], and Nieto et al. [59] propose formal frameworks to verify the correctness of CRDTs using proof assistants. However, these frameworks use abstract specifications that are disconnected from actual implementations (e.g. Akka's CRDT implementations in Scala). Hence, a particular implementation may be flawed, even if the specification was proven to be correct.

While interactive proofs are more convincing (as the proof logic is machine-checked), they require significant programmer intervention which is time-consuming and reserved to experts [45, 60]. Recent works try to automate (part of) the verification process of CRDTs. Nagar and Jagannathan [56] automatically verify CRDTs under different consistency models but require a first-order logic specification of the CRDT's operations that is cumbersome and error-prone. Liu et al. [54] extend Liquid Haskell [75] to verify CRDTs but significant parts need to be proven manually due to the way how Liquid Haskell encodes user-defined functions in SMT. For example, their Map CRDT required more than 1000 lines of proof code. We conclude that developing RDTs is reserved for experts in distribution and verification.

To simplify the development of RDTs, we propose VeriFx, a specialized functional object-oriented programming language for designing, implementing, and *automatically* verifying RDTs. The main challenge behind VeriFx's design consists in striking a good balance between expressiveness and automated verification. We designed VeriFx to support familiar, high-level language constructs that are suited to implement RDTs, without breaking automated verification. To *implement* RDTs, VeriFx provides extensive functional collections including tuples, sets, maps, vectors, and lists. These collections are immutable which is said to be desirable for the implementation of RDTs and their integration in distributed systems [30]. To *verify* RDT implementations, VeriFx features a novel proof construct that enables

programmers to express correctness properties. For each proof, VeriFx automatically derives proof obligations and discharges them using SMT solvers. This is possible because VeriFx efficiently encodes all functional collections and their operations using the Combinatory Array Logic [23] for SMT solvers, which is decidable. This enables VeriFx to automatically verify complex RDTs built atop these collections. VeriFx provides libraries that ease the implementation and verification of CRDT and OT data types. Internally, these libraries use the proof construct to define the necessary correctness properties. Verified RDTs can be transpiled to one of the supported target languages (currently Scala and JavaScript).

VeriFx is reminiscent of existing object-oriented languages (like Scala) and demonstrates that it is possible to automatically verify real-world RDT implementations without requiring separate specifications. This avoids mismatches between the implementation and the specification and simplifies software maintenance. We argue that the ability to implement RDTs and automatically verify them in the *same* language allows programmers to catch mistakes early during the development process.

To demonstrate the applicability of VeriFx, we implemented and *automatically* verified 51 CRDTs, including well-known CRDTs [2, 8, 14, 40, 66, 67] and new variants. 50 of these CRDTs were verified in a matter of seconds and one could not be verified due to its recursive nature. The CRDTs we verified feature highly optimized designs and many are used in industrial databases such as Riak [12], Cassandra [73], and AntidoteDB [1]. We also applied VeriFx to OT and verified *all* transformation functions described by Imine et al. [34] and some unpublished designs [33].

In summary, we make the following contributions:

- VeriFx, the first high-level programming language that enables programmers to implement RDTs by composing functional collections, express correctness properties about those RDTs within the same language, and automatically verify those properties. The novelty consists of carefully crafting the language such that every language construct is efficiently encoded without breaking automated verification.
- We devise VeriFx libraries that simplify the implementation of CRDT and OT data types and automatically verify the necessary correctness properties.
- We give the first fully automated and mechanized proofs for 51 CRDTs, including the state-based and operation-based CRDTs proposed by Shapiro et al. [67], delta state-based CRDTs proposed by Almeida et al. [2], pure op-based CRDTs proposed by Baquero et al. [8], and many more. To the best of our knowledge, this is the most extensive treatment of verified RDTs to date. Prior efforts [27, 54, 59, 79] verified only a few CRDTs due to the labour-intensive nature of the verification process.
- We reproduce the study of Imine et al. [34] regarding the verification of OT functions.

2 Motivation

To motivate the need for a language with automated proof capabilities, consider a distributed system in Scala with replicated data on top of Akka's highly-available distributed key-value store [52]. The store provides built-in CRDTs, e.g. sets, counters, etc. However, our system requires a Two-Phase Set (2PSet) CRDT [67] that is not provided by Akka. We thus need to implement it and verify our implementation.

For the implementation, we can take the specification from Shapiro et al. [67]. For the verification, we typically need a complete formalization of the implementation and its correctness properties which can then be proven manually using proof assistants. The resulting interactive proofs are complex and require considerable expertise. For example, Nieto et al. [59]'s implementation of a 2PSet in OCaml is only 25 LoC but its specification in Coq is 80 LoC and requires an additional 73 LoC to verify.

Alternatively, programmers could resort to Liu et al. [54]'s extension of Liquid Haskell that automates part of the verification process. However, non-trivial RDTs still require significant manual proof efforts: 200+ LoC for a replicated set and 1000+ LoC for a map [54]. Thus, we cannot reasonably assume that programmers have the time nor the skills to manually verify their implementation [45, 60].

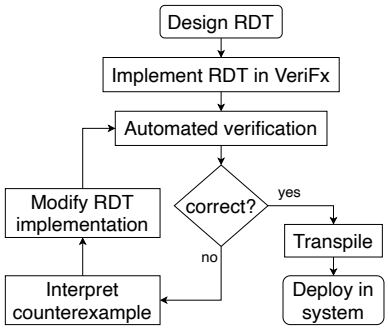


Figure 1 Envisioned workflow.

We argue that verification needs to be fully automatic to be accessible to non-experts. Figure 1 shows the workflow for developing RDTs using VeriFx, our novel language with a syntax reminiscent of Scala. Programmers start from a new or existing RDT design and implement it in VeriFx which verifies the implementation automatically without requiring a separate formalization. If the implementation is not correct, VeriFx returns a counterexample in which the replicas diverge. After interpreting the counterexample, the programmer needs to fix the RDT implementation and verify it again. This iterative process repeats until the implementation is correct. Verified RDT implementations can be transpiled to a mainstream language (e.g. Scala) and deployed in an actual system.

Our envisioned workflow verifies RDT implementations before deployment. Moreover, our workflow benefits from a feedback loop allowing programmers to correct implementations based on concrete counterexamples. In contrast, traditional verification techniques such as interactive theorem provers do not provide such feedback; when programmers fail to verify a property, they do not know if the implementation is flawed or if the chosen proof strategy is not suited. Similarly, Liquid Haskell [75] may fail to verify a property and raise a type error without providing additional information as to why the refinement type is not met. Next, we illustrate each step of our workflow by means of an existing 2PSet design in which VeriFx uncovered a bug. The corrected version was then transpiled to Scala, and deployed on Akka.

2.1 Design and Implementation

Specification 1 shows the design of the 2PSet CRDT taken from Shapiro et al. [67] unaltered. The 2PSet is a state-based CRDT whose state (the A and R sets) thus forms a join semilattice, i.e. a partial order $\leq_v$ with a least upper bound (LUB) $\sqcup_v$ for all states. Elements are added to the 2PSet by adding them to the A set and removed by *adding* them to the R set. An element is in the 2PSet if it is in A and not in R. Hence, removed elements can never be added again. Replicas are merged by computing the LUB of their states, which in this case is the union of their respective A and R sets.

The `compare(S,T)` operation checks if $S \leq_v T$ and is used to define state equivalence [68]: $S \equiv T \iff S \leq_v T \wedge T \leq_v S$. Since state equivalence is defined in terms of $\leq_v$ on the lattice, replicas may be considered equivalent even though they are not identical. This is relevant

Specification 1 2PSet CRDT taken from Shapiro et al. [67].

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

Listing 1 2PSet implementation in VeriFx.

```
class TwoPSet[V](added: Set[V], removed: Set[V])
    extends CvRDT[TwoPSet[V]] {
  def lookup(element: V) =
    this.added.contains(element) &&
    !this.removed.contains(element)
  def add(element: V) =
    new TwoPSet(this.added.add(element), this.removed)
  def remove(element: V) =
    new TwoPSet(this.added, this.removed.add(element))
  def compare(that: TwoPSet[V]) =
    this.added.subsetOf(that.added) ||
    this.removed.subsetOf(that.removed)
  def merge(that: TwoPSet[V]) =
    new TwoPSet(this.added.union(that.added),
                this.removed.union(that.removed))
}
```

for CRDTs that keep additional information. For example, CRDTs often use a Lamport clock [43] together with unique replica identifiers to generate globally unique IDs. Every replica identifier is different and is not part of the lattice even though it is part of the state.

Listing 1 shows the implementation of the 2PSet CRDT in VeriFx, which is a straightforward translation of Specification 1. The `TwoPSet` class is polymorphic in the type of values it stores. It defines the `added` and `removed` fields which correspond to the A and R sets respectively. The `add` and `remove` methods return an updated copy of the state. The class extends the `CvRDT` trait[1] provided by VeriFx's CRDT library (explained in Section 5.1). This trait requires the class to implement the `compare` and `merge` methods.

2.2 Verification

We now verify our 2PSet implementation in VeriFx. State-based CRDTs guarantee convergence if the merge function is idempotent, commutative, and associative [68]. VeriFx provides several `CvRDTProof` traits which encode these correctness conditions (explained later in Section 5.1). To verify the `TwoPSet`, we define a `TwoPSetProof` object that extends the `CvRDTProof1` trait (where `1` is the rank). The trait takes as argument the type constructor of the CRDT we want to verify (i.e. `TwoPSet`):

```
object TwoPSetProof extends CvRDTProof1[TwoPSet]
```

The `TwoPSetProof` object inherits automated correctness proofs for the polymorphic `TwoPSet` CRDT. When executing this object, VeriFx will automatically try to verify those proofs. In this case, VeriFx proves that the `TwoPSet` guarantees convergence (independent of the type of elements it holds), according to the notion of state equivalence that is derived from `compare`. However, VeriFx warns the user that the proof for state equivalence fails, which means that the derived notion of equivalence does not correspond to structural equality. As explained before, this may be normal in some CRDT designs but it requires further investigation.

VeriFx provides the following counterexample for the equivalence proof:

```
enum V { v }
val s: TwoPSet[V] = TwoPSet({v}, {})
val t: TwoPSet[V] = TwoPSet({v}, {v})
```

[1] VeriFx traits can declare abstract methods and fields, and provide default implementations for methods.

Listing 2 Transpiled 2PSet in Scala.

```
1  case class TwoPSet[V](added: Set[V], removed: Set[V])
       extends CvRDT[TwoPSet[V]] { // CvRDT trait provided
       by our CRDT library is also compiled to Scala
2    def lookup(element: V) = this.added.contains(element) &&
3                             !this.removed.contains(element)
4    def add(element: V): TwoPSet[V] =
5      TwoPSet[V](this.added + element, this.removed)
6    def remove(element: V): TwoPSet[V] =
7      TwoPSet[V](this.added, this.removed + element)
8    def compare(that: TwoPSet[V]): Boolean =
9      this.added.subsetOf(that.added) &&
10     this.removed.subsetOf(that.removed)
11   def merge(that: TwoPSet[V]): TwoPSet[V] =
12     TwoPSet[V](this.added.union(that.added),
13                this.removed.union(that.removed)) }
```

Listing 3 Modified 2PSet implementation for integration with Akka's distributed key-value store.

```
1  @SerialVersionUID(1L)
2  case class TwoPSet[V](
3    added: Set[V], removed: Set[V])
       extends CvRDT[TwoPSet[V]] with
       ReplicatedData with Serializable {
4    type T = TwoPSet[V]
5    // The remainder of the implementation
       is unchanged
6  }
```

The counterexample defines an enumeration `V` containing a single value `v`. It then defines two instances `s` and `t` of a `TwoPSet[V]` that are considered equivalent $s \equiv t$ (according to the definition of `compare`) but are not structurally equivalent $s \neq t$. These two instances should indeed not be considered equivalent since $v \in s$ but $v \notin t$ according to `lookup`. Looking back at Spec. 1, we notice that the original specification of `compare` from Shapiro et al. [67] defines replica `s` to be smaller or equal to replica `t` iff $s.A \subseteq t.A$ *or* $s.R \subseteq t.R$. Since $s.A = t.A$ it follows that $s \leq_v t \wedge t \leq_v s$ and thus they are considered equal ($s \equiv t$) without even considering the removed elements (i.e. the R sets). Based on this counterexample, we modify `compare` to consider both the A sets *and* the R sets:

```
def compare(that: TwoPSet[V]) =
  this.added.subsetOf(that.added) && this.removed.subsetOf(that.removed)
```

We verify the implementation again and VeriFx proves that this modified implementation still guarantees convergence and that the definition of equality that is derived from compare now corresponds to structural equality, i.e. $s \equiv t \iff s = t$.

This example showcases the importance of automated verification as it detected an error in the specification that would have percolated to the implementation. We completed the verification of the 2PSet CRDT in VeriFx without providing any verification-specific code.

2.3 Deployment

The final step in our workflow consists of automatically transpiling the CRDT implementation from VeriFx to Scala and integrating the CRDT in our distributed application which uses Akka's distributed key-value store. Listing 2 shows the transpiled implementation of the 2PSet in Scala. To store the RDT in Akka's distributed key-value store, we need to perform two manual modifications which are shown in Listing 3. First, the RDT must extend Akka's `ReplicatedData` trait (Line 3) which requires at least the definition of a type member T corresponding to the actual type of the CRDT (Line 4) and a `merge` method for CRDTs of that type (which we already have). Second, the RDT must be serializable. For simplicity, we use Java's built-in serializer[2]. Hence, it suffices to extend the `Serializable` trait (Line 3) and to annotate the class with a serial version (Line 1). After applying these modifications, our verified `TwoPSet` can be stored in Akka's distributed key-value store and will automatically be replicated across the cluster and be kept eventually consistent.

[2] In production it is safer and more efficient to implement a custom serializer [53], e.g. with Protobuf [28].

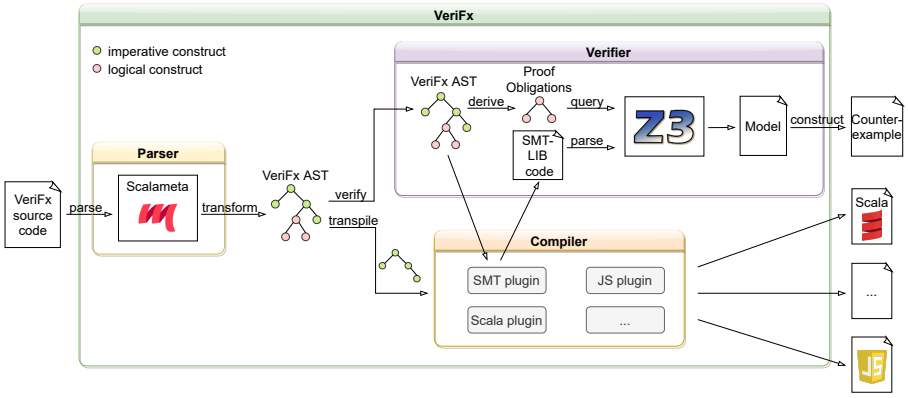


Figure 2 VeriFx's plugin architecture.

3 The VeriFx Language

The goal of this work is to build a familiar high-level programming language that is suited to implement RDTs and *automatically* verify them. The main challenge is to efficiently encode every feature of the language without breaking automatic verification. The result of this exercise is VeriFx, a functional object-oriented programming language with Scala-like syntax and a type system that resembles Featherweight Generic Java [32]. VeriFx features a novel proof construct to express correctness properties about programs. For every proof construct a proof obligation is derived that is discharged automatically by an SMT solver (cf. Section 4).

VeriFx advocates for the object-oriented programming paradigm as it is widespread across programmers and fits the conceptual representation of replicated data as "shared" objects. The functional aspect of VeriFx, in particular its immutable collections, makes it suitable for implementing and integrating RDTs in distributed systems, as argued by Helland [30].

The remainder of this section is organized in three parts. First, we give an overview of VeriFx's architecture. Second, we define its syntax. Third, we describe its functional collections. VeriFx's type system is described in Appendix A.

3.1 Overall Architecture

Figure 2 provides an overview of VeriFx's architecture. VeriFx programs consist of imperative code and proof code (i.e. logic statements). VeriFx uses Scala Meta [65] to parse VeriFx source code into an AST representing the program. This is possible because every piece of VeriFx code is valid Scala syntax (but not necessarily semantically correct).

The AST representing a VeriFx program can be verified or transpiled to other languages. Transpilation is done by the compiler which features *compiler plugins*. These plugins dictate the compilation of the AST to the target language. Currently, VeriFx comes with compiler plugins for Scala, JavaScript, and SMT-LIB [74], a standardized language for SMT solvers. Support for other languages can be added by implementing a compiler plugin for them.

To verify the proofs that are defined by a VeriFx program, the verifier derives the necessary proof obligations from the AST. VeriFx then compiles the program to SMT-LIB and automatically discharges the proof obligations using the Z3 SMT solver [22]. For every proof, the outcome is: *accepted*, *rejected*, or *unknown*. Accepted means that the property holds, rejected means that a counterexample was found for which the property does not hold, and unknown means that the property could not be verified within a certain time

$$
\begin{array}{rll}
L & ::= & \texttt{class}\ C\langle\overline{X}\rangle\,(\overline{v}:\overline{T})\,\{\,\overline{M}\,\} \\
 & \mid & \texttt{class}\ C\langle\overline{X}\rangle\,(\overline{v}:\overline{T})\ \texttt{extends}\ I\langle\overline{P}\rangle\{\,\overline{M}\,\} \\
J & ::= & \texttt{object}\ O\,\{\,\overline{A}\,\} \\
 & \mid & \texttt{object}\ O\ \texttt{extends}\ I\langle\overline{T}\rangle\,\{\,\overline{A}\,\} \\
F & ::= & \texttt{trait}\ I\langle\overline{X}<:\overline{T}\rangle\,\{\,\overline{B}\,\} \\
 & \mid & \texttt{trait}\ I\langle\overline{X}<:\overline{T}\rangle\ \texttt{extends}\ I\langle\overline{P}\rangle\,\{\,\overline{B}\,\} \\
N & ::= & \texttt{enum}\ E\langle\overline{X}\rangle\,\{\,K\,(\overline{v}:\overline{T})\,\} \\
A & ::= & M \mid R \\
B & ::= & \mathit{valD} \mid \mathit{methodD} \mid M \mid R \\
R & ::= & \texttt{proof}\ p\langle\overline{X}\rangle\,\{\,e\,\} \\
\mathit{valD} & ::= & \texttt{val}\ x:T \\
\mathit{methodD} & ::= & \texttt{def}\ m\langle\overline{X}\rangle(\overline{x}:\overline{T}):T \\
M & ::= & \texttt{def}\ m\langle\overline{X}\rangle\,(\overline{x}:\overline{T}):T=e \\
T & ::= & \texttt{int} \mid \texttt{string} \mid \texttt{bool} \mid C\langle\overline{T}\rangle \\
 & \mid & I\langle\overline{T}\rangle \mid E\langle\overline{T}\rangle \mid \ddot{T}\to T \\
e & ::= & \texttt{num} \mid \texttt{str} \mid \texttt{true} \mid \texttt{false} \mid x \mid\ !e \\
 & \mid & e\oplus e \mid e\otimes e \mid e.v \mid e.m\,\langle\overline{T}\rangle\,(\,\overline{e}\,) \\
 & \mid & \texttt{val}\ x:T=e\ \texttt{in}\ e \mid (\ddot{x}:\ddot{T})\Rightarrow e \mid e(\overline{e}) \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & \mid & \texttt{new}\ C\langle\overline{T}\rangle(\,\overline{e}\,) \mid \texttt{new}\ K\langle\overline{T}\rangle(\,\overline{e}\,) \\
 & \mid & e\ \texttt{match}\ \{\ddot{\texttt{case}}\ \ddot{r}\ddot{\Rightarrow}\ddot{e}\} \\
 & \mid & \texttt{forall}\,(\ddot{x}:\ddot{T})\texttt{.}\,e \mid \texttt{exists}\,(\ddot{x}:\ddot{T})\texttt{.}\,e \\
 & \mid & e\implies e \\
r & ::= & K(\overline{x}) \mid x \mid \_\_
\end{array}
$$

Figure 3 VeriFx syntax. The metavariable C ranges over class names; O ranges over object names; I ranges over trait names; E ranges over enumeration names; K ranges over constructor names of enumerations; T, P and Q range over types; X and Y range over type variables; v ranges over field names; x and y range over parameter and variable names; m ranges over method names; p ranges over proof names; and e ranges over expressions. An overline, e.g. $\overline{X}$, denotes zero or more. A dashed overline, e.g. $\ddot{X}$, denotes one or more.

frame (which is configurable). When a proof is rejected by Z3, VeriFx constructs a high-level counterexample that consists of concrete assignments of values to variables that violate the given property. Note that VeriFx can automatically verify application-specific properties because it derives the proof obligations from the program itself.

3.2 Syntax

Figure 3 defines the syntax of VeriFx. VeriFx programs consist of one or more statements which can be the definition of an object O, a class $C\langle\overline{X}\rangle$, a trait $I\langle\overline{X}\rangle$, or an enumeration $E\langle\overline{X}\rangle$. Objects, classes, enumerations, and traits can be polymorphic and inherit from a single trait (except enumerations). Objects define zero or more methods and proofs. Classes contain zero or more fields and (polymorphic) methods. Traits can declare values and methods that need to be provided by concrete classes extending the trait, and define (polymorphic) methods and proofs. Traits can express upper bounds on their type parameters to restrict possible extensions. Enumerations define one or more constructors, each containing zero or more fields. Programmers can deconstruct enumerations by pattern matching on them.

Unique to VeriFx is its proof construct which is defined by a name and a (well-typed) boolean expression that expresses the property that must be verified. A proof is accepted if its body always evaluates to true, otherwise it is rejected; when rejected, VeriFx provides a concrete counterexample for which the property does not hold. Proofs can be polymorphic, allowing properties to be proved for all possible type instantiations. Polymorphic proofs are useful to prove that a polymorphic RDT converges independent of its type of values.

VeriFx supports a variety of expressions, including literal values, arithmetic $\oplus$ and boolean operations $\otimes$, negation, field accesses, and method calls, variable definitions, if tests, anonymous functions and function calls, class and enum instantiations, pattern matching, quantified formulas, and logical implication. Functions are *first-class* and take at least one argument. Nullary functions can be expressed as constants.

Single inheritance is supported from traits to foster code re-use but some restrictions are imposed. E.g, the arguments of a class method need to be concrete (cannot be of a trait type) because proofs about these methods require reasoning about all subtypes but these may not necessarily be known at compile time. In contrast, enumerations are supported because their constructors are fixed and known at compile time.

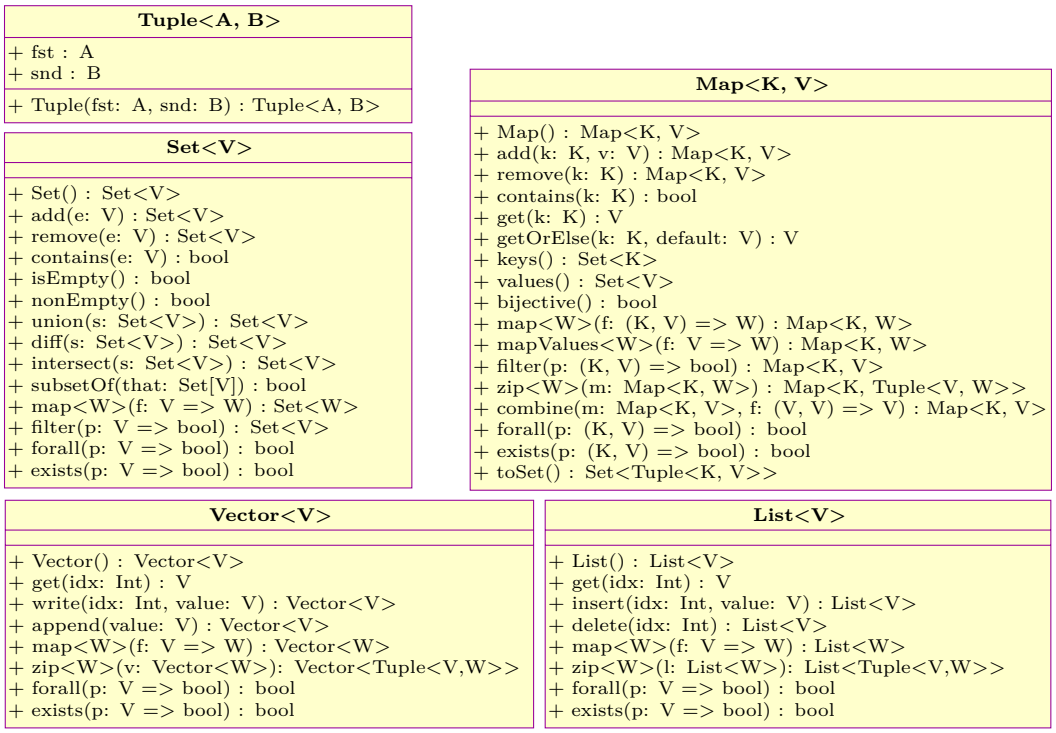


Figure 4 An overview of VeriFx's built-in functional collections.

3.3 Functional Collections

VeriFx features built-in collections for tuples, sets, maps, vectors, and lists. Every operation of these collections are verifiable and can be arbitrarily composed to build custom RDTs. All collections are immutable, "mutators" thus return an updated copy of the object. Figure 4 provides an overview of the interface exposed by these collections, which is heavily inspired by functional programming.

Sets. Support the typical set operations and can be mapped over or filtered using user-provided functions. The `forall` and `exists` methods check if a given predicate holds for all (respectively for at least one) element of the set.

Maps. Associate keys to values. Support adding key-value pairs, removing keys, and fetching the value associated with a key. The `keys` (resp. `values`) method returns a set containing all keys (resp. values) in the map. The `bijective` method checks if there is a one-to-one correspondence between keys and values. Maps support well-known functional operations; `zip` returns a map of tuples containing only the keys that are present in both maps and stores their values in a tuple; `combine` returns a map containing *all* entries from both maps, using a user-provided function `f` to combine values that are present in both maps.

Vectors. Represent a sequence of elements that are indexed from `0` to `size-1`. Elements can be written to a certain index which will overwrite the existing value at that index. One can append a value to the vector which will write that value at index `size`, thereby, making the vector grow. Like sets and maps, programmers can map functions over vectors, zip vectors, and check predicates for all or for one element of a vector.

Lists. Represent a sequence of elements in a linked list. Unlike vectors, `insert` does not overwrite the existing value at that index. Instead, the existing value at that index and all subsequent values are moved one position to the right. Elements can also be deleted from a list, making the list shrink.

$$
\begin{array}{llllll}
T & ::= \mathsf{int} \mid \mathsf{string} \mid \mathsf{bool} & G & ::= \mathsf{adt}\ A\langle \overline{X} \rangle \{ \overline{K(\overline{v} : \overline{T})} \} & C & ::= \mathsf{const}\ x\ T \quad R ::= \mathsf{assert}\ e \\
 & \mid \mathsf{Array}\langle \ddot{T}, T \rangle \mid A\langle \overline{T} \rangle \mid S\langle \overline{T} \rangle & e & ::= e[\ddot{e}] \mid e[\ddot{e}] := e \mid \lambda(\ddot{x} : \ddot{T}).e & D & ::= \mathsf{sort}\ S\ i \quad H ::= \mathsf{check}() \\
F & ::= \mathsf{fun}\ f\langle \overline{X} \rangle (\overline{x} : \overline{T}) : T = e & & \mid \forall(\ddot{x} : \ddot{T}).e \mid \exists(\ddot{x} : \ddot{T}).e \mid \ldots & &
\end{array}
$$

Figure 5 Core SMT syntax. The metavariable S ranges over user-declared sorts; A ranges over names of algebraic data types (ADTs); K ranges over ADT constructor names; X ranges over type variables; v ranges over field names; f ranges over function names; T ranges over types; x ranges over variable names; e ranges over expressions; and i ranges over integers.

4 Automated Verification

VeriFx leverages SMT solvers to enable automated verification. Such solvers try to (automatically) determine whether or not a given formula is satisfiable. Modern SMT solvers support various specialized theories (for bit vectors, arrays, etc.) and are very powerful if care is taken to encode programs efficiently using these theories. However, SMT-LIB [74], the language of SMT solvers, is low-level and is not meant to be used directly by programmers to verify high-level programs. Instead, semi-automatic program verification usually involves implementing the program in an Intermediate Verification Language (IVL) which internally compiles to SMT-LIB to discharge the proof obligations using an appropriate SMT solver. IVLs like Dafny [44], Spec# [11], and Why3 [26] are designed to be general-purpose but this breaks automated verification since programmers need to specify preconditions and postconditions on methods, loop invariants, etc.

VeriFx can be seen as a specialized high-level IVL that was carefully designed such that every feature has an efficient SMT encoding; leaving out features that break automated verification. For example, VeriFx does not support traditional loop statements but instead provides higher-order operations (map, filter, etc.) on top of its functional collections. The resulting language is surprisingly expressive given its automated verification capabilities.

The remainder of this section shows how VeriFx compiles programs to SMT and derives proof obligations that can be discharged automatically by SMT solvers. Afterward, we explain how VeriFx leverages a specialized theory of arrays to efficiently encode its functional collections. These encodings are key to our approach because they enable fully automated verification of RDTs built atop VeriFx's functional collections. VeriFx's encodings significantly differ from related work such as Why3 [26] and Liquid Haskell [75] which encode higher-order operations like map, and filter. recursively which hampers automated verification.

Appendix C.4 exemplifies VeriFx's compilation rules using a concrete example.

4.1 Core SMT

The semantics of VeriFx are defined using translation functions from VeriFx to Core SMT, a reduced version of SMT that suffices to verify VeriFx programs. Figure 5 defines the syntax of Core SMT. Valid types include integers, strings, booleans, arrays, ADTs $A\langle \overline{T} \rangle$, and user-declared sorts $S\langle \overline{T} \rangle$. Arrays are *total* and map values of the key types to a value of the element type. Arrays can be multidimensional and map several keys to a value.

Core SMT programs consist of one or more statements which can be the declaration of a constant or sort[3], assertions, the definition of a function or ADT, or a call to check. Constant declarations take a name and a type. Sort declarations take a name and a non-negative number i representing their arity, i.e. how many type parameters the sort takes.

[3] The literature on SMT solvers uses the term "sort" to refer to types and type constructors.

Declared constants and sorts are *uninterpreted* and the SMT solver is free to assign any valid interpretation. Assertions are boolean formulas that constrain the possible interpretations of the program, e.g. $\mathsf{assert\ age} >= 18$.

Function definitions consist of a name f, optional type parameters $\overline{X}$, formal parameters $\overline{x} : \overline{T}$, a return type T, and a body containing an expression e. Valid expressions include array accesses $e[\,\overline{e}\,]$, array updates $e[\,\overline{e}\,] := e$, anonymous functions, quantified formulas (the full list of expressions is shown in Appendix B). Updating an array returns a modified copy of the array. Note that arrays are total and that anonymous functions define an array. For example, $\lambda(x : \mathsf{int},\ y : \mathsf{int}).x + y$ defines an $\mathsf{Array}\langle\mathsf{int}, \mathsf{int}, \mathsf{int}\rangle$ that maps two integers to their sum. As arrays are first-class values in SMT, it follows that lambdas are also first-class.

ADT definitions consist of a name A, optional type parameters $\overline{X}$, and one or more constructors. A constructor has a name K and optionally defines fields with name v and type T. Constructors are called like regular functions and return an instance of the data type.

The decision procedure (check) checks the satisfiability of the SMT program. If the program's assertions are satisfiable, check returns a concrete model, i.e. an interpretation of the constants and sorts that satisfies the assertions. A property p can be proven by showing that the negation $\neg p$ is unsatisfiable, i.e. that no counterexample exists.

Note that our Core SMT language includes lambdas and polymorphic functions which are not part of SMT-LIB v2.6. Nevertheless, they are described in the preliminary proposal for SMT-LIB v3.0 [35] and Z3 already supports lambdas. For the time being, VeriFx monomorphizes polymorphic functions when they are compiled to Core SMT. For example, given a polymorphic identity function `id<X> :: X -> X`, VeriFx creates a monomorphic version `id_int :: int -> int` when encountering a call to `id` with an integer argument.

4.2 Compiling VeriFx to SMT

Similar to Dafny in [44], we describe the semantics of VeriFx by means of translation functions that compile VeriFx to Core SMT. Types are translated by the $[\![]\!]_t$ function:

$$
\begin{array}{llllllllll}
[\![\texttt{bool}]\!]_t & = & \mathsf{bool} & [\![\texttt{int}]\!]_t & = & \mathsf{int} & [\![\texttt{string}]\!]_t & = & \mathsf{string} \\
[\![C\langle\overline{\texttt{T}}\rangle]\!]_t & = & C\langle\overline{[\![\texttt{T}]\!]_t}\rangle & [\![E\langle\overline{\texttt{T}}\rangle]\!]_t & = & E\langle\overline{[\![\texttt{T}]\!]_t}\rangle & [\![\overrightarrow{T} \to P]\!]_t & = & \mathsf{Array}\langle\overrightarrow{[\![T]\!]_t}, [\![P]\!]_t\rangle
\end{array}
$$

Primitive types are translated to the corresponding primitive type in Core SMT. Class types and enumeration types keep the same type name and their type arguments are translated recursively $\overline{[\![\texttt{T}]\!]_t}$. Functions are encoded as arrays from the argument types to the return type. Trait types do not exist in the compiled SMT program because traits are compiled away by VeriFx, i.e. only the types of the classes that implement the trait exist in the SMT program.

We now take a look at the translation function $def[\![]\!]$ which compiles VeriFx's main constructs: enumerations, classes, and objects. Enumerations are encoded as ADTs:

$$
def[\![\texttt{enum}\ E\ \langle\overline{X}\rangle\ \{\ \overline{K\ (\overline{v} : \overline{T})}\ \}]\!] \ = \ \mathsf{adt}\ E\langle\overline{X}\rangle\{\overline{K(\overline{v} : \overline{[\![T]\!]_t})}\}
$$

For every enumeration an ADT is constructed with the same name, type parameters, and constructors. The types of the fields are translated recursively.

Classes are encoded as ADTs with one constructor and class methods become functions:

$$
\begin{array}{l}
def[\![\texttt{class}\ C\ \langle\overline{X}\rangle\ (\overline{v} : \overline{T})\ \{\ \overline{M}\ \}\ \texttt{extends}\ I\langle\overline{P}\rangle]\!] \ = \\
\quad \mathsf{adt}\ C\langle\overline{X}\rangle\{\ K(\overline{v} : \overline{[\![T]\!]_t})\ \}\ ;\ \overline{method[\![C, \overline{X}, M]\!]}\ ;\ \overline{method[\![C, \overline{X}, M'[\overline{P}/\overline{Y}]]\!]} \\
\quad \text{where } K = \texttt{str\_concat}(C, \text{"\_ctor"}) \text{ and } I \text{ is defined as } \texttt{trait}\ I\ \langle\overline{Y}\rangle\ \{\ \overline{M'}\ ;\ \ldots\ \} \\
method[\![C, \overline{X}, \texttt{def}\ m\ \langle\overline{Y}\rangle\ (\overline{x} : \overline{T}) : T_r = e]\!] \ = \mathsf{fun}\ f\langle\overline{X}, \overline{Y}\rangle(this : C\langle\overline{X}\rangle, \overline{x} : \overline{[\![T]\!]_t}) : [\![T_r]\!]_t = [\![e]\!] \\
\quad \text{where } f = \texttt{str\_concat}(C, \text{"\_"}, m)
\end{array}
$$

The ADT keeps the name of the class and its type parameters, and defines one constructor containing the class' fields. Since the name of the constructor must differ from the ADT's name, the compiler defines a unique name K which is the name of the class followed by "_ctor". Class methods $\overline{M}$ are compiled to regular functions by function $method[\![]\!]$. Further, a class inherits all concrete methods $\overline{M'}$ defined by its super trait that are not overridden. This entails substituting the trait's type parameters $\overline{Y}$ by the concrete type arguments $\overline{P}$ defined by the class. As such, traits are compiled away and do not exist in the transpiled SMT program.

For every method, a function is created with a unique name f that is the name of the class followed by an underscore and the name of the method. In the argument list, the body, and the return type of a method, programmers can refer to type parameters of the class and type parameters of the method. Therefore, the compiled SMT function takes both the class' type parameters $\overline{X}$ and the method's type parameters $\overline{Y}$. Without loss of generality we assume that a method's type parameters do not override the class' type parameters which can be achieved through α-conversion. The method's parameters become parameters of the function. In addition, the function takes an additional parameter *this* referring to the receiver of the method call which should be of the class' type. The types of the parameters and the return type are translated using function $[\![]\!]_t$. The body of the method must be a well-typed expression. Expressions are translated by the $[\![]\!]$ function:

$$
\begin{array}{lcl}
[\![x]\!] & = & x \\
[\![\texttt{val } x : T = e_1 \texttt{ in } e_2]\!] & = & \mathsf{let}\, x = [\![e_1]\!]\, \mathsf{in}\, [\![e_2]\!] \\
[\![(\overline{x} : \overline{T}) \Rightarrow e]\!] & = & \lambda(\overline{x} : \overline{[\![T]\!]_t}).[\![e]\!] \\
[\![e_1(\overline{e_2})]\!] & = & [\![e_1]\!][\ \overline{[\![e_2]\!]}\ ] \\
[\![\texttt{new } C\langle\overline{T}\rangle(\,\overline{e}\,)]\!] & = & C'\langle\overline{[\![T]\!]_t}\rangle(\overline{[\![e]\!]}) \\
\quad \text{where } C' = \texttt{str\_concat}(C, \text{"}\_ctor\text{"}) &&
\end{array}
\qquad
\begin{array}{lcl}
[\![\texttt{new } K\langle\overline{T}\rangle(\,\overline{e}\,)]\!] & = & K\langle\overline{[\![T]\!]_t}\rangle(\overline{[\![e]\!]}) \\
[\![e.v]\!] & = & [\![e]\!].v \\
[\![e_1.m\ \langle\overline{T}\rangle\,(\,\overline{e}\,)]\!] & = & m'\langle\overline{[\![P]\!]_t},\ \overline{[\![T]\!]_t}\rangle([\![e_1]\!],\ \overline{[\![e]\!]}) \\
\quad \text{where } \mathit{typeof}(e_1) = C\langle\overline{P}\rangle &&
\end{array}
$$

and $m' = \texttt{str\_concat}(C, \text{"}\_\text{"}, m)$
and $\overline{P} \cap \overline{T} = \emptyset$

Primitive values, variable references, and parameter references remain unchanged in Core SMT. The definition of an immutable variable is translated to a let expression. Anonymous functions remain anonymous functions in Core SMT, the type of the parameters and the body are compiled recursively. Remember that anonymous functions in SMT define (multidimensional) arrays from one or more arguments to the function's return value. Hence, function calls are translated to array accesses. To instantiate a class or ADT, the compiler calls the data type's constructor function. For classes, the constructor's name is the name of the class followed by "_ctor". To access a field, the compiler translates the expression and accesses the field on the translated expression. To invoke a method m on an object e_1 the compiler calls the corresponding function m' which by convention is the name of the class followed by an underscore and the name of the method. Recall that the function takes both the class' type arguments $\overline{T}$ and the method's type arguments $\overline{P}$ as well as an additional argument e_1 which is the receiver of the call. The complete set of compilation rules for expressions is provided in Appendix C.1 as part of the additional material.

Objects are singletons that can define methods and proofs, and are compiled as follows:

$$
\begin{array}{l}
\mathit{def}[\![\texttt{object } O \texttt{ extends } I\langle\overline{T}\rangle\ \{\ \overline{M}\ ;\ \overline{R}\ \}]\!] \ = \\
\quad \mathit{def}[\![\texttt{class } O'(\,)\ \{\overline{M}\}\ \texttt{extends } I\langle\overline{T}\rangle]\!]\ ;\ \mathsf{const}\ O\ O'\ ;\ \mathsf{assert}\ O == O'()\ ;\ \mathit{def}[\![\overline{R}]\!]
\end{array}
$$

The object is compiled to a regular class with a fresh name O'. Then, a single instance of that class is created and assigned to a constant named after the object O. The proofs defined by the object are compiled to functions. This translation is the subject of the next section.

4.3 Deriving Proof Obligations

We previously verified a 2PSet CRDT using VeriFx's CRDT library which internally uses our novel proof construct to define the necessary correctness properties (discussed in Section 5). However, programmers can also define custom proofs, for instance to verify data invariants.

We now explain how proof obligations are derived from user-defined proofs in VeriFx programs. Proofs are compiled to regular functions without arguments. While, the name and type parameters remain unchanged, the body of the proof is compiled and becomes the function's body. Proofs always return a boolean since the body is a logical formula whose satisfiability must be checked.

$$def[\![\texttt{proof}\ p\ \langle\overline{X}\rangle\ \{\ e\ \}]\!] \ = \ \mathsf{fun}\ p\langle\overline{X}\rangle() : \mathsf{bool} = [\![e]\!]$$

To check if the property described by a proof holds, the negation of the proof must be unsatisfiable – if no counterexample exists it constitutes a proof that the property is correct. A (polymorphic) proof called p with zero or more type parameters i is checked as follows:

$$prove(p, i) \quad = \quad \mathsf{sort}\ S_1\ 0\ ;\ \ldots\ ;\ \mathsf{sort}\ S_i\ 0\ \ ;\ \ \mathsf{assert}\ \neg p\langle S_1, \ldots, S_i\rangle()\ \ ;\ \ \mathsf{check}() == \mathsf{UNSAT}$$

For every type parameter, an *uninterpreted* sort is declared. Then, the proof function is called with those sorts as type arguments and we check that the negation is unsatisfiable. If the negation is unsatisfiable, the (polymorphic) proof holds for all possible instantiations of its type parameters. The underlying SMT solver can generate an actual proof which could be reconstructed by proof assistants as shown by Böhme et al. [15], Böhme and Weber [16].

4.4 Encoding Functional Collections Efficiently in SMT

Some IVLs feature collections with rich APIs (e.g. Why3 [26]) but encode operations on these collections recursively. Traditional SMT solvers fail to verify recursive definitions automatically because they require inductive proofs, which is beyond the capabilities of most solvers. However, many SMT solvers support specialized array theories. A key insight of this paper consists of efficiently encoding the collections and their operations using the Combinatory Array Logic (CAL) [23] which is decidable. As a result, VeriFx can automatically verify RDTs that are built by arbitrary compositions of functional collections. Next, we describe the encoding of sets using this array logic, while maps are described in Appendix C.3.

Set Encoding. Sets are encoded as arrays from the element type to a boolean type that indicates whether the element is in the set:

$$[\![\mathsf{Set}\ \langle T\rangle]\!]_t = \mathsf{Array}\langle [\![T]\!]_t, \mathsf{bool}\rangle$$

An empty set corresponds to an array containing false for every element. We can create such an array by defining a lambda that ignores its argument and always returns false:

$$[\![\texttt{new Set}\ \langle T\rangle(\ )]\!] = \lambda(x : [\![T]\!]_t).\mathsf{false}$$

Operations on sets are compiled as follows:

$$[\![e_1.add(e_2)]\!] = [\![e_1]\!][\,[\![e_2]\!]\,] := \mathsf{true} \qquad [\![e_1.remove(e_2)]\!] = [\![e_1]\!][\,[\![e_2]\!]\,] := \mathsf{false}$$

$$[\![e_1.filter(e_2)]\!] = \lambda(x : [\![T]\!]_t).[\![e_1]\!][\,x\,] \wedge [\![e_2]\!][\,x\,] \qquad [\![e_1.contains(e_2)]\!] = [\![e_1]\!][\,[\![e_2]\!]\,]$$

where $typeof(e_1) = \texttt{Set}\langle T\rangle \wedge typeof(e_2) = T \rightarrow \texttt{bool}$

$$[\![e_1.map(e_2)]\!] = \lambda(y : [\![P]\!]_t).\exists(x : [\![T]\!]_t).[\![e_1]\!][\,x\,] \wedge [\![e_2]\!][\,x\,] = y$$

where $typeof(e_1) = \texttt{Set}\langle T\rangle \wedge typeof(e_2) = T \rightarrow P$

An element e_2 is added to a set e_1 by setting the entry for e_2 in the array that results from transforming e_1 to true. Similarly, an element is removed by changing its entry in the array to false. An element is in the set if its entry is true. A set e_1 containing elements of type

T can be filtered such that only the elements that fulfil a given predicate $e_2 : T \to \texttt{bool}$ are retained. Calls to *filter* are compiled to a lambda that defines a set (i.e. an array from elements to booleans) containing only the elements x that are in the original set e_1 (i.e. $[\![e_1]\!][\ x\ ]$) and fulfil predicate e_2 (i.e. $[\![e_2]\!][\ x\ ]$). Similarly, a function $e_2 : T \to P$ can be mapped over a set e_1 of Ts, yielding a set of Ps. Calls to *map* are compiled to a lambda that defines a set containing elements y of type $[\![P]\!]_t$ such that an element x exists that is in the original set e_1 (i.e. $[\![e_1]\!][\ x\ ]$) and maps to y (i.e. $[\![e_2]\!][\ x\ ] = y$). The remaining methods are described in Appendix C.2 as part of the additional material.

5 Implementing and Verifying Replicated Data Types

VeriFx aims to simplify the development of correct RDTs by integrating automated verification capabilities in the language. Based on our experience implementing RDTs, we noticed that RDTs need to fulfill specific correctness properties that are well-defined for each RDT family. Therefore, VeriFx features built-in libraries for the development and automated verification of two well-known RDT families: CRDTs [68] and OT [25]. These libraries are written in VeriFx and define proofs that encode the necessary correctness properties such that programmers do not need to redefine these proofs for every RDT they implement.

This section discusses the aforementioned libraries. For each library, we formally define the correctness properties that must be verified for that specific RDT family. Section 5.1 describes the implementation of a general execution model for CRDTs and its verification library in VeriFx. Next, Section 5.2 presents a library for implementing RDTs using OT and verifying the transformation functions. VeriFx is not limited to these families of RDTs; programmers can build custom libraries for implementing and verifying other abstractions or families of RDTs. Last, Section 5.3 explains how to encode common assumptions such as causal delivery in VeriFx since the CRDT and OT libraries do not make specific assumptions.

5.1 CRDT Library

CRDTs guarantee *strong eventual consistency* (SEC), a consistency model that strengthens eventual consistency with the *strong convergence* property which requires replicas that received the same updates, possibly in a different order, to be in the same state. VeriFx's CRDT library supports all CRDT families: state-based [68], delta state-based [2], op-based [68], and pure op-based CRDTs [8]. The remainder explains how our library supports each family.

5.1.1 State-based CRDTs

State-based CRDTs (CvRDTs for short) periodically broadcast their state to all replicas and merge incoming states by computing the least upper bound (LUB) of the incoming state and their own state. Shapiro et al. [68] showed that CvRDTs converge if the merge function $\sqcup_v$ is idempotent, commutative, and associative. We define these properties based on their work:

Idempotent: $\forall x \in \Sigma : \mathit{reachable}(x) \implies x \equiv x \sqcup_v x$

Commutative: $\forall x, y \in \Sigma : \mathit{reachable}(x) \land \mathit{reachable}(y) \land \mathit{compatible}(x, y)$
$\implies (x \sqcup_v y \equiv y \sqcup_v x) \land \mathit{reachable}(x \sqcup_v y)$

Associative: $\forall x, y, z \in \Sigma : \mathit{reachable}(x) \land \mathit{reachable}(y) \land \mathit{reachable}(z) \land$
$\mathit{compatible}(x, y) \land \mathit{compatible}(x, z) \land \mathit{compatible}(y, z)$
$\implies ((x \sqcup_v y) \sqcup_v z \equiv x \sqcup_v (y \sqcup_v z)) \land \mathit{reachable}((x \sqcup_v y) \sqcup_v z)$

Listing 4 Trait for the implementation of CvRDTs in VeriFx.

```
trait CvRDT[T <: CvRDT[T]] {
  def merge(that: T): T
  def compare(that: T): Boolean
  def reachable(): Boolean = true
  def compatible(that: T): Boolean =
        true
  def equals(that: T): Boolean = {
    this.asInstanceOf[T].compare(that)
    &&
    that.compare(this.asInstanceOf[T])
  }
}
```

Listing 5 Trait for the verification of CvRDTs in VeriFx. The arrow function `=>:` implements logical implication.

```
trait CvRDTProof[T <: CvRDT[T]] {
  proof mergeIdempotent {
    forall (x: T) { x.reachable() =>: x.merge(x).equals(x) } }
  proof mergeCommutative {
    forall (x: T, y: T) {
      (x.reachable() && y.reachable() && x.compatible(y)) =>:
      (x.merge(y).equals(y.merge(x)) &&
           x.merge(y).reachable())}}
  proof mergeAssociative {
    forall (x: T, y: T, z: T) {
      (x.reachable() && y.reachable() && z.reachable() &&
       x.compatible(y) && x.compatible(z) && y.compatible(z))
        =>: (x.merge(y).merge(z).equals(x.merge(y.merge(z))) &&
             x.merge(y).merge(z).reachable()) } }
  proof equalityCheck {
    forall (x: T, y: T) { x.equals(y) == (x == y) } } }
```

Σ denotes the set of all states. A state is *reachable* if it can be reached starting from the initial state and applying only supported operations. Two states are *compatible* if they represent different replicas of the same CRDT object[4]. As explained in Section 2.1, Shapiro et al. [68] define state equivalence in terms of $\leq_v$ on the lattice: $S \leq_v T \wedge T \leq_v S \implies S \equiv T$.

VeriFx's CRDT library provides traits for the implementation and verification of CvRDTs, shown in Listings 4 and 5 respectively. Listing 4 shows the `CvRDT` trait that was used in Listing 1 to implement the `TwoPSet` CRDT. Every state-based CRDT that extends the `CvRDT` trait must provide a type argument which is the actual type of the CRDT and provide an implementation for the `merge` and `compare` methods. By default, all states are considered reachable and compatible, and state equivalence is defined in terms of `compare`. These methods can be overridden by the concrete CRDT that implements the trait.

Listing 5 shows the `CvRDTProof` trait used to verify CvRDT implementations. This trait defines one type parameter `T` that must be a CvRDT type and defines proofs to check that its merge function adheres to the aforementioned properties (i.e. is idempotent, commutative, and associative). It also defines an additional proof, `equalityCheck`, that checks that the notion of state equivalence that, by default, relies on structural equality (i.e., the default implementation of the `equals` method computes structural equality). However, programmers can override the `equals` method to use another notion of state equivalence if needed.

Objects can extend the `CvRDTProof` trait to inherit automated correctness proofs for the given CRDT type. Note that the trait's type parameter `T` expects a concrete CvRDT type (e.g. `PNCounter`) and will not work for polymorphic CvRDTs (e.g. `ORSet`) because those are type constructors. Instead, the CRDT library provides additional `CvRDTProof1`, `CvRDTProof2`, and `CvRDTProof3` traits to verify polymorphic CvRDTs that expect 1, 2, or 3 type arguments respectively. For example, the `TwoPSet[V]` from Section 2 is polymorphic in the type of values it stores; the `TwoPSetProof` object thus extended the `CvRDTProof1` trait because the `TwoPSet` expects one type argument.

5.1.2 Delta state-based CRDTs

Delta state-based CRDTs are a family of state-based CRDTs that exchange only the changes to the state (called deltas) instead of the full state in order to reduce the amount of data that is sent. Mutator operations return a delta which is joined into the replica's local state, propagated to the other replicas, and eventually joined into the state of all replicas.

[4] The `compatible` predicate can be used to encode certain assumptions. For example, replicas have unique identifiers which enables them to generate unique tags.

VeriFx's CRDT library provides a `DeltaCRDT` trait that specializes the `CvRDT` trait and can be used to implement delta state-based CRDTs. When extending `DeltaCRDT` traits, programmers must implement a `merge` method that joins delta states into the local state.

To verify delta state-based CRDTs, programmers can reuse the `CvRDTProof` trait since delta state-based CRDTs are essentially state-based CRDTs. As shown in Listing 5, the `CvRDTProof` trait verifies that the `merge` is idempotent, commutative, and associative **for all** valid states. This valid states contain all valid delta states as they are a subset of the full state.

5.1.3 Op-based CRDTs

Op-based CRDTs (CmRDTs for short) execute update operations in two phases, called *prepare* and *effect*. The prepare phase executes locally at the source replica (only if its source precondition holds) and prepares a message to be broadcast[5] to all replicas (including itself). The effect phase applies such incoming messages and updates the state (only if its downstream precondition holds, otherwise the message is ignored).

Shapiro et al. [68] and Gomes et al. [27] have shown that CmRDTs guarantee SEC if all concurrent operations commute. Hence, for any CmRDT it suffices to show that all pairs of concurrent operations commute. Formally, for any operation o_1 that is enabled by some reachable replica state s_1 (i.e. o_1's source precondition holds in s_1) and any operation o_2 that is enabled by some reachable replica state s_2, if these operations can be concurrent, and s_1, s_2, and s_3 are compatible replica states, then we must show that on any reachable replica state s_3 the operations commute and the intermediate and resulting states are reachable:

$$\begin{aligned}&\forall s_1, s_2, s_3 \in \Sigma, \forall o_1, o_2 \in \Sigma \to \Sigma : \mathit{reachable}(s_1) \land \mathit{reachable}(s_2) \land \mathit{reachable}(s_3) \land \\ &\quad \mathit{enabledSrc}(o_1, s_1) \land \mathit{enabledSrc}(o_2, s_2) \land \mathit{canConcur}(o_1, o_2) \land \\ &\quad \mathit{compatible}(s_1, s_2) \land \mathit{compatible}(s_1, s_3) \land \mathit{compatible}(s_2, s_3) \\ &\quad \implies o_2 \cdot o_1 \cdot s_3 \equiv o_1 \cdot o_2 \cdot s_3 \land \mathit{reachable}(o_1 \cdot s_3) \land \mathit{reachable}(o_2 \cdot s_3) \land \mathit{reachable}(o_1 \cdot o_2 \cdot s_3)\end{aligned}$$

We use the notation $o \cdot s$ to denote the application of an operation o on state s if its downstream precondition holds, otherwise, it returns the state unchanged.

Listing 6 shows the `CmRDT` trait that must be extended by op-based CRDTs with concrete type arguments for the supported operations, exchanged messages, and the CRDT type itself. A CRDT that extends the `CmRDT` trait must implement the `prepare` and `effect` methods. The `tryEffect` method has a default implementation that applies the operation if its downstream precondition holds, otherwise, it returns the state unchanged. By default, we assume all states are reachable, all operations are enabled at the source and downstream, all operations can occur concurrently, and all states are compatible. For most CmRDTs these settings do not need to be altered but some CmRDTs have other assumptions which can be encoded by overriding the appropriate method. For example, in an OR-Set [67] it is not possible to delete tags added concurrently; this can be encoded by overriding `canConcur`.

Similar to state-based CRDTs, our CRDT library provides a `CmRDTProof` trait and several versions to verify op-based CRDTs. These traits define a general proof of correctness that checks that all operations commute based on the previously described formula.

5.1.4 Pure op-based CRDTs

Pure op-based CRDTs are a family of op-based CRDTs that exchange only the operations instead of data-type specific messages. The effect phase stores incoming operations in a partially ordered log of (concurrent) operations. Queries are computed against the log and

[5] While some CmRDT do not require causal delivery, the overall model assumes reliable causal broadcast.

Listing 6 Polymorphic `CmRDT` trait to implement op-based CRDTs in VeriFx.

```
trait CmRDT[Op, Msg, T <: CmRDT[Op, Msg, T]] {
  def prepare(op: Op): Msg
  def effect(msg: Msg): T
  def tryEffect(msg: Msg): T = if (this.enabledDown(msg)) this.effect(msg) else this.asInstanceOf[T]
  def reachable(): Boolean = true // by default all states are considered reachable
  def canConcur(x: Msg, y: Msg): Boolean = true // all ops can occur concurrently
  def compatible(that: T): Boolean = true // all states are compatible
  def enabledSrc(op: Op): Boolean = true // no source preconditions by default
  def enabledDown(msg: Msg): Boolean = true // no downstream preconditions by default
  def equals(that: T): Boolean = this == that
}
```

operations do not need to commute. Data-type specific redundancy relations dictate which operations to store in the log and when to remove operations from the log. VeriFx's CRDT library provides a `PureOpBasedCRDT` trait which is a specialization of the `CmRDT` trait for implementing pure op-based CRDTs. The implementing CRDT inherits the prepare and effect phase (the same for all pure op-based CRDTs) and only needs to implement the redundancy relations. Since pure op-based CRDTs are essentially operation-based CRDTs, programmers can reuse the `CmRDTProof` traits to verify pure op-based CRDT implementations.

5.2 OT Library

The Operational Transformation (OT) [25] approach applies operations locally and propagates them asynchronously to the other replicas. Incoming operations are transformed against previously executed concurrent operations such that the modified operation preserves the intended effect. Operations are functions from state to state: $Op : \Sigma \to \Sigma$ and are transformed using a transformation function $T : Op \times Op \to Op$. Thus, $T(o_1, o_2)$ denotes the operation that results from transforming o_1 against a previously executed concurrent operation o_2. Suleiman et al. [70] and Sun et al. [72] proved that replicas eventually converge if the transformation function satisfies two properties: TP_1 and TP_2. Property TP_1 states that any two enabled concurrent operations o_i and o_j must commute after transforming them:

$$\forall o_i, o_j \in Op, \forall s \in \Sigma : enabled(o_i, s) \land enabled(o_j, s) \land canConcur(o_i, o_j) \implies T(o_j, o_i)(o_i(s)) = T(o_i, o_j)(o_j(s))$$

Property TP_2 states that given three enabled concurrent operations o_i, o_j, and o_k, the transformation of o_k does not depend on the order of the transformation of operations o_i and o_j:

$$\forall o_i, o_j, o_k \in Op, \forall s \in \Sigma : enabled(o_i, s) \land enabled(o_j, s) \land enabled(o_k, s) \land canConcur(o_i, o_j) \land canConcur(o_j, o_k) \land canConcur(o_i, o_k) \implies T(T(o_k, o_i), T(o_j, o_i)) = T(T(o_k, o_j), T(o_i, o_j))$$

Note that properties TP_1 and TP_2 only need to hold for states in which the operations can be generated, represented by the relation $enabled : Op \times \Sigma \to \mathbb{B}$, and only if the two operations can occur concurrently, represented by the relation $canConcur : Op \times Op \to \mathbb{B}$.

VeriFx provides a library for implementing and verifying RDTs that use operational transformations. Programmers can build custom RDTs by extending the `OT` trait shown in Listing 7. Every RDT that extends the `OT` trait must provide concrete type arguments for the state and operations, and implement the `transform` and `apply` methods. The `transform` method transforms an incoming operation against a previously executed concurrent operation. The `apply` method applies an operation on the state. By extending this trait, the RDT inherits proofs for TP_1 and TP_2. By default, these proofs assume that operations are always enabled and that all operations can occur concurrently. If this is not the case, the RDT can override the `enabled` and `canConcur` methods respectively.

Listing 7 Polymorphic `OT` trait to implement and verify RDTs using operational transformation.

```
1  trait OT[State, Op] {
2    def transform(x: Op, y: Op): Op
3    def apply(state: State, op: Op): State
4    def enabled(op: Op, state: State): Boolean = true
5    def canConcur(x: Op, y: Op): Boolean = true
6    proof TP1 {
7      forall (opI: Op, opJ: Op, st: State) {
8        (this.enabled(opI, st) && this.enabled(opJ, st) && this.canConcur(opI, opJ)) =>: {
9          this.apply(this.apply(st, opI), this.transform(opJ, opI)) ==
10         this.apply(this.apply(st, opJ), this.transform(opI, opJ)) } } }
11   proof TP2 {
12     forall (opI: Op, opJ: Op, opK: Op, st: State) {
13       (this.enabled(opI, st) && this.enabled(opJ, st) && this.enabled(opK, st) &&
14        this.canConcur(opI, opJ) && this.canConcur(opJ, opK) && this.canConcur(opI, opK)) =>: {
15         this.transform(this.transform(opK, opI), this.transform(opJ, opI)) ==
16         this.transform(this.transform(opK, opJ), this.transform(opI, opJ)) } } } }
```

Although VeriFx supports the general execution model of OT, most transformation functions described by the literature were specifically designed for collaborative text editing. They model text documents as a sequence of characters and operations insert or delete characters at a given position in the document. Every paper thus describes four transformations functions, one for every pair of operations: insert-insert, insert-delete, delete-insert, delete-delete.

Likewise, VeriFx's OT library provides a `ListOT` trait that models the state as a list of values and supports insertions and deletions. RDTs extending the `ListOT` trait need to implement four methods (`Tii`, `Tid`, `Tdi`, `Tdd`) corresponding to the transformation functions for transforming insertions against insertions (`Tii`), insertions against deletions (`Tid`), deletions against insertions (`Tdi`), and deletions against deletions (`Tdd`). The trait provides a default implementation of `transform` that dispatches to the corresponding transformation function based on the type of operations, and a default implementation of `apply` that inserts or deletes a value from the underlying list.

5.3 Encoding RDT-Specific Assumptions

Some RDTs (most notably op-based CRDTs) assume causal delivery of operations but VeriFx and its CRDT and OT libraries do not make any assumptions. In VeriFx, assumptions must either be guaranteed by the RDT's implementation or be explicitly encoded in the proofs.

We now show, using the OR-Set CRDT [67], how to encode RDT-specific assumptions. The OR-Set CRDT assumes that 1) replicas can generate globally unique tags, and 2) add and remove operations of the same element are delivered in causal order. These assumptions imply that replicas cannot add a tag and concurrently remove the same tag. The first assumption can be guaranteed by the RDT implementation if every replica has a unique ID that is combined with a local counter that increases monotonically to generate unique tags. The latter assumption about causal delivery can be explicitly encoded in the proof. One could also model the underlying causal communication protocols in VeriFx to remove this assumption.

Listing 8 shows an excerpt from the implementation of the OR-Set CRDT. It overrides the `compatible` predicate (Line 7) to encode the fact that replicas have unique IDs, and overrides the `canConcur` predicate (Line 8 to 16) such that the proof does not consider `add` and `remove` operations if the tag generated by `add` is contained in the set of tags that are removed (because causal delivery precludes `remove` from having observed that tag). This example shows how to encode specific assumptions but, in practice, many RDT implementations do not require any assumptions. Only 7 out of the 51 verified CRDTs (cf. Section 6.1) required assumptions, all of which are related to causal delivery and logical timestamps.

Listing 8 Excerpt from the implementation of the OR-Set CRDT [67].

```
class Tag[ID](replica: ID, counter: Int)
enum SetOp[V, ID] { Add(e: V) | Remove(e: V) }
enum SetMsg[V, ID] { AddMsg(e: V, tag: Tag[ID]) | RemoveMsg(e: V, tags: Set[Tag[ID]]) }
class ORSet[V, ID](id: ID, counter: Int, elements: Map[V, Set[Tag[ID]]])
    extends CmRDT[SetOp[V, ID], SetMsg[V, ID], ORSet[V, ID]] {
  // ...
  override def compatible(that: ORSet[V, ID]) = this.id != that.id
  override def canConcur(x: SetMsg[V, ID], y: SetMsg[V, ID]) = x match {
    case AddMsg(_, tag) => y match {
      case AddMsg(_, _) => true
      case RemoveMsg(_, tags) => !tags.contains(tag) // tag cannot be in tags because of causal delivery
    }
    case RemoveMsg(_, tags) => y match {
      case AddMsg(_, tag) => !tags.contains(tag) // tag cannot be in tags because of causal delivery
      case RemoveMsg(_, _) => true
} } }
```

6 Evaluation

We now evaluate the applicability of VeriFx to implement and verify RDTs. Our evaluation is twofold. First, we implement and verify numerous CRDTs taken from literature as well as some new variants[6]. Also, we verify well-known OT functions and some unpublished designs.

All experiments reported were conducted on AWS using an m5.xlarge VM with 4 virtual CPUs and 16 GiB of RAM. All benchmarks are implemented using JMH [61], a benchmarking library for the JVM. We configured JMH to execute 20 warmup iterations followed by 20 measurement iterations for every benchmark. To avoid run-to-run variance JMH repeats every benchmark in 3 fresh JVM forks, yielding a total of 60 samples per benchmark.

We do not conduct a performance evaluation for the transpiled RDT implementations as the transpilation merely changes the syntax to Scala or JavaScript but does not modify the RDT's design. Thus, the transpilation step does not affect the RDT's performance.

6.1 Verifying Conflict-free Replicated Data Types

We implemented and verified an extensive portfolio comprising 51 CRDTs, coming from literature [2, 8, 14, 40, 66, 67], open source projects [9], and industrial databases [1, 12, 52]. To the best of our knowledge, we are the first to mechanically verify all CRDTs from Shapiro et al. [67], all delta state-based CRDTs from Almeida et al. [2], all pure op-based CRDTs from Baquero et al. [8], and the map CRDT from Kleppmann [40].

Table 1 summarizes the verification results, including the average verification time and code size of each CRDT. When applicable, we mention which CRDTs are used in industrial databases. VeriFx was able to verify all implemented CRDTs except the Replicated Growable Array (RGA) [67] due to the recursive nature of the insertion algorithm (cf. Section 7). We found three issues: 1) the Two-Phase Set CRDT (described in Section 2) converges but is not functionally correct, 2) the original Map CRDT proposed by Kleppmann [40] diverges as VeriFx found the same counterexample as described in their technical report, and 3) the Molli, Weiss, Skaf (MWS) Set is incomplete. We now describe the implementation and verification of the Map CRDTs from [40], while Appendix E discusses the MWS Set.

[6] All implementations and proofs are provided as supplementary material in this submission.

Table 1 Verification results for CRDTs implemented and verified in VeriFx. S = state-based, D = delta state-based, O = op-based, P = pure op-based CRDT. ◷ = timeout, ⓐ = adaptation of an existing CRDT, ⓘ = incomplete definition. The database column includes databases that are known to use these CRDTs. Some delta state-based CRDTs use a dot kernel abstraction that is not counted in the LoC, this is indicated in the LoC column with an asterisk.

CRDT	Type	LoC	Correct	Time	Database	Source
Counter	O	17	✓	3.2 s	AntidoteDB	Shapiro et al. [67]
Grow-Only Counter	S	27	✓	4.3 s		Shapiro et al. [67]
Grow-Only Counter	D	27	✓	4.4 s	Akka	Almeida et al. [2]
Dynamic Grow-Only Counter	S	27	✓	4.4 s	Riak	ⓐ Shapiro et al. [67]
Positive-Negative Counter	S	12	✓	5.9 s		Shapiro et al. [67]
Positive-Negative Counter	D	17	✓	6.8 s	Akka	Almeida et al. [2]
Dynamic Positive-Negative Counter	S	17	✓	9.3 s	Riak	ⓐ Shapiro et al. [67]
Lex Counter	D	46	✓	4.7 s	Cassandra	Baquero et al. [9]
Causal Counter	D	28*	✓	6.7 s	Riak	Baquero et al. [9]
Enable-Wins Flag	P	18	✓	4.0 s		Baquero et al. [8]
Enable-Wins Flag	D	14*	✓	5.7 s	Riak	Baquero et al. [9]
Enable-Wins Flag	O	44	✓	3.6 s	AntidoteDB	AntidoteDB [4]
Disable-Wins Flag	P	20	✓	3.9 s		Baquero et al. [8]
Disable-Wins Flag	D	14*	✓	5.8 s	Riak	Baquero et al. [9]
Disable-Wins Flag	O	50	✓	3.8 s	AntidoteDB	AntidoteDB [3]
Multi-Value Register	S	63	✓	8.8 s		Shapiro et al. [67]
Multi-Value Register	D	12*	✓	7.1 s		Almeida et al. [2]
Multi-Value Register	P	18	✓	4.1 s		Baquero et al. [8]
Last-Writer-Wins Register	S	16	✓	5.3 s	Riak	Shapiro et al. [67]
Last-Writer-Wins Register	O	38	✓	4.4 s		Shapiro et al. [67]
Grow-Only Set	O	17	✓	3.9 s	AntidoteDB	Shapiro et al. [67]
Grow-Only Set	S	8	✓	5.3 s	Riak	Shapiro et al. [67]
Grow-Only Set	D	9	✓	3.9 s		Baquero et al. [9]
Two-Phase Set	O	27	✓	4.4 s		Shapiro et al. [67]
Two-Phase Set	S	16	✗	6.3 s		Shapiro et al. [67]
Two-Phase Set	D	25	✓	4.5 s		Baquero et al. [9]
Unique Set	O	39	✓	4.4 s		Shapiro et al. [67]
Add-Wins Set	P	28	✓	4.3 s		Baquero et al. [8]
Remove-Wins Set	P	42	✓	4.5 s		Baquero et al. [8]
Last-Writer-Wins Set	S	36	✓	6.6 s		Shapiro et al. [67]
Remove-Wins Last-Writer-Wins Set	D	28	✓	4.8 s		Baquero et al. [9]
Positive-Negative Set	S	36	✓	9.6 s		Shapiro et al. [67]
Observed-Removed Set	O	75	✓	6.2 s	AntidoteDB	Shapiro et al. [67]
Observed-Removed Set	S	34	✓	7.6 s		Shapiro [66]
Optimized OR Set	S	78	✓	30.2 s	Riak	Bieniusa et al. [14]
Add-Wins OR Set	D	28	✓	6.5 s		Almeida et al. [2]
Optimized Add-Wins OR-Set	D	16*	✓	7.3 s		Almeida et al. [2]
Optimized Remove-Wins OR-Set	D	27*	✓	8.5 s		Baquero et al. [9]
Molli, Weiss, Skaf (MWS) Set	O	45	✓	4.7 s		ⓘ Shapiro et al. [67]
Grow-Only Map	S	32	✓	9.1 s		new data type
Buggy Map	O	87	✗	65.2 s		Kleppmann [40]
Corrected Map	O	101	✓	49.4 s		Kleppmann [40]
2P2P Graph	O	58	✓	7.8 s		Shapiro et al. [67]
2P2P Graph	S	41	✓	10.7 s		ⓐ Shapiro et al. [67]
Add-Only Directed Acyclic Graph	O	42	✓	4.7 s		Shapiro et al. [67]
Add-Only Directed Acyclic Graph	S	30	✓	8.7 s		ⓐ Shapiro et al. [67]
Add-Remove Partial Order	O	61	✓	10.4 s		Shapiro et al. [67]
Add-Remove Partial Order	S	49	✓	13.2 s		ⓐ Shapiro et al. [67]
Replicated Growable Array	O	156	◷	/		Shapiro et al. [67]
Continuous Sequence	O	108	✓	9.2 s		ⓐ Shapiro et al. [67]
Continuous Sequence	S	53	✓	11.4 s		ⓐ Shapiro et al. [67]

6.1.1 Map CRDTs

Kleppmann [40] describes the implementation of a Map CRDT which he believed to be "obviously correct" only to find out it contains a bug that causes divergence after spending hours trying to verify it. He then tweeted the buggy pseudo code of the Map CRDT and challenged his 29400 followers (mainly software engineers) to find the bug. Only one person managed to manually identify the bug and one other person came close (at the time, both were Ph.D. students specialized in RDTs). Kleppmann later tweeted a variation on the algorithm: "Here is a variant of the algorithm that is correct (I believe)".

We used VeriFx to implement and *automatically* verify both the buggy Map CRDT and the corrected Map CRDT, which had not been formally verified. The full implementation and verification of the buggy map CRDT is explained in Appendix D. We now present the key takeaways from our experience implementing and verifying these map CRDTs.

Implementation. The implementation of the map CRDTs mainly consisted of translating the mathematical specifications to VeriFx. We introduced slight changes to the design to improve efficiency. For example, the specification keeps a set of triples where each triple holds a key, a value, and a timestamp. Since every key appears at most in one triple, our implementation uses a dictionary to efficiently map keys to their value and timestamp.

Verification. After implementing the buggy map CRDT, we proceeded to its automated verification but VeriFx generated invalid counterexamples. For instance, one in which two distinct replicas generated the same timestamp. This is not possible because the design assumes that replicas have unique IDs and combines them with Lamport clocks [43] to generate unique timestamps. However, VeriFx does not know this assumption nor does it know the relation between a replica's clock and the values it observed. In practice, other CRDTs make similar *implicit* assumptions which is the reason they are complex and difficult to get right. VeriFx helped us to explicitly encode all assumptions as it kept returning invalid counterexamples which helped us find and formulate the missing assumptions. Listing 11 in Appendix D.3 shows the encoding of these assumptions.

Counterexample. After explicitly defining all assumptions, VeriFx found a valid counterexample for the buggy map CRDT that is equivalent to the one found manually by Nair [40]. It consists of a corner case in which the `Put` and `Delete` operations do not commute and thus may cause replicas to diverge. We detail the counterexample in Appendix D.3.

Corrected Map CRDT. After finding the counterexample for the buggy map CRDT, we also verified the corrected map CRDT from Kleppmann [40]. This did not require additional efforts since we already distilled all assumptions for the buggy map CRDT. VeriFx automatically proved that the corrected design indeed guarantees convergence, which to the best of our knowledge, is the first mechanical proof of correctness for this CRDT.

As shown in Table 1, the verification times for the buggy and corrected map CRDTs are slightly higher compared to the other CRDTs we verified, but are still very fast for a fully automated verification approach. The higher times come from the fact that these CRDTs are too complex to directly prove convergence of all operation pairs. Hence, we use a subproof for every operation pair. The total verification time is the sum of the times of the subproofs.

6.1.2 Conclusion

Based on Table 1, we conclude that VeriFx is suited to verify CRDT implementations since all were verified mechanically and fully automatically in a matter of seconds. To the best of our knowledge, this is the most extensive portfolio of verified RDTs to date. It is representative of real-world use cases as it includes several CRDTs used in industrial databases.

Table 2 Verification results of OT functions in VeriFx.

Transformation Function	LoC	Props		Time	
		TP_1	TP_2	TP_1	TP_2
Ellis and Gibbs [25]	84	✗	✗	115 s	29 s
Ressel et al. [64]	78	✓	✗	68 s	30 s
Sun et al. [72]	68	✗	✗	321 s	13 s
Suleiman et al. [69]	85	✗	✗	34 s	40 s
Imine et al. [34]	83	✓	✗	61 s	17 s
$\text{Register}_{\text{V1}}$ [33]	6	✗	✓	3 s	3 s
$\text{Register}_{\text{V2}}$ [33]	6	✓	✗	3 s	3 s
$\text{Register}_{\text{V3}}$ [33]	7	✓	✓	3 s	3 s
Stack [33]	47	✗	✓	5 s	5 s

Listing 9 Excerpt from the implementation of Imine et al. [34]'s functions.

```
1  enum Op { Ins(p: Int, ip: Int, c: Int) |
       Del(p: Int) | Id() }
2  object Imine extends ListOT[Int, Op] {
3    def Tii(x: Ins, y: Ins) = {
4      val p1 = x.p; val ip1 = x.ip; val c1 = x.c
5      val p2 = y.p; val ip2 = y.ip; val c2 = y.c
6      if (p1 < p2) x
7      else if (p1 > p2) new Ins(p1 + 1, ip1, c1)
8      else if (ip1 < ip2) x
9      else if (ip1 > ip2) new Ins(p1+1, ip1, c1)
10     else if (c1 < c2) x
11     else if (c1 > c2) new Ins(p1+1, ip1, c1)
12     else new Id() }
13   def Tid(x: Ins, y: Del) =
14     if (x.p > y.p) new Ins(x.p - 1, x.ip, x.c)
15     else x
16   def Tdi(x: Del, y: Ins) =
17     if (x.p < y.p) x else new Del(x.p + 1)
18   def Tdd(x: Del, y: Del) = if (x.p < y.p) x
19     else if (x.p > y.p) new Del(x.p - 1)
20     else new Id() }
```

Overall, the main challenge to building such an extensive portfolio consisted of finding and encoding the correct assumptions. Those assumptions were usually gradually discovered as VeriFx returned counterexamples that cannot occur in practice, which indicates that one or more assumptions are missing. In particular, counterexamples were crucial for verifying the Map CRDTs from [40]. These are the designs that took the longest to implement and verify and they required an afternoon of work.

6.2 Verifying Operational Transformation

We now show that VeriFx is general enough to verify other distributed abstractions such as Operational Transformation (OT). We implemented all transformation functions for collaborative text editing defined by Imine et al. [34] and verified TP_1 and TP_2 in VeriFx.

Table 2 summarizes the verification results. For each transformation function, the table shows the code size, whether or not it satisfies TP_1 and TP_2, and the average verification time. As shown in the table, the functions proposed by Ellis and Gibbs [25], Sun et al. [72], and Suleiman et al. [69] do not satisfy TP_1 nor TP_2. Ressel et al. [64]'s functions satisfy TP_1 but not TP_2. These results confirm prior findings by Imine et al. [34]. VeriFx also found that the functions proposed by Imine et al. [34] do not satisfy TP_2, which confirms the findings of Li and Li [49] and Oster et al. [62]. That same counterexample also invalidates the transformation functions of Suleiman for TP_2. Imine et al. [34] wrongly proved Suleiman's functions [69] correct, but VeriFx found counterexamples for both properties (the counterexample for TP_1 was manually found in [63]). We believe that the specification defined in Imine et al. [34] may have missed those counterexamples due to a wrong encoding of assumptions. Finally, in a private communication, Imine [33] asked us to verify (unpublished) OT designs for replicated registers and stacks. Out of the three register designs verified in VeriFx, only one is correct for both TP_1 and TP_2. Regarding the stack design, it guarantees TP_2 but not TP_1. VeriFx provided meaningful counterexamples for each incorrect design.

To exemplify our approach to verifying OT, we now describe the implementation and verification of Imine et al. [34]'s transformation functions in VeriFx, which are shown in Listing 9. The enumeration `Op` on Line 1 defines the three supported operations:

- `Ins(p, ip, c)` represents the insertion of character `c`[7] at position `p`. Initially, `c` was inserted at position `ip`. Transformations may change `p` but leave `ip` untouched.
- `Del(p)` represents the deletion of the character at position `p`.
- `Id()` acts as a no-op (to which operations may be transformed).

Object `Imine` extends the `ListOT` trait and implements the transformation functions (`Tii`, `Tid`, `Tdi`, `Tdd`) required for collaborative text editing (cf. Section 5.2). The implementation of these transformation functions is a straightforward translation from their description by Imine et al. [34]. The resulting object inherits automated proofs for TP_1 and TP_2. When running these proofs, VeriFx reports that the transformation functions guarantee TP_1 but not TP_2.

Based on the results shown in Table 2, we conclude that VeriFx is suited to verify other RDT families such as OT. Due to the number of cases that have to be considered, the verification times are longer than for CRDTs but are still acceptable for static verification [21].

7 Discussion

We now discuss the main design decisions behind VeriFx, its limitations and trade-offs.

Traits. For simplicity, VeriFx only supports single inheritance from traits. However, it could be extended to support multiple inheritance. Traits are not meant for subtyping because subtyping complicates verification as every subtype needs to be verified but these might not be known at compile time. Hence, class fields, method parameters, etc. cannot be of a trait type. Programmers can, however, define enumerations as these have a fixed number of constructors, which are known at compile time. Note that traits can define type parameters with upper type bounds. The type checker uses these bounds to ensure that every extending class or trait is well-typed. The compiled SMT program does not contain traits as they are compiled away (cf. Section 4.2). Proofs, classes, and methods cannot have bounds on type parameters because the compiler does not know all subtypes.

Functional collections. VeriFx encodes higher-order operations on collections (e.g. `map`, `filter`) using arrays, which are treated as function spaces in the Combinatory Array Logic (CAL) [23]. Hence, anonymous functions (lambdas) merely define arrays that are first-class. SMT solvers can efficiently reason about VeriFx's functional collections because CAL is decidable. However, some operations are encoded using universal or existential quantifiers which may hamper decidability. In practice, VeriFx can verify RDTs involving complex functional operations. Unfortunately, VeriFx's collections do not yet provide aggregation methods (e.g. `fold` and `reduce`) because this is beyond the capabilities of CAL. These restrictions may soon be lifted as SMT-LIB v3 [35] preliminary plans incorporate new theories that include aggregation functions such as fold.

Trade-off between expressiveness and verifiability. All constructs in VeriFx were carefully designed to have efficient SMT encoding. Overall, general loop constructs cannot be verified automatically as those require inductive proofs. A key insight of VeriFx is that *implicit* loop constructs (e.g. `map`, etc.) enable the automatic verification of an extensive portfolio of RDTs. Even though the language does not provide general loop constructs, programmers can define recursive methods. While VeriFx will not prove facts about (unbounded) recursive methods out-of-the-box, programmers can still verify these implementations by explicitly defining inductive proofs. This requires devising a suitable induction hypothesis and defining two proofs: one for the base case, and another for the induction step. Then, VeriFx can verify both proofs. This approach has been used to verify nested CRDT designs [13].

[7] We represent characters using integers that correspond to their ASCII code.

8 Related Work

We focus our comparison of related work on verification languages and approaches for verifying RDTs, invariants in distributed systems, and operational transformation.

Verification languages. Verification languages can be classified into three categories: interactive, auto-active, and automated [45]. Interactive languages include proof assistants like Coq and Isabelle/HOL in which programmers define theorems and prove them manually using proof tactics. Although automation tactics exist, proving complex theorems requires considerable manual proof efforts. Vazou et al. [76] introduce the idea of refinement reflection in Liquid Haskell [75], where user-defined functions are reflected in a decidable fragment of SMT logic and can be used in refinement types to express correctness properties. Similarly, in VeriFx *every* construct of the language and its collections are reflected in SMT logic such that arbitrary VeriFx programs can be reflected in the logic. However, VeriFx enables *automated* verification of user-defined correctness properties whereas, Liquid Haskell requires programmers to express correctness properties using refinement types and *manually* write proofs as Haskell functions. Moreover, VeriFx offers an iterative process where incorrect designs are improved based on the counterexamples, whereas Liquid Haskell only raises a type error. Auto-active verification languages like Dafny [44] and Spec# [11] verify programs based on annotations provided by the programmer (e.g. preconditions, postconditions, loop invariants). Intermediate verification languages (IVLs) like Boogie [10] and Why3 [26] automate the proof task by generating verification conditions (VCs) from source code and discharging them using one or more SMT solvers. IVLs are not meant to be used by programmers directly. Instead, programs written in some verification language (e.g. Dafny, Spec#) are translated to an IVL to verify the VCs. Regarding automated verification, the work by Kaki and Jagannathan [37] integrated an automated verification framework in a refinement type system. Programmers write relational specifications that define structural relations for the RDT at hand and express correctness properties as refinement types atop operations. However, writing relational specifications for advanced data types is non-trivial and can be rather verbose, as noted by the authors themselves. In contrast, VeriFx does not require separate specifications.

Verifying SEC for RDTs. Burckhardt et al. [20] propose a formal framework that enables the specification and verification of RDTs. Attiya et al. [5] use a variation on this framework to provide specifications of replicated lists and prove the correctness of an existing text editing protocol. Gomes et al. [27] and Zeller et al. [79] propose formal frameworks in the Isabelle/HOL theorem prover to mechanically verify SEC for CRDTs. In contrast to VeriFx, Gomes et al. [27] only consider operation-based CRDTs but model the underlying network to reason about causal delivery of messages. Nieto et al. [59] developed libraries to implement and verify op-based CRDTs in separation logic. Their approach requires programmers to write Coq specifications atop the provided libraries and manually prove correctness. Liu et al. [54] extend Liquid Haskell with typeclass refinements and use them to prove SEC for some of their own CRDTs. While simple proofs can be discharged automatically by the underlying SMT solver, advanced CRDTs also require significant proof efforts (as discussed in Section 2). All the aforementioned verification techniques require significant effort and expertise whereas, VeriFx fully automatically verified 51 well-known CRDTs. Liang and Feng [51] propose a new correctness criterion for CRDTs that extends SEC with functional correctness and enables manual verification of CRDT implementations and client programs using them. They mainly focus on functional correctness and provide paper proofs rather than automated verification. Wang et al. [77]

propose replication-aware linearizability, a criterion that enables sequential reasoning to prove the correctness of CRDT implementations. The CRDTs were manually encoded in the Boogie verification tool to prove correctness. Those encodings are non-trivial and differ from real-world CRDT implementations. Nagar and Jagannathan [56] developed a proof rule that is parametrized by the consistency model and automatically checks convergence for CRDTs. Unfortunately, their framework introduces imprecision and may reject correct CRDTs. Moreover, their framework requires a first-order logic specification of the CRDT. In contrast, VeriFx can verify high-level CRDT implementations instead of specifications. Finally, Jagadeesan and Riely [36] introduce a notion of validity for RDTs and manually prove it for some CRDTs. We do not consider validity in this work.

Verifying applications invariants. Some work has focused on verifying application invariants under weak consistency. Bailis et al. [6] introduce invariant confluent operations that maintain application invariants, even without coordination. Whittaker and Hellerstein [78] devise a decision procedure for invariant confluence that can be checked automatically. Other work has focused on verifying invariants for RDTs [7, 29, 57, 58, 80]. Soteria [57] verifies program invariants for state-based RDTs. Repliss [80] verifies program invariants for applications that are built on top of their CRDT library. CISE [29, 58] proposes a proof rule to check that a chosen consistency level for operations preserves the application invariants. IPA [7] detects invariant-breaking operations and proposes changes to the operations in order to preserve the invariants. All these approaches assume that the underlying RDT is correct, while VeriFx enables programmers to verify that this is the case. This paper does not consider RDTs with mixed consistency levels as [24, 31, 46–48, 50, 55, 81, 82].

Verifying operational transformation functions. Ellis and Gibbs [25] first proposed an algorithm for OT together with a set of transformation functions. Several works [70,72] showed that integration algorithms like adOPTed [64], SOCT2 [70], and GOTO [71] guarantee convergence iff the transformation functions satisfy the TP_1 and TP_2 properties. Ellis and Gibbs [25]'s functions do not satisfy these properties [64, 70, 72] and, over the years, several functions were proposed [64, 69, 72]. Imine et al. [34] used SPIKE, an automated theorem prover, to verify the correctness of these functions and found counterexamples for all of them, except for Suleiman et al. [69]'s functions. As shown in Section 6.2, we reproduced their study and generated similar counterexamples. Imine et al. [34] proposed a simpler set of functions which later was found to also violate TP_2 [49, 62]. VeriFx also found this counterexample.

9 Conclusion

To support the development of correct RDTs, we propose VeriFx, a high-level programming language powerful enough to implement RDTs as CRDTs and OT, and verify them automatically without requiring annotations or programmer intervention. Our approach enables programmers to implement RDTs, and express and verify correctness properties, all within the *same* language. This avoids gaps between the implementation and verification. VeriFx high-level counterexamples enable programmers to iteratively improve their implementation. Once verified, RDTs can be transpiled to mainstream languages, e.g. Scala and JavaScript.

VeriFx shows that automated verification based on SMT solving can verify real-world RDT implementations with minimal programmer intervention. This work accounts for the most extensive portfolio of mechanically verified RDTs to date, including 51 CRDTs and 9 OT designs. All were verified in a matter of seconds or minutes and with minimal effort.

In this work, we focused on verifying correctness properties in the domain of RDTs. In future work, we would like to explore the applicability of VeriFx to other domains.

References

1 Deepthi Devaki Akkoorath and Annette Bieniusa. Antidote: The highly-available geo-replicated database with strongest guarantees. Technical report, Technical Report. Tech. U. Kaiserslautern., 2016.

2 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based CRDTs by delta-mutation. In Ahmed Bouajjani and Hugues Fauconnier, editors, *Int. Conference on Networked Systems*, pages 62–76, Agadir, Morocco, 2015. Springer-Verslag.

3 AntidoteDB. Implementation of a Disable-Wins Flag CRDT in AntidoteDB. `https://github.com/AntidoteDB/antidote/blob/master/apps/antidote_crdt/src/antidote_crdt_flag_dw.erl`. Accessed: 2022-07-19.

4 AntidoteDB. Implementation of an Enable-Wins Flag CRDT in AntidoteDB. `https://github.com/AntidoteDB/antidote/blob/master/apps/antidote_crdt/src/antidote_crdt_flag_ew.erl`. Accessed: 2022-07-19.

5 Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 259–268, New York, NY, USA, 2016. ACM. `doi:10.1145/2933057.2933090`.

6 Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014. `doi:10.14778/2735508.2735509`.

7 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. IPA: Invariant-preserving applications for weakly consistent replicated databases. *Proc. VLDB Endow.*, 12(4):404–418, December 2018.

8 Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. *CoRR*, abs/1710.04469, 2017. `arXiv:1710.04469`.

9 Carlos Baquero, Omer Katz, Brian Cannard, and Georges Younes. JGraphT: a Java library of graph theory data structures and algorithms. `https://github.com/CBaquero/delta-enabled-crdts`. Accessed: 22-11-2022.

10 Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

11 Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

12 Basho Technologies. Riak KV. `https://riak.com/products/riak-kv/index.html`. Accessed: 22-11-2022.

13 Jim Bauwens and Elisa Gonzalez Boix. Nested pure operation-based CRDTs. In *To Appear in 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, WA*, LIPIcs, 2023.

14 Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012. `arXiv:1210.3368`.

15 Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, pages 183–198, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

16 Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In *International Conference on Interactive Theorem Proving*, pages 179–194. Springer, 2010.

17 Eric Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45:23–29, February 2012.

18 Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 7, New York, NY, USA, 2000. ACM. doi:10.1145/343477.343502.

19 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In *26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag.

20 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535848.

21 Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing.

22 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

23 Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *2009 Formal Methods in Computer-Aided Design*, pages 45–52, 2009. doi:10.1109/FMCAD.2009.5351142.

24 Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. ECROs: Building Global Scale Systems from Sequential Code. *Proc. ACM Program. Lang.*, 5(OOPSLA), November 2021.

25 C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM. doi:10.1145/67544.66963.

26 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

27 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133933.

28 Google. Protocol buffers. https://developers.google.com/protocol-buffers. Accessed: 10-10-2022.

29 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. *SIGPLAN Not.*, 51(1):371–384, January 2016. doi:10.1145/2914770.2837625.

30 Pat Helland. Immutability changes everything: We need it, we can afford it, and the time is now. *Queue*, 13(9):101–125, November 2015. doi:10.1145/2857274.2884038.

31 Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication coordination analysis and synthesis. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290387.

32 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. doi:10.1145/503502.503505.

33 Abdessamad Imine. Exchange of mails regarding OT, and unpublished register and stack designs. personal communication.

34 Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the Eighth Conference on European Conference on Computer Supported Cooperative Work*, ECSCW'03, pages 277–293, USA, 2003. Kluwer Academic Publishers.

35 The SMT-LIB Initiative. SMT-LIB Version 3.0 - Preliminary Proposal. http://smtlib.cs.uiowa.edu/version3.shtml. Accessed: 23-11-2022.

36 Radha Jagadeesan and James Riely. Eventual consistency for CRDTs. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 968–995, Cham, 2018. Springer International Publishing.

37 Gowtham Kaki and Suresh Jagannathan. A relational framework for higher-order shape analysis. *SIGPLAN Not.*, 49(9):311–324, August 2014. doi:10.1145/2692915.2628159.

38 Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360580.

39 Martin Kleppmann. A critique of the CAP theorem. *CoRR*, abs/1509.05393, 2015. arXiv:1509.05393.

40 Martin Kleppmann. Assessing the understandability of a distributed algorithm by tweeting buggy pseudocode. Technical Report UCAM-CL-TR-969, University of Cambridge, Computer Laboratory, May 2022. doi:10.48456/tr-969.

41 Martin Kleppmann and Alastair R Beresford. A conflict-free replicated JSON datatype. *IEEE Trans. on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.

42 Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: Synthesizing CRDTs with verified lifting. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563336.

43 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

44 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

45 K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In *Usable Verification Workshop*, 2010.

46 Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

47 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, USA, 2012. USENIX Association.

48 Cheng Li, Nuno Preguiça, and Rodrigo Rodrigues. Fine-grained consistency for geo-replicated systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 359–372, Boston, MA, 2018. USENIX Association.

49 Du Li and Rui Li. Preserving operation effects relation in group editors. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, CSCW '04, pages 457–466, New York, NY, USA, 2004. ACM. doi:10.1145/1031607.1031683.

50 Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hampa: Solver-aided recency-aware replication. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*, pages 324–349, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-53288-8_16.

51 Hongjin Liang and Xinyu Feng. Abstraction for conflict-free replicated data types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 636–650, New York, NY, USA, 2021. ACM. doi:10.1145/3453483.3454067.

52 Lightbend, Inc. Akka. https://akka.io/. Accessed: 22-11-2022.

53 Lightbend Inc. Serialization. https://doc.akka.io/docs/akka/current/serialization.html. Accessed: 10-10-2022.

54 Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. Verifying replicated data types with typeclass refinements in liquid Haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428284.

55 Mae Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 226–241, New York, NY, USA, 2018. ACM. doi:10.1145/3192366.3192375.

56 Kartik Nagar and Suresh Jagannathan. Automated parameterized verification of CRDTs. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 459–477, Cham, 2019. Springer International Publishing.

57 Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. Proving the safety of highly-available distributed objects. In *Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*, pages 544–571, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-44914-8_20.

58 Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE tool: proving weakly-consistent applications correct. In Peter Alvaro and Alysson Bessani, editors, *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*, pages 2:1–2:3. ACM, 2016. doi:10.1145/2911151.2911160.

59 Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. Modular verification of op-based CRDTs in separation logic. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563351.

60 Peter W. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 13–25, New York, NY, USA, 2018. ACM. doi:10.1145/3209108.3209109.

61 OpenJDK. jmh - OpenJDK. https://openjdk.java.net/projects/code-tools/jmh/. Accessed: 13-05-2020.

62 Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 1–10, 2006. doi:10.1109/COLCOM.2006.361867.

63 Aurel Randolph, Hanifa Boucheneb, Abdessamad Imine, and Alejandro Quintero. On synthesizing a consistent operational transformation approach. *IEEE Transactions on Computers*, 64(4):1074–1089, 2015. doi:10.1109/TC.2014.2308203.

64 Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 288–297, New York, NY, USA, 1996. ACM. doi:10.1145/240080.240305.

65 Scalameta. Scalameta: Library to read, analyze, transform and generate Scala programs. https://scalameta.org/. Accessed: 24-11-2022.

66 Marc Shapiro. Replicated Data Types. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia Of Database Systems*, volume Replicated Data Types, pages 1–5. Springer-Verlag, July 2017. doi:10.1007/978-1-4899-7993-3_80813-1.

67 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report RR-7506, INRIA – Centre Paris-Rocquencourt, January 2011.

68 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Grenoble, France, 2011. Springer-Verslag.

69 Maher Suleiman, Michèle Cart, and Jean Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge*, GROUP '97, pages 435–445, New York, NY, USA, 1997. ACM. doi:10.1145/266838.267369.

70 Maher Suleiman, Michèle Cart, and Jean Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, pages 36–45, USA, 1998. IEEE Computer Society.

71 Chengzheng Sun and Clarence Ellis. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Proc. of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68, 1998.

72 Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998. doi:10.1145/274444.274447.

73 The Apache Software Foundation. Cassandra: Open source NoSQL database. https://cassandra.apache.org/_/index.html. Accessed: 24-11-2022.

74 The SMT-LIB Initiative. SMT-LIB. https://smtlib.cs.uiowa.edu. Accessed: 24-11-2022.

75 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, 2014. ACM. doi:10.1145/2628136.2628161.

76 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: Complete verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158141.

77 Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. Replication-aware linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 980–993, New York, NY, USA, 2019. ACM. doi:10.1145/3314221.3314617.

78 Michael J. Whittaker and Joseph M. Hellerstein. Interactive checks for coordination avoidance. *Proc. VLDB Endow.*, 12(1):14–27, 2018. doi:10.14778/3275536.3275538.

79 Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of CRDTs. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 33–48, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

80 Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Combining state- and event-based semantics to verify highly available programs. In Farhad Arbab and Sung-Shik Jongmans, editors, *Formal Aspects of Component Software*, pages 213–232, Cham, 2020. Springer International Publishing.

81 Xin Zhao and Philipp Haller. Observable atomic consistency for CvRDTs. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 23–32, 2018.

82 Xin Zhao and Philipp Haller. Replicated data types that unify eventual consistency and observable atomic consistency. *Journal of Logical and Algebraic Methods in Programming*, 114:100561, 2020.

A VeriFx's Type System

We now present VeriFx's type system. An environment Γ is a partial and finite mapping from variables to types. A type environment Δ is a finite set of type variables. VeriFx's type system consists of a judgment for type wellformdness $\Delta \vdash T$ ok which says that type T is well-formed in context Δ, and a judgment for typing $\Delta; \Gamma \vdash e : T$ which says that in context Δ and environment Γ, the expression e is of type T. We abbreviate $\Delta \vdash T_1$ ok, $\ldots$, $\Delta \vdash T_n$ ok to $\Delta \vdash \overline{T}$ ok, and $\Delta; \Gamma \vdash e_1 : T_1, \ldots, \Delta; \Gamma \vdash e_n : T_n$ to $\Delta; \Gamma \vdash \overline{e} : \overline{T}$.

Below we define well-formed types:

$$\frac{}{\Delta \vdash \texttt{string}\ \text{ok}}\ (\textit{WF-String}) \qquad \frac{}{\Delta \vdash \texttt{bool}\ \text{ok}}\ (\textit{WF-Bool}) \qquad \frac{}{\Delta \vdash \texttt{int}\ \text{ok}}\ (\textit{WF-Int})$$

$$\frac{\begin{array}{c}\Delta \vdash \overline{T}\ \text{ok} \\ \texttt{class}\ C\,\langle \overline{X} \rangle\,(\ldots)\,\{\ldots\} \\ \textit{or}\ \ \texttt{class}\ C\,\langle \overline{X} \rangle\,(\ldots)\ \texttt{extends}\ I\langle \ldots \rangle\{\ldots\}\end{array}}{\Delta \vdash C\langle \overline{T} \rangle\ \text{ok}}\ (\textit{WF-Class})$$

$$\frac{\begin{array}{c}\Delta \vdash \overline{T}\ \text{ok} \qquad \overline{T} <: \overline{P} \\ \texttt{trait}\ I\,\langle \overline{X <: P} \rangle\,\{\ldots\} \\ \textit{or}\ \ \texttt{trait}\ I\,\langle \overline{X <: P} \rangle\ \texttt{extends}\ I\langle \ldots \rangle\,\{\ldots\}\end{array}}{\Delta \vdash I\langle \overline{T} \rangle\ \text{ok}}\ (\textit{WF-Trait})$$

$$\frac{\begin{array}{c}\Delta \vdash \overline{T}\ \text{ok} \\ \texttt{enum}\ E\,\langle \overline{X} \rangle\,\{\ldots\}\end{array}}{\Delta \vdash E\langle \overline{T} \rangle\ \text{ok}}\ (\textit{WF-Enum}) \qquad \frac{X \in \Delta}{\Delta \vdash X\ \text{ok}}\ (\textit{WF-TVar})$$

Primitive types are always well-formed. A type variable X is valid if it is in scope: $X \in \Delta$, i.e. the surrounding method or class defined the type parameter. Class types and enumeration types are valid if a corresponding class or enumeration definition exists and all type arguments are well-formed.

We now define a few auxiliary definitions which are needed for the typing rules. The *fields* function takes a class type and returns its fields and their types:

$$\frac{\texttt{class}\ C\,\langle \overline{X} \rangle\,(\overline{v} : \overline{T})\,\{\,\overline{M}\,\}\ \ \textit{or}\ \ \texttt{class}\ C\,\langle \overline{X} \rangle\,(\overline{v} : \overline{T})\ \texttt{extends}\ I\langle \overline{Q} \rangle\{\,\overline{M}\,\}}{\mathit{fields}(C\langle \overline{P} \rangle) = [\overline{P}/\overline{X}]\ \overline{v} : \overline{T}}\ (\text{F-class})$$

The *ftypes* function takes an enumeration type and the name of one of its constructors and returns the type of the fields of that constructor.

$$\frac{\texttt{enum}\ E\,\langle \overline{X} \rangle\,\{\,K(\overline{v} : \overline{T}), \ldots\,\}}{\mathit{ftypes}(E\langle \overline{P} \rangle, K) = [\overline{P}/\overline{X}]\ \overline{T}}\ (\text{FT-enum})$$

The *mtype* function takes the name of a method and the type of a class, and returns the actual type signature of the method. If the method is not found in the class (MT-class-rec rule) it is looked up in the hierarchy of super traits by the MT-trait rules. For polymorphic methods, the returned type signature is polymorphic:

$$\frac{\begin{array}{c}\texttt{class}\ C\,\langle \overline{X} \rangle\,(\ldots)\,\{\,\overline{M}\,\}\ \ \textit{or}\ \ \texttt{class}\ C\,\langle \overline{X} \rangle\,(\ldots)\ \texttt{extends}\ I\langle \overline{Q} \rangle\{\,\overline{M}\,\} \\ \texttt{def}\ m\,\langle \overline{Y} \rangle\,(\overline{x} : \overline{T}) : T = e \in \overline{M}\end{array}}{\mathit{mtype}(m, C\langle \overline{P} \rangle) = [\overline{P}/\overline{X}]\ (\langle \overline{Y} \rangle \overline{T} \to T)}\ (\text{MT-class})$$

$$\frac{\begin{array}{c}\texttt{class}\ C\,\langle \overline{X} \rangle\,(\ldots)\ \texttt{extends}\ I\langle \overline{Q} \rangle\{\,\overline{M}\,\} \\ \texttt{def}\ m\,\langle \overline{Y} \rangle\,(\overline{x} : \overline{T}) : T = e \notin \overline{M}\end{array}}{\mathit{mtype}(m, C\langle \overline{P} \rangle) = \mathit{mtype}(m, I\langle \overline{Q} \rangle)}\ (\text{MT-class-rec})$$

$$\frac{\begin{array}{c}\texttt{trait}\ I\,\langle \overline{X <: T'} \rangle\,\{\,\overline{M}\,\}\ \ \textit{or}\ \ \texttt{trait}\ I\,\langle \overline{X <: T'} \rangle\ \texttt{extends}\ I'\langle \ldots \rangle\,\{\,\overline{M}\,\} \\ \texttt{def}\ m\,\langle \overline{Y} \rangle\,(\overline{x} : \overline{T}) : T = e \in \overline{M}\end{array}}{\mathit{mtype}(m, I\langle \overline{P} \rangle) = [\overline{P}/\overline{X}]\ (\langle \overline{Y} \rangle \overline{T} \to T)}\ (\text{MT-trait})$$

$$\frac{\begin{array}{c}\texttt{trait}\ I\,\langle \overline{X <: T'} \rangle\,\{\,\overline{M}\,\}\ \ \textit{or}\ \ \texttt{trait}\ I\,\langle \overline{X <: T'} \rangle\ \texttt{extends}\ I'\langle \overline{P} \rangle\,\{\,\overline{M}\,\} \\ \texttt{def}\ m\,\langle \overline{Y} \rangle\,(\overline{x} : \overline{T}) : T = e \notin \overline{M}\end{array}}{\mathit{mtype}(m, I\langle \overline{P} \rangle) = \mathit{mtype}(m, I'\langle \overline{P} \rangle)}\ (\text{MT-trait-rec})$$

Similarly, we assume that there are functions $valNames(I\langle\overline{P}\rangle)$ and $declaredMethods(I\langle\overline{P}\rangle)$ that return all fields, respectively all methods, declared by a trait (and its super traits). The *ctors* function takes an enumeration type and returns the names of its constructors.

$$\frac{\texttt{enum}\ E\ \langle\overline{X}\rangle\ \{\ \overline{K\ (\overline{x} : \overline{T})}\ \}}{ctors(E\langle\overline{P}\rangle) = \overline{K}}\ \text{(C-ENUM)}$$

Figure 6 shows the typing rules for expressions. Most rules are a simplification of Featherweight Generic Java [32] without subtyping. Quantified formulas are boolean expressions if their body also types to a boolean expression in the environment that is extended with the quantified variables (T-UNI and T-EXI rules). The logical implication is a well-typed boolean expression if both the antecedent and the consequent are boolean expressions (T-IMPL rule).

Classes are well-formed if the types of the fields are well-formed and all its methods are well-formed (T-CLASS1 rule). If the class extends a trait, it must also implement all fields and methods declared by the hierarchy of super traits (T-CLASS2 rule). The typing rules for trait definitions and object definitions can be defined similarly.

When instantiating an enumeration through one of its constructors $\texttt{new}\ K\langle\overline{P}\rangle(\ \overline{e}\ )$, the provided arguments $\overline{e}$ need to match the types of the constructors' fields, and the result effectively is an object of the enumeration type $E\langle\overline{P}\rangle$.

Programmers can pattern match on enumerations but the cases must be exhaustive, i.e. every constructor must be matched by at least one case. If all cases are of type T, then the resulting pattern match expression is also of type T.

Finally, the body of a proof must be a well-typed boolean expression.

B Core SMT Expressions

We will now discuss the expressions that are supported by Core SMT. Those expressions are common to most SMT solvers, except lambdas which, as mentioned before, are described by the preliminary proposal for SMT-LIB v3.0 and are only implemented by some SMT solvers such as Z3 [22].

Figure 7 provides an overview of all Core SMT expressions. The simplest expressions are literal values representing integers, strings, and booleans. Core SMT supports the typical arithmetic operators ($+$, $-$, $*$, $/$) and boolean operators ($\wedge$, $\vee$, and negation $\neg$) as well as universal and existential quantification, and logical implication. Immutable variables are defined by let bindings. Pattern matching is supported but the cases must be exhaustive. For example, when pattern matching against an algebraic data type every constructor must be handled. Core SMT supports two types of patterns: constructor patterns $n(\overline{n})$ that match a specific ADT constructor n and binds names to its fields $\overline{n}$, and wildcard patterns that match anything and give it a name n. References v refer to variables that are in scope, e.g. function parameters or variables introduced by let binding or pattern matching. If statements are supported but an else branch is mandatory and both branches must evaluate to the same sort. Functions can be called and type arguments can be provided explicitly to disambiguate polymorphic functions. For example, we defined an ADT $\mathsf{Option}\langle T\rangle$ with two constructors Some and None. When calling the None constructor we need to explicitly provide a type argument since it cannot be inferred from the call, e.g. $\mathsf{None}\langle\mathsf{int}\rangle()$. Finally, fields of an ADT can be accessed by their name. Arrays and lambdas were already discussed in Section 4.1.

$$\frac{}{\Delta;\Gamma \vdash \texttt{num} : \texttt{int}}\ (\text{T-NUM}) \qquad \frac{}{\Delta;\Gamma \vdash \texttt{str} : \texttt{string}}\ (\text{T-STR}) \qquad \frac{}{\Delta;\Gamma \vdash \texttt{true} : \texttt{bool}}\ (\text{T-TRUE})$$

$$\frac{}{\Delta;\Gamma \vdash \texttt{false} : \texttt{bool}}\ (\text{T-FALSE}) \qquad \frac{x \in dom(\Gamma)}{\Delta;\Gamma \vdash x : \Gamma(x)}\ (\text{T-VAR}) \qquad \frac{\Delta;\Gamma \vdash e : \texttt{bool}}{\Delta;\Gamma \vdash !e : \texttt{bool}}\ (\text{T-NEG})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \texttt{int} \quad \Delta;\Gamma \vdash e_2 : \texttt{int}}{\Delta;\Gamma \vdash e_1 \oplus e_2 : \texttt{int}}\ (\text{T-OP1}) \qquad \frac{\Delta;\Gamma \vdash e_1 : \texttt{bool} \quad \Delta;\Gamma \vdash e_2 : \texttt{bool}}{\Delta;\Gamma \vdash e_1 \otimes e_2 : \texttt{bool}}\ (\text{T-OP2})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \texttt{bool} \quad \Delta;\Gamma \vdash e_2 : T \quad \Delta;\Gamma \vdash e_3 : T}{\Delta;\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T}\ (\text{T-IF}) \qquad \frac{\Delta \vdash T_1\ \text{ok} \quad \Delta;\Gamma \vdash e_1 : T_1 \quad \Delta;\Gamma, x : T_1 \vdash e_2 : T_2}{\Delta;\Gamma \vdash \texttt{val}\ x : T_1 = e_1\ \texttt{in}\ e_2 : T_2}\ (\text{T-VAL})$$

$$\frac{\Delta \vdash \ddot{T}\ \text{ok} \quad \Delta;\Gamma, \ddot{x} : \ddot{T} \vdash e : T}{\Delta;\Gamma \vdash (\ddot{x} : \ddot{T}) \Rightarrow e : \ddot{T} \to T}\ (\text{T-ABS}) \qquad \frac{\Delta;\Gamma \vdash e_1 : \ddot{T} \to T \quad \Delta;\Gamma \vdash \ddot{e_2} : \ddot{T}}{\Delta;\Gamma \vdash e_1(\ddot{e_2}) : T}\ (\text{T-CALL})$$

$$\frac{fields(C\langle\overline{P}\rangle) = \overline{v} : \overline{T} \quad \Delta \vdash C\langle\overline{P}\rangle\ \text{ok} \quad \Delta;\Gamma \vdash \overline{e} : \overline{T}}{\Delta;\Gamma \vdash \texttt{new}\ C\langle\overline{P}\rangle(\overline{e}) : C\langle\overline{P}\rangle}\ (\text{T-N-CLASS}) \qquad \frac{\Delta;\Gamma \vdash e : T_o \quad fields(T_o) = \overline{v} : \overline{T}}{\Delta;\Gamma \vdash e.v_i : T_i}\ (\text{T-FIELD})$$

$$\frac{\Delta;\Gamma \vdash e_o : T_o \quad \Delta \vdash \overline{P}\ \text{ok} \quad mtype(m, T_o) = \langle\overline{X}\rangle\overline{T} \to T \quad \Delta;\Gamma \vdash \overline{e} : [\overline{P}/\overline{X}]\overline{T}}{\Delta;\Gamma \vdash e_o.m\ \langle\overline{P}\rangle\ (\overline{e}) : [\overline{P}/\overline{X}]T}\ (\text{T-INV}) \qquad \frac{\Delta \vdash \ddot{T}\ \text{ok} \quad \Delta;\Gamma, \ddot{x} : \ddot{T} \vdash e : \texttt{bool}}{\Delta;\Gamma \vdash \texttt{forall}\ (\ddot{x} : \ddot{T})\ \texttt{.}\ e : \texttt{bool}}\ (\text{T-UNI})$$

$$\frac{\Delta \vdash \ddot{T}\ \text{ok} \quad \Delta;\Gamma, \ddot{x} : \ddot{T} \vdash e : \texttt{bool}}{\Delta;\Gamma \vdash \texttt{exists}\ (\ddot{x} : \ddot{T})\ \texttt{.}\ e : \texttt{bool}}\ (\text{T-EXI}) \qquad \frac{\Delta;\Gamma \vdash e_1 : \texttt{bool} \quad \Delta;\Gamma \vdash e_2 : \texttt{bool}}{\Delta;\Gamma \vdash e_1 \implies e_2 : \texttt{bool}}\ (\text{T-IMPL})$$

$$\frac{ctors(E\langle\overline{P}\rangle) = \ddot{K} \quad K \in \ddot{K} \quad ftypes(E\langle\overline{P}\rangle, K) = \overline{T} \quad \Delta \vdash E\langle\overline{P}\rangle\ \text{ok} \quad \Delta;\Gamma \vdash \overline{e} : \overline{T}}{\Delta;\Gamma \vdash \texttt{new}\ K\langle\overline{P}\rangle(\overline{e}) : E\langle\overline{P}\rangle}\ (\text{T-N-ENUM})$$

$$\frac{\Delta;\Gamma \vdash e_0 : E\langle\overline{P}\rangle \quad (ctors(E\langle\overline{P}\rangle) \setminus \dot{c} = \emptyset) \vee (\texttt{case}\ x \Rightarrow e \in \dot{c}) \vee (\texttt{case}\ \_ \Rightarrow e \in \dot{c}) \quad for\ each\ c \in \dot{c} : \Delta;\Gamma \vdash c : T\ IN\ e_0\ \texttt{match}\ \{\ldots\}}{\Delta;\Gamma \vdash e_0\ \texttt{match}\ \{\dot{c}\} : T}\ (\text{T-MATCH})$$

$$\frac{\Delta;\Gamma \vdash e_0 : E\langle\overline{P}\rangle \quad ftypes(E\langle\overline{P}\rangle, K) = \overline{Q} \quad \Delta;\Gamma, \overline{x} : \overline{Q} \vdash e : T}{\Delta;\Gamma \vdash \texttt{case}\ K(\overline{x}) \Rightarrow e : T\ IN\ e_0\ \texttt{match}\ \{\ldots\}}\ (\text{T-CTOR-PTN})$$

$$\frac{\Delta;\Gamma \vdash e_0 : E\langle\overline{P}\rangle \quad \Delta;\Gamma, x : E\langle\overline{P}\rangle \vdash e : T}{\Delta;\Gamma \vdash \texttt{case}\ x \Rightarrow e : T\ IN\ e_0\ \texttt{match}\ \{\ldots\}}\ (\text{T-NAMED-PTN})$$

$$\frac{\Delta;\Gamma \vdash e : T}{\Delta;\Gamma \vdash \texttt{case}\ \_ \Rightarrow e : T\ IN\ e_0\ \texttt{match}\ \{\ldots\}}\ (\text{T-WCARD-PTN})$$

$$\frac{\Delta = \overline{X} \quad \Delta \vdash \overline{T}\ \text{ok}}{\texttt{enum}\ E\ \langle\overline{X}\rangle\ \{\ K\ (\overline{v} : \overline{T})\ \}\ OK}\ (\text{T-ENUM})$$

$$\frac{\Delta = \overline{X}, \overline{Y} \quad \Delta \vdash \overline{T}, T\ \text{ok} \quad \texttt{class}\ C\ \langle\overline{X}\rangle\ (\ldots)\ \{\ldots\}\ or\ \texttt{trait}\ C\ \langle\overline{X} <: \overline{Q}\rangle\ \{\ldots\}\ or\ \texttt{trait}\ C\ \langle\overline{X} <: \overline{Q}\rangle\ \texttt{extends}\ \ldots\ \{\ldots\} \quad \Delta;\overline{x} : \overline{T}, this : C\langle\overline{X}\rangle \vdash e : T}{\texttt{def}\ m\ \langle\overline{Y}\rangle\ (\overline{x} : \overline{T}) : T = e\ OK\ IN\ C\langle\overline{X}\rangle}\ (\text{T-METHOD})$$

$$\frac{\Delta = \overline{X} \quad \Delta;\emptyset \vdash e : \texttt{bool}}{\texttt{proof}\ p\ \langle\overline{X}\rangle\ \{\ e\ \}\ OK}\ (\text{T-PROOF}) \qquad \frac{\Delta = \overline{X} \quad \Delta \vdash \overline{T}\ \text{ok} \quad \overline{M}\ OK\ IN\ C\langle\overline{X}\rangle}{\texttt{class}\ C\ \langle\overline{X}\rangle\ (\overline{v} : \overline{T})\ \{\ \overline{M}\ \}\ OK}\ (\text{T-CLASS1})$$

$$\frac{\Delta = \overline{X} \quad \Delta \vdash \overline{T}\ \text{ok} \quad \Delta \vdash I\langle\overline{P}\rangle\ \text{ok} \quad \texttt{trait}\ I\ \langle\ldots\rangle\ \{\ B\ \}\ \ or\ \ \texttt{trait}\ I\ \langle\ldots\rangle\ \texttt{extends}\ \ldots\ \{\ B\ \} \quad valNames(I\langle\overline{P}\rangle) \subset \overline{v} \quad declaredMethods(I\langle\overline{P}\rangle) \subset \overline{M} \quad \overline{M}\ OK\ IN\ C\langle\overline{X}\rangle}{\texttt{class}\ C\ \langle\overline{X}\rangle\ (\overline{v} : \overline{T})\ \texttt{extends}\ I\langle\overline{P}\rangle\{\ \overline{M}\ \}\ OK}\ (\text{T-CLASS2})$$

$$\frac{\Delta = \overline{X} \quad \Delta \vdash \overline{T}\ \text{ok} \quad \Delta \vdash I'\langle\overline{P}\rangle\ \text{ok} \quad \texttt{trait}\ I'\ \langle\ldots\rangle\ \{\ldots\}\ \ or\ \ \texttt{trait}\ I'\ \langle\ldots\rangle\ \texttt{extends}\ \ldots\ \{\ldots\} \quad B = \overline{valD} \cup \overline{methodD} \cup \overline{M} \quad \overline{M}\ OK\ IN\ I\langle\overline{X}\rangle \quad valNames(I'\langle\overline{P}\rangle) \subset \overline{valD} \quad declaredMethods(I'\langle\overline{P}\rangle) \subset (\overline{methodD} \cup \overline{M})}{\texttt{trait}\ I\ \langle\overline{X} <: \overline{T}\rangle\ \texttt{extends}\ I'\ \langle\overline{P}\rangle\ \{\ \overline{B}\ \}\ OK}\ (\text{T-TRAIT})$$

Figure 6 Typing VeriFx expressions.

$$
\begin{array}{lll}
e ::= \mathsf{num} \mid \mathsf{str} \mid \mathsf{true} \mid \mathsf{false} & & (\textit{primitive values}) \\
\quad \mid\ e[\,\vec{e}\,] \mid e[\,\vec{e}\,] := e \mid \lambda(\vec{x} : \vec{\vec{T}}).e & & \\
\quad \mid\ x \mid e \oplus e \mid e \otimes e \mid \neg e & & \\
\quad \mid\ \mathsf{match}(e,\ \mathsf{case}(\mathit{ptn},\ e)) & & (\textit{pattern matching}) \\
\quad \mid\ \mathsf{let}\, x = e\, \mathsf{in}\, e & & (\textit{let expression}) \\
\quad \mid\ \mathsf{if}(e,\ e,\ e) & & (\textit{conditional expression}) \\
\quad \mid\ e(e) & & (\textit{function call}) \\
\quad \mid\ f(\overline{T})(e) & & (\textit{function call with explicit type arguments}) \\
\quad \mid\ e.v & & (\textit{field access}) \\
\quad \mid\ \forall(\vec{x} : \vec{\vec{T}}).e \mid \exists(\vec{x} : \vec{\vec{T}}).e & & (\textit{quantified formulas}) \\
\quad \mid\ e \implies e & & (\textit{logical implication}) \\
\mathit{ptn} ::= K(\overline{x}) \mid x & & (\textit{patterns})
\end{array}
$$

Figure 7 All Core SMT expressions.

C Compiler Semantics

We now discuss the compiler semantics that was not discussed in the main body of the paper. First, we provide all compilation rules for expressions in Appendix C.1. Then, we provide all compilation rules for sets and maps in Appendices C.2 and C.3 respectively.

C.1 Compiling Expressions

Figure 8 shows the compilation rules for expressions. The operands of binary operators $\oplus$ are compiled recursively. A negated expression is compiled to the negation of the compiled expression. For if statements, the condition, and both branches are compiled recursively. In VeriFx, `this` can be used inside the body of a method to refer to the current object. The reference is compiled to a similar *this* reference in Core SMT which refers to the *this* parameter which is always the first parameter of any method (cf. compilation of class methods in Section 4.2). We explained how to compile the remaining expressions in Section 4.2.

Figure 9 shows the compilation rules for logic expressions which in VeriFx can only occur within the body of proofs. For quantified formulas the types of the variables $\vec{\vec{T}}$ and the formula e are compiled. For logical implications, the antecedent and the consequent are compiled recursively.

Finally, pattern match expressions are compiled to similar pattern match expressions in Core SMT (shown in Figure 10). To this end, every pattern is compiled recursively. Core SMT supports two types of patterns: constructor patterns $n_1(\overline{n_2})$ that match an algebraic data type constructor n_1 and bind its fields to the provided names $\overline{n_2}$, and wildcard patterns n that match any value and give it a name n. Every VeriFx pattern is compiled into the corresponding Core SMT pattern. The first pattern, $n_1(\overline{n_2})$, matches an ADT constructor n_1 and binds its fields to $\overline{n_2}$. It is compiled to an equivalent constructor pattern in Core SMT. The other two patterns match any expression and are compiled to an equivalent wildcard pattern in Core SMT.

C.2 Compiling Sets

In Section 4.4 we explained how basic set operations (`add`, `remove`, `contains`) and some advanced operations (`filter`, `map`) are compiled to Core SMT. Now, we explain how the remaining operations on sets are compiled. Figure 11 shows the compilation rules for

$$
\begin{array}{lcl}
[\![x]\!] & = & x \\
[\![e_1 \oplus e_2]\!] & = & [\![e_1]\!] \oplus [\![e_2]\!] \\
[\![!e]\!] & = & \neg [\![e]\!] \\
[\![\texttt{val}\ x : T = e_1\ \texttt{in}\ e_2]\!] & = & \mathsf{let}\ x = [\![e_1]\!]\ \mathsf{in}\ [\![e_2]\!] \\
[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] & = & \mathsf{if}([\![e_1]\!],\ [\![e_2]\!],\ [\![e_3]\!]) \\
[\![(\vec{x} : \vec{T}) \Rightarrow e]\!] & = & \lambda(\vec{x} : \vec{[\![T]\!]_t}).[\![e]\!] \\
[\![e_1(\overline{e_2})]\!] & = & [\![e_1]\!][\ \overline{[\![e_2]\!]}\ ] \\
[\![\texttt{new Set}\langle \texttt{T}\rangle(\ )]\!] & = & \lambda(x : [\![T]\!]_t).\mathsf{false} \\
[\![\texttt{new Map}\langle T, P\rangle(\ )]\!] & = & \lambda(x : [\![T]\!]_t).\mathsf{None}\langle [\![P]\!]_t\rangle() \\
[\![\texttt{new}\ C\langle \overline{T}\rangle(\overline{e})]\!] & = & C'\langle \overline{[\![T]\!]_t}\rangle(\overline{[\![e]\!]}) \\
\quad \text{where } C' = \texttt{str\_concat}(C, \text{"\_ctor"}) & & \\
[\![\texttt{new}\ K\langle \overline{T}\rangle(\overline{e})]\!] & = & K\langle \overline{[\![T]\!]_t}\rangle(\overline{[\![e]\!]}) \\
[\![e.v]\!] & = & [\![e]\!].v \\
[\![e_1.m\ \langle \overline{T}\rangle\ (\overline{e})]\!] & = & m'\langle \overline{[\![P]\!]_t},\ \overline{[\![T]\!]_t}\rangle([\![e_1]\!],\ \overline{[\![e]\!]}) \\
\quad \text{where } \mathit{typeof}(e_1) = C\langle \overline{P}\rangle & & \\
\quad \text{and } m' = \texttt{str\_concat}(C, \text{"\_"}, m) \text{ and } \overline{P} \cap \overline{T} = \emptyset & &
\end{array}
$$

Figure 8 Compiling expressions.

$$
\begin{array}{lcl}
[\![\texttt{forall}\ (\vec{x} : \vec{T})\ \texttt{.}\ e]\!] & = & \forall(\vec{x} : \vec{[\![T]\!]_t}).[\![e]\!] \\
[\![\texttt{exists}\ (\vec{x} : \vec{T})\ \texttt{.}\ e]\!] & = & \exists(\vec{x} : \vec{[\![T]\!]_t}).[\![e]\!] \\
[\![e_1 \implies e_2]\!] & = & [\![e_1]\!] \implies [\![e_2]\!]
\end{array}
$$

Figure 9 Compiling logical expressions.

operations over sets. The union of two sets e_1 and e_2 is compiled to a lambda which defines an array of elements x of type $[\![T]\!]_t$ containing only elements that are in at least one of the two sets, i.e. $[\![e_1]\!][\ x\ ] \vee [\![e_2]\!][\ x\ ]$. Similarly, the intersection of two sets e_1 and e_2 is compiled to a lambda which defines an array containing only elements that are in both sets, i.e. $[\![e_1]\!][\ x\ ] \wedge [\![e_2]\!][\ x\ ]$. For set difference, the lambda defines an array containing only elements that are in e_1 and not in e_2. A set e_1 is a subset of e_2 iff all elements from e_1 are also in e_2. A set e is non-empty if an element x exists that is in the set, i.e. $[\![e]\!][\ x\ ]$. A set e is empty if all elements x are not in the set. A predicate $e_2 : T \to \texttt{bool}$ holds for all elements of a set e_1 if for every element x that is in the set the predicate is true, i.e. $[\![e_1]\!][\ x\ ] \implies [\![e_p]\!][\ x\ ]$. A predicate $e_2 : T \to \texttt{bool}$ holds for at least one element of a set e_1 if there exists an element x that is in the set and for which the predicate holds, i.e. $[\![e_1]\!][\ x\ ] \wedge [\![e_p]\!][\ x\ ]$.

C.3 Compiling Maps

Maps are encoded as arrays from the key type to an optional value:

$$[\![\texttt{Map}\ \langle T, P\rangle]\!]_t = \mathsf{Array}\langle [\![T]\!]_t, \mathsf{Option}\langle [\![P]\!]_t\rangle\rangle$$

Optional values indicate the presence or absence of a value for a certain key. The option type is defined as an ADT with two constructors: $\mathsf{Some}(\mathit{value})$ which holds a value and $\mathsf{None}()$ indicating the absence of a value. An empty map corresponds to an array containing $\mathsf{None}()$ for every key and is created by a lambda that returns $\mathsf{None}()$ for every key:

$$
\begin{array}{lcl}
[\![e \texttt{ match } \{\overline{\texttt{case } r \Rightarrow e_c}\}]\!] & = & \mathsf{match}([\![e]\!],\ \overline{pat[\![\texttt{case } r \Rightarrow e_c]\!]}) \\
pat[\![\texttt{case } K(\overline{x}) \Rightarrow e]\!] & = & \mathsf{case}(K(\overline{x}),\ [\![e]\!]) \\
pat[\![\texttt{case } x \Rightarrow e]\!] & = & \mathsf{case}(x,\ [\![e]\!]) \\
pat[\![\texttt{case } \_ \Rightarrow e]\!] & = & \mathsf{case}(\_,\ [\![e]\!])
\end{array}
$$

Figure 10 Compiling pattern match expressions.

$$
\begin{array}{l}
[\![e_1.add(e_2)]\!] = [\![e_1]\!][\ [\![e_2]\!]\ ] := \mathsf{true} \\
[\![e_1.remove(e_2)]\!] = [\![e_1]\!][\ [\![e_2]\!]\ ] := \mathsf{false} \\
[\![e_1.contains(e_2)]\!] = [\![e_1]\!][\ [\![e_2]\!]\ ] \\
[\![e_1.filter(e_2)]\!] = \lambda(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \wedge [\![e_2]\!][\ x\ ] \quad \text{where } typeof(e_1) = \texttt{Set}\langle T\rangle \\
[\![e_1.map(e_2)]\!] = \lambda(y : [\![P]\!]_t).\exists(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \wedge [\![e_2]\!][\ x\ ] = y \\
\quad \text{where } typeof(e_1) = \texttt{Set}\langle T\rangle \wedge typeof(e_2) = T \to P \\
[\![e_1.union(e_2)]\!] = \lambda(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \vee [\![e_2]\!][\ x\ ] \\
\quad \text{where } typeof(e_1) = \texttt{Set}\langle T\rangle \wedge typeof(e_2) = \texttt{Set}\langle T\rangle \\
[\![e_1.intersect(e_2)]\!] = \lambda(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \wedge [\![e_2]\!][\ x\ ] \\
\quad \text{where } typeof(e_1) = \texttt{Set}\langle T\rangle \wedge typeof(e_2) = \texttt{Set}\langle T\rangle \\
[\![e_1.diff(e_2)]\!] = \lambda(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \wedge \neg[\![e_2]\!][\ x\ ] \\
\quad \text{where } typeof(e_1) = \texttt{Set}\langle T\rangle \wedge typeof(e_2) = \texttt{Set}\langle T\rangle \\
[\![e_1.subsetOf(e_2)]\!] = \forall(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \implies [\![e_2]\!][\ x\ ] \\
\quad \text{where } typeof(e_1) = \texttt{Set}\langle T\rangle \wedge typeof(e_2) = \texttt{Set}\langle T\rangle \\
[\![e.nonEmpty()]\!] = \exists(x : [\![T]\!]_t).[\![e]\!][\ x\ ] \quad \text{where } typeof(e) = \texttt{Set}\langle T\rangle \\
[\![e.isEmpty()]\!] = \forall(x : [\![T]\!]_t).\neg[\![e]\!][\ x\ ] \quad \text{where } typeof(e) = \texttt{Set}\langle T\rangle \\
[\![e_1.forall(e_p)]\!] = \forall(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \implies [\![e_p]\!][\ x\ ] \\
\quad \text{where } typeof(e_1) = \texttt{Set}\langle T\rangle \text{ and } typeof(e_p) = T \to \texttt{bool} \\
[\![e_1.exists(e_p)]\!] = \exists(x : [\![T]\!]_t).[\![e_1]\!][\ x\ ] \wedge [\![e_p]\!][\ x\ ] \\
\quad \text{where } typeof(e_1) = \texttt{Set}\langle T\rangle \text{ and } typeof(e_p) = T \to \texttt{bool}
\end{array}
$$

Figure 11 Compiling set operations.

$$[\![\texttt{new Map}\langle T, P\rangle(\ )]\!] = \lambda(x : [\![T]\!]_t).\mathsf{None}\langle[\![P]\!]_t\rangle()$$

Operations on maps are compiled as follows:

$$
\begin{array}{l}
map[\![e_m.add(e_k, e_v)]\!] = [\![e_m]\!][\ [\![e_k]\!]\ ] := \mathsf{Some}([\![e_v]\!]) \\
map[\![e_m.remove(e_k)]\!] = [\![e_m]\!][\ [\![e_k]\!]\ ] := \mathsf{None}\langle[\![V]\!]_t\rangle() \\
map[\![e_m.contains(e_k)]\!] = [\![e_m]\!][\ [\![e_k]\!]\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \\
map[\![e_m.get(e_k)]\!] = [\![e_m]\!][\ [\![e_k]\!]\ ].value \\
map[\![e_m.getOrElse(e_k, e_v)]\!] = \mathsf{if}([\![e_m]\!][\ [\![e_k]\!]\ ] = \mathsf{None}\langle[\![V]\!]_t\rangle(),\ [\![e_v]\!],\ [\![e_m]\!][\ [\![e_k]\!]\ ].value)
\end{array}
$$

A key-value pair $e_k \mapsto e_v$ is added to a map e_m by updating the entry for the compiled key $[\![e_k]\!]$ in the compiled array $[\![e_m]\!]$ with the compiled value, $\mathsf{Some}([\![e_v]\!])$. A key e_k is removed from a map e_m by updating the corresponding entry to $\mathsf{None}\langle[\![V]\!]_t\rangle()$, thereby indicating the absence of a value. Note that None is polymorphic but the type parameter cannot be inferred from the arguments; it is thus passed explicitly. A key e_k is present in a map e_m if the value that is associated to the key is not $\mathsf{None}\langle[\![V]\!]_t\rangle()$. The `get` method fetches the value that is associated to a key e_k in a map e_m. To this end, the compiled key $[\![e_k]\!]$ is accessed in the compiled map $[\![e_m]\!]$ and the value it holds is then fetched by accessing the `value` field of

the Some constructor. Even though the entry that is read from the array is an option type (i.e. a None or a Some) we can access the `value` field because the interpretation of `value` is underspecified in SMT. If the entry is a None, the SMT solver can assign any interpretation to the `value` field. Hence, the `get` method on maps should only be called if the key is known to be present in the map, e.g. after calling `contains`. VeriFx also features a safe variant, called `getOrElse`, which returns a default value if the key is not present.

Next, we explain how to encode the advanced map operations. Figure 12 defines the SMT encoding for all advanced map operations. The `keys` method on maps returns a set containing only the keys that are present in the map. Calls to `keys` on a map e_m of type `Map` $\langle$`K`, `V`$\rangle$ are compiled to a lambda which defines a set of keys k of the compiled key type $[\![K]\!]_t$ such that a key is present in the set iff it is present in the compiled map: $[\![e_m]\!][\ k\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle()$. A predicate e_p of type $(K, V) \rightarrow$ `bool` holds for all elements of a map e_m of type `Map` $\langle$`K`, `V`$\rangle$ iff it holds for every key k that is present in the map and its associated value:

$$\underbrace{[\![e_m]\!][\ k\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle()}_{e_m.contains(k)} \implies \underbrace{[\![e_p]\!][\ k, [\![e_m]\!][\ k\ ].value\ ]}_{e_p(k,\, e_m.get(k))}$$

Similarly, the `values` method returns a set with all values of the map. To this end, it defines an array containing all values for which at least one key exists that maps to that value.

A predicate e_p of type $(K, V) \rightarrow$ `bool` holds for at least one element of a map e_m of type `Map` $\langle$`K`, `V`$\rangle$ iff there exists a key k with associated value v that is present in the map and for which the predicate holds. Mapping a function e_f over the key-value pairs of a map e_m is encoded as a lambda that defines an array containing only the keys that are present in the compiled map $[\![e_m]\!]$ and whose values are the result of applying e_f on the original value, i.e. $\mathsf{Some}([\![e_f]\!][\ k, [\![e_m]\!][\ k\ ].value\ ])$. The `mapValues` method is similar except that it applies the provided function only on the value. A map e_m can be filtered using a predicate e_p such that the resulting map only contains key-value pairs that fulfill the predicate. Calls to `filter` are encoded as a lambda that defines an array containing only the key-value pairs that are in the compiled map:

$$\mathsf{if}(\overbrace{[\![e_m]\!][\ k\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle()}^{\text{in original map}} \wedge \overbrace{[\![e_p]\!][\ k, [\![e_m]\!][\ k\ ].value\ ]}^{\text{predicate holds}},\ \underbrace{\mathsf{Some}([\![e_m]\!][\ k\ ].value)}_{\text{then keep the value}},\ \underbrace{\mathsf{None}\langle[\![V]\!]_t\rangle()}_{\text{else not in the map}}\ )$$

To zip two maps e_{m_1} and e_{m_2} the compiler creates a lambda that defines an array containing only the keys that are present in both maps and the value is a tuple holding the corresponding values from both maps:

$$\mathsf{Some}(\mathit{Tuple\_ctor}([\![e_{m_1}]\!][\ k\ ].value, [\![e_{m_2}]\!][\ k\ ].value))$$

To combine two maps e_{m_1} and e_{m_2} with a function e_f the compiler creates a lambda that defines an array containing all the keys from e_{m_1} and e_{m_2}. If a key is present in both maps their values are combined using the provided function e_f:

$$\mathsf{Some}([\![e_f]\!][\ [\![e_{m_1}]\!][\ k\ ].value, [\![e_{m_2}]\!][\ k\ ].value\ ])$$

If a key-value pair is present in only one of the maps it is also present in the new map. If a key is not present in e_{m_1} neither in e_{m_2} then it is also not present in the resulting map.

Vectors and Lists. The encoding of sets and maps is very useful to build new data structures in VeriFx without having to encode them manually in SMT. For example, vectors and lists are implemented on top of maps. Internally, they map indices between 0 and $size - 1$

$$
\begin{aligned}
&map[\![e_m.keys()]\!] = \lambda(x : [\![K]\!]_t).[\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \\
&\quad \text{where } typeof(e_m) = \mathtt{Map}\langle K, V\rangle \\
&map[\![e_m.values()]\!] = \lambda(x : [\![V]\!]_t).\exists(k : [\![K]\!]_t).[\![e_m]\!][\ k\ ] = \mathsf{Some}(x) \\
&\quad \text{where } typeof(e_m) = \mathtt{Map}\langle K, V\rangle \\
&map[\![e_m.bijective()]\!] = \forall(k_1 : [\![K]\!]_t,\ k_2 : [\![K]\!]_t). \\
&\qquad\qquad (k_1 \neq k_2 \wedge [\![e_m]\!][\ k_1\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \wedge [\![e_m]\!][\ k_2\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle()) \\
&\qquad\qquad\quad \implies [\![e_m]\!][\ k_1\ ] \neq [\![e_m]\!][\ k_2\ ] \\
&\quad \text{where } typeof(e_m) = \mathtt{Map}\langle K, V\rangle \\
&map[\![e_m.forall(e_p)]\!] = \forall(x : [\![K]\!]_t).[\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \implies [\![e_p]\!][\ x, [\![e_m]\!][\ x\ ].value\ ] \\
&\quad \text{where } typeof(e_m) = \mathtt{Map}\langle K, V\rangle \text{ and } typeof(e_p) = (K, V) \to \mathtt{bool} \\
&map[\![e_m.exists(e_p)]\!] = \exists(x : [\![K]\!]_t).[\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \wedge [\![e_p]\!][\ x, [\![e_m]\!][\ x\ ].value\ ] \\
&\quad \text{where } typeof(e_m) = \mathtt{Map}\langle K, V\rangle \text{ and } typeof(e_p) = (K, V) \to \mathtt{bool} \\
&map[\![e_m.map(e_f)]\!] = \lambda(x : [\![K]\!]_t).\ \mathsf{if}([\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle(), \\
&\qquad\qquad \mathsf{Some}([\![e_f]\!][\ x, [\![e_m]\!][\ x\ ].value\ ]), \\
&\qquad\qquad \mathsf{None}\langle[\![W]\!]_t\rangle()) \\
&\quad \text{where } typeof(e_m) = \mathtt{Map}\langle K, V\rangle \text{ and } typeof(e_f) = V \to W \\
&map[\![e_m.mapValues(e_f)]\!] = \lambda(x : [\![K]\!]_t).\ \mathsf{if}([\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle(), \\
&\qquad\qquad \mathsf{Some}([\![e_f]\!][\ [\![e_m]\!][\ x\ ].value\ ]), \\
&\qquad\qquad \mathsf{None}\langle[\![W]\!]_t\rangle()) \\
&\quad \text{where } typeof(e_m) = \mathtt{Map}\langle K, V\rangle \text{ and } typeof(e_f) = V \to W \\
&map[\![e_m.filter(e_p)]\!] = \lambda(x : [\![K]\!]_t).\ \mathsf{if}([\![e_m]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \wedge [\![e_p]\!][\ x, [\![e_m]\!][\ x\ ].value\ ], \\
&\qquad\qquad \mathsf{Some}([\![e_m]\!][\ x\ ].value), \\
&\qquad\qquad \mathsf{None}\langle[\![V]\!]_t\rangle()) \\
&\quad \text{where } typeof(e_m) = \mathtt{Map}\langle K, V\rangle \text{ and } typeof(e_p) = (K, V) \to \mathtt{bool} \\
&map[\![e_{m_1}.zip(e_{m_2})]\!] = \lambda(x : [\![K]\!]_t).\ \mathsf{if}([\![e_{m_1}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \wedge [\![e_{m_2}]\!][\ x\ ] \neq \mathsf{None}\langle[\![W]\!]_t\rangle(), \\
&\qquad\qquad \mathsf{Some}(Tuple\_ctor([\![e_{m_1}]\!][\ x\ ].value, [\![e_{m_2}]\!][\ x\ ].value)), \\
&\qquad\qquad \mathsf{None}\langle[\![\mathtt{Tuple}\langle V, W\rangle]\!]_t\rangle()) \\
&\quad \text{where } typeof(e_{m_1}) = \mathtt{Map}\langle K, V\rangle \text{ and } typeof(e_{m_2}) = \mathtt{Map}\langle K, W\rangle \\
&map[\![e_{m_1}.combine(e_{m_2}, e_f)]\!] = \lambda(x : [\![K]\!]_t).\mathsf{if}([\![e_{m_1}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle() \wedge [\![e_{m_2}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle(), \\
&\qquad\qquad \mathsf{Some}([\![e_f]\!][\ [\![e_{m_1}]\!][\ x\ ].value, [\![e_{m_2}]\!][\ x\ ].value\ ]), \\
&\qquad\qquad \mathsf{if}([\![e_{m_1}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle(), \\
&\qquad\qquad\quad [\![e_{m_1}]\!][\ x\ ], \\
&\qquad\qquad\quad \mathsf{if}([\![e_{m_2}]\!][\ x\ ] \neq \mathsf{None}\langle[\![V]\!]_t\rangle(), \\
&\qquad\qquad\qquad [\![e_{m_2}]\!][\ x\ ], \\
&\qquad\qquad\qquad \mathsf{None}\langle[\![V]\!]_t\rangle()))) \\
&\quad \text{where } typeof(e_{m_1}) = \mathtt{Map}\langle K, V\rangle \text{ and } typeof(e_{m_2}) = \mathtt{Map}\langle K, V\rangle \text{ and } typeof(e_f) = (V, V) \to V \\
&map[\![e_m.toSet()]\!] = \lambda(x : \mathsf{Tuple}\langle[\![K]\!]_t, [\![V]\!]_t\rangle).[\![e_m]\!][\ x.fst\ ] = \mathsf{Some}(x.snd) \\
&\quad \text{where } typeof(e_m) = \mathtt{Map}\langle K, V\rangle
\end{aligned}
$$

Figure 12 Compiling advanced map operations.

to their value, and provide a traditional interface on top (cf. Figure 4). Note that this encoding of vectors and lists on top of maps is only used when verifying proofs in SMT; when compiling to a target language (e.g. Scala or JavaScript), VeriFx leverages the language's built-in vector and list data structures.

C.4 Compilation Example

Figure 13 shows a concrete example of a polymorphic set implemented in VeriFx and its compiled code in Core SMT. The *MSet* class defines a type parameter V corresponding to the type of elements it holds. It also contains one field *set* of type $Set\langle V\rangle$ and defines a polymorphic method *map* that takes a function $f : V \to W$ and returns a new *MSet* that results from applying f on every element. The compiled Core SMT code defines an ADT *MSet* with one type parameter V and one constructor *MSet_ctor*. The constructor defines one field *set* of sort $\mathsf{Array}\langle V, \mathsf{bool}\rangle$ which is the compiled sort for sets. In addition, a polymorphic *MSet_map* function is defined which takes two type parameters V and W

```
class MSet[V](set: Set[V]) {
  def map[W](f: V => W): MSet[W] =
    new MSet(this.set.map(f))
}
```

(a) A polymorphic class in VeriFx.

$$\begin{aligned}
&\mathsf{adt}\ \mathit{MSet}\langle V\rangle\{\ \mathit{MSet_ctor}(\mathit{set} : \mathsf{Array}\langle V, \mathsf{bool}\rangle)\ \}\\
&\mathsf{fun}\ \mathit{MSet_map}\langle V,\ W\rangle(\mathit{this} : \mathit{MSet}\langle V\rangle,\\
&\qquad\qquad\qquad\qquad f : \mathsf{Array}\langle V, W\rangle) : \mathit{MSet}\langle W\rangle =\\
&\quad \mathit{MSet_ctor}(\\
&\qquad \lambda(y : W).\exists(x : V).\mathit{this.set}[\ x\ ] \wedge f[\ x\ ] = y)
\end{aligned}$$

(b) Compiled Core SMT code.

Figure 13 Example of a polymorphic class in VeriFx and the compiled Core SMT code.

which correspond to *MSet*'s type parameter and *map*'s type parameter respectively. The function takes two arguments, the object that receives the call and the function f. The function's body calls the *MSet* constructor with the result of mapping f over the set.

D Implementation and Verification of the Buggy Map CRDT

Section 6.1.1 reported on our experience implementing and verifying the buggy and corrected map CRDTs proposed by Kleppmann [40]. In this appendix, we explain the implementation and verification of the *buggy* map CRDT in detail using code examples. We also discuss the counterexample found by VeriFx.

Specification 2 The buggy map CRDT algorithm, taken from [40].

on initialisation **do**
  $values := \{\}$
end on

on request to read value for key k **do**
  if $\exists t, v.(t, k, v) \in values$ **then return** v **else return** null
end on

on request to set key k to value v **do**
  $t :=$ newTimestamp() ▷ globally unique, e.g. Lamport timestamp
  broadcast $(\texttt{set}, t, k, v)$ by causal broadcast (including to self)
end on

on delivering $(\texttt{set}, t, k, v)$ by causal broadcast **do**
  $previous := \{(t', k', v') \in values \mid k' = k\}$
  if $previous = \{\} \vee \forall (t', k', v') \in previous.\ t' < t$ **then**
    $values := (values \setminus previous) \cup \{(t, k, v)\}$
  end if
end on

on request to delete key k **do**
  if $\exists t, v.\ (t, k, v) \in values$ **then**
    broadcast $(\texttt{delete}, t)$ by causal broadcast (including to self)
  end if
end on

on delivering $(\texttt{delete}, t)$ by causal broadcast **do**
  $values := \{(t', k', v') \in values \mid t' \neq t\}$
end on

D.1 Original Specification

The buggy map CRDT is a replicated dictionary storing key-value pairs where the values are regular values (i.e. no nested CRDTs). Specification 2 shows the specification of the buggy map CRDT. It defines a `read` operation to fetch the value associated with a certain

Listing 10 Excerpt from the implementation of the buggy map CRDT in VeriFx.

```
enum MapOp[K, V] { Put(k: K, v: V) | Delete(k: K) }
enum MapMsg[K, V] {
  PutMsg(t: Clock, k: K, v: V) |
  DeleteMsg(t: Clock, k: K) |
  NopMsg()
}
class KMap[K, V](clock: Clock, values: Map[K, Tuple[Clock, V]])
    extends CmRDT[MapOp[K, V], MapMsg[K, V], KMap[K, V]] {
  def contains(k: K): Boolean = this.values.contains(k)
  def get(k: K): V = this.values.get(k).snd

  // Prepare phase for the "put" operation
  // "put" corresponds to the "set" operation in the specification
  def preparePut(k: K, v: V) = {
    val t = this.clock
    new PutMsg(t, k, v)
  }
  // Effect phase for incoming "put" messages
  def put(t: Clock, k: K, v: V) = {
    val newClock = this.clock.sync(t)
    if (!this.values.contains(k) ||
        this.values.get(k).fst.smaller(t))
      new KMap(newClock, this.values.add(k, new Tuple(t, v)))
    else
      new KMap(newClock, this.values)
  }

  // Prepare phase for the "delete" operation
  def prepareDelete(k: K) = {
    if (this.values.contains(k)) {
      val t = this.values.get(k).fst
      new DeleteMsg[K, V](t, k)
    }
    else
      new NopMsg[K, V]()
  }
  // Effect phase for incoming "delete" messages
  def delete(t: Clock, k: K) = {
    if (this.values.contains(k) && this.values.get(k).fst == t)
      new KMap(this.clock, this.values.remove(k))
    else
      new KMap(this.clock, this.values)
  }

  override def equals(that: KMap[K, V]) =
    this.values == that.values
}
```

key, and two update operations: `set` and `delete` which assign a value to a key, respectively, delete a certain key. Every operation consists of two parts, a prepare phase (denoted "on request") that prepares a message to be broadcast to every replica (including itself), and an effect phase (denoted "on delivering") that applies the incoming message. We briefly explain both update operations:

`set`(k, v). When preparing a `set` operation that assigns a value v to a key k, the replica generates a new and globally unique timestamp t and broadcasts a $(\texttt{set}, t, k, v)$ message. When receiving such a message, the replica checks if it already stores a value for this key. If this is not the case, or if the previous value has a smaller timestamp $t' < t$, then it assigns the incoming value v to the key k, thereby, overriding any previous value. On the other hand, if the previous value has a bigger timestamp, then the incoming `set` message is ignored and the previous value is kept.

`delete`(k). When preparing a `delete` operation that deletes a key k, the replica fetches the timestamp t at which that key was inserted and broadcasts a $(\texttt{delete}, t)$ message. Note that the key itself is not added to the message because `set` always inserts a single key with a unique timestamp, hence, the timestamp t uniquely identifies the key. When receiving a $(\texttt{delete}, t)$ message, the replica removes the key that was inserted at timestamp t (if it is still present).

D.2 Implementation in VeriFx

Listing 10 shows the implementation of the buggy map CRDT in VeriFx. Every replica (i.e. every instance of the `KMap` class) maintains a local Lamport clock (consisting of a counter and a replica identifier) and keeps a dictionary that maps keys to timestamped values (i.e. a tuple containing a timestamp and a value). This implementation strategy is slightly different from Spec. 2 but more efficient because a dictionary allows for constant-time lookup, insertion, and deletion. We also extended the `DeleteMsg` such that it not only contains the timestamp t but also the key to be deleted (Line 4). This allows for an efficient implementation of `delete` since the replica knows which key to delete and does not have to loop over the map to find the key whose value has timestamp t.

We override equality - which by default is structural equality - because replicas have different Lamport clocks [43] as our implementation of the clocks keeps a unique replica identifier. Hence, two replicas are considered equal if they have the same values, independent of their clocks. We also renamed the `set` operation to `put`. The remainder of the implementation is a straightforward translation from the specification.

D.3 Verification in VeriFx

After implementing the buggy map CRDT in VeriFx we proceeded to the verification of the map. As explained in Section 6.1.1, VeriFx returned invalid counterexamples because it is not aware of the CRDT's assumptions which are *implicit* in the design. For instance, VeriFx does not know that replicas have unique IDs nor does it know the relation between a replica's clock and the values it observed. We need to encode these assumptions explicitly such that VeriFx does not consider cases that cannot occur in practice. To this end, we override the `reachable` and `compatible` predicates (cf. Section 5.1). The former defines which states are reachable (i.e. valid), while the latter defines which replicas are compatible.

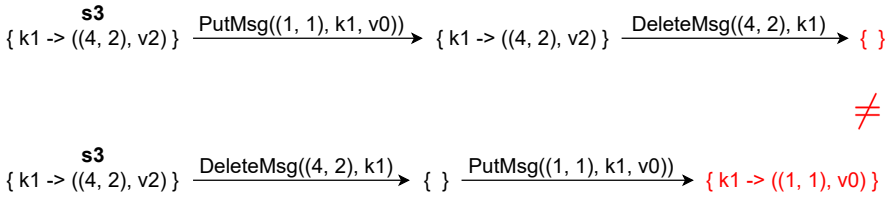
Listing 11 shows the implementation of the `reachable` and `compatible` predicates. First, we define a state to be reachable iff every value has a unique timestamp (Line 3 to 6) and all values have a timestamp whose count is smaller than the replica's local clock (Line 10).

Listing 11 Encoding the assumptions of the Map CRDT in VeriFx.

```
override def reachable(): Boolean = {
  // every value must have a unique timestamp
  !(exists(k1: K, k2: K) {
    k1 != k2 &&
    this.values.get(k1).fst == this.values.get(k2).fst
  }) &&
  // All the values in the map must have a timestamp < than our local clock
  // (since we sync our clock on incoming updates)
  this.values.values().forall((entry: Tuple[Clock, V]) =>
   entry.fst.counter < this.clock.counter)
}
private def noValueFromFuture(r1: KMap[K, V], r2: KMap[K, V]) {
  r1.values.values().forall((entry: Tuple[Clock, V]) => {
    val t = entry.fst
    (t.replica == r2.clock.replica) =>:
      (t.counter < r2.clock.counter)
  })
}
override def compatible(that: KMap[K, V]) = {
  // replicas have unique IDs
  (this.clock.replica != that.clock.replica) &&
  // we have no value from the future of the other replica
  this.noValueFromFuture(this, that) &&
  // the other did not observe a value from our future
  this.noValueFromFuture(that, this) &&
  // unique timestamps
  !(exists(k1: K, k2: K) {
    k1 != k2 && this.values.get(k1).fst == that.values.get(k2).fst
  }) &&
  // replicas cannot store different values for the same key and timestamp
  !(exists(k: K) {
    val thisTuple = this.values.get(k)
    val thisTimestamp = thisTuple.fst
    val thisValue = thisTuple.snd
    val thatTuple = that.values.get(k)
    val thatTimestamp = thatTuple.fst
    val thatValue = thatTuple.snd
    (thisTimestamp == thatTimestamp) && (thisValue != thatValue)
  })
}
```

```
enum V { v0 | v2 }
enum K { k1 }
val s1 = KMap(Clock(1, 1), Map())
val s2 = KMap(Clock(2, 9), Map(k1 -> (Clock(4, 2), v2)))
val s3 = KMap(Clock(3, 3), Map(k1 -> (Clock(4, 2), v2)))
val x = Put(k1, v0) // operation generated by s1
// The prepare phase will broadcast the following message:
// s1.preparePut(k1, v0) = PutMsg(Clock(1, 1), k1, v0))
val y = Delete(k1) // operation generated by s2
// s2.prepareDelete(k1) = DeleteMsg(Clock(4, 2), k1)
```

(a) Simplified counterexample returned by VeriFx.



(b) Visualization of the counterexample returned by VeriFx.

Figure 14 Counterexample for the buggy Map CRDT, found by VeriFx.

The latter property follows from the fact that the dictionary is constructed by successive insertions and every insertion synchronizes the replica's clock with the timestamp of the inserted element.

Second, we define two replicas to be compatible iff:

- they have unique IDs (Line 21),
- they did not observe values with a timestamp that is bigger than the current clock of the replica that inserted that value (Line 23 to 25) because that would mean that some replica observed a value from the future of the origin replica which is not possible,
- they do not have the same timestamp for different keys (Line 27 to 29) because every insertion inserts a single key with a unique timestamp,
- for every key k for which they store the same timestamp t they also store the same value v (Line 31 to 39) because every timestamp uniquely identifies one insertion: `PutMsg(t, k, v)`.

Clearly, the above assumptions are not straightforward and are in fact implicit in the original specification, but are nevertheless vital to the correctness of the algorithm. In practice, many CRDTs make similar implicit assumptions which is the reason they are complex and difficult to get right.

Counterexample. After defining all assumptions described above, VeriFx found a valid counterexample which is shown in Figure 14a. We simplified the counterexample by renaming the keys and values and removing those that do not affect the outcome. The counterexample is equivalent to the one that was found manually by Nair (cf. [40]). It consists of a corner case in which the `Put` and `Delete` operations do not commute and thus may cause replicas to diverge.

Recall that a counterexample is a mapping from variables (defined by the proof) to values that break the proof. In this case, the `CmRDTProof2` trait (cf. Section 5.1) that was used to check commutativity of the operations, defines three variables `s1`, `s2`, and `s3` representing

the state of the replicas, and two variables `x = Put(k1, v0)` and `y = Delete(k1)` representing concurrent operations that were generated by replica `s1` and `s2` respectively. These replicas first prepare a message for the operations (respectively, `PutMsg(Clock(1, 1), k1, v0))` and `DeleteMsg(Clock(4, 2), k1))` and broadcast those messages to every replica. Every replica receives these messages, possibly in a different order, and applies them.

Depending on the order in which replica `s3` applies the operations, the outcome is different. This is visualized in Figure 14b. If `s3` first processes the `DeleteMsg(Clock(4, 2), k1)` message then key `k1` is gone because the stored timestamp matches the timestamp that was requested to delete. Afterwards, when processing the `PutMsg(Clock(1, 1), k1, v0)` message, the replica will add key `k1` with value `v0`. When applying the operations the other way around, the outcome is different because the `PutMsg(Clock(1,1), k1, v0)` message is ignored since its timestamp is smaller than the timestamp `s3` currently stores for that key: `Clock(1,1) < Clock(4, 2)`. Later, when processing the `DeleteMsg(Clock(4,2), k1)` message, `s3` effectively deletes key `k1` because the timestamp matches the one that is stored. Thus, after the first execution, the resulting state contains key `k1`, whereas, after the second execution, `k1` is not present in the map. This explains the divergence bug.

E Verification of the MWS Set

Specification 3 describes the MWS Set, which associates a count to every element. An element is considered in the set if its count is strictly positive. `remove` decreases the element's count, while `add` increments the count by the amount that is needed to make it positive (or by 1 if it is already positive). Listing 12 shows the implementation of the MWS Set in VeriFx as a polymorphic class that extends the `CmRDT` trait (cf. Section 5.1.3). The type arguments passed to `CmRDT` correspond to the supported operations (`SetOp`s), the messages that are exchanged (`SetMsg`s), and the CRDT type itself (`MWSSet`). The `SetOp` enumeration defines two types of operations: `Add(e)` and `Remove(e)`.

The `MWSSet` class has a field, called `elements`, that maps elements to their count (Line 3). Like all op-based CRDTs, the `MWSSet` implements two methods: `prepare` and `effect`. The `prepare` method pattern matches on the operation and delegates it to the corresponding source method which prepares a `SetMsg` to be broadcast to all replicas. The class overrides the `enabledSrc` method to implement the source precondition on `remove`, as defined by Spec. 3. When replicas receive incoming messages, they are processed by the `effect` method which delegates them to the corresponding downstream method which performs the actual update. For example, the `removeDownstream` method processes incoming `RmvMsg`s by decreasing some count k' by 1. Unfortunately, k' is undefined in Spec. 3.

We believe that k' is either defined by the source replica and included in the propagated message (Spec. 4), or, k' is defined as the element's count at the downstream replica (Spec. 5). We implemented both possibilities in VeriFx (Listings 13 and 14) and verified them to find out which one, if any, is correct. To this end, the companion object of the `MWSSet` class (cf. Line 26 in Listing 12) extends the `CmRDTProof1` trait (cf. Section 5.1.3), passing along three type arguments: the type of operations `SetOp`, the type of messages being exchanged `SetMsg`, and the CRDT type constructor `MWSSet`. The object extends the `CmRDTProof1` trait since the `MWSSet` class is polymorphic and expects one type argument. When executing the proof inherited by the companion object, VeriFx automatically proves that the possibility implemented by Listing 14 is correct and that the one of Listing 13 is wrong. We thus successfully completed the MWS Set implementation using VeriFx's integrated verification capabilities.

Specification 3 Op-based MWS Set CRDT taken from Shapiro et al. [67].

```
payload set S = {(element, count), ...}
    initial E × {0}
query lookup (element e) : boolean b
    let b = ((e, k) ∈ S ∧ k > 0)
update add (element e)
    atSource (e) : integer j
        if ∃(e, k) ∈ S : k ≤ 0 then
            let j = |k| + 1
        else
            let j = 1
    downstream (e, j)
        let k' : (e, k') ∈ S
        S := S\{(e, k')} ∪ {(e, k' + j)}
update remove (element e)
    atSource (e)
        pre lookup(e)
    downstream (e)
        S := S\{(e, k')} ∪ {(e, k' − 1)}
```

Specification 4 Remove with k' defined at source.

```
update remove (element e)
    atSource (e) : integer k'
        pre lookup(e)
        let k' : (e, k') ∈ S
    downstream (e, k')
        S := S\{(e, k')} ∪ {(e, k' − 1)}
```

Specification 5 Remove with k' defined in downstream.

```
update remove (element e)
    atSource (e)
        pre lookup(e)
    downstream (e)
        let k' : (e, k') ∈ S
        S := S\{(e, k')} ∪ {(e, k' − 1)}
```

Listing 12 MWS Set implementation in VeriFx.

```
enum SetOp[V] { Add(e: V) | Remove(e: V) }
enum SetMsg[V] { AddMsg(e: V, dt: Int) | RmvMsg(e: V) }
class MWSSet[V](elements: Map[V, Int]) extends
      CmRDT[SetOp[V], SetMsg[V], MWSSet[V]] {
  override def enabledSrc(op: SetOp[V]) = op match {
    case Add(_) => true
    case Remove(e) => this.preRemove(e) }
  def prepare(op: SetOp[V]) = op match {
    case Add(e) => this.add(e)
    case Remove(e) => this.remove(e) }
  def effect(msg: SetMsg[V]) = msg match {
    case AddMsg(e, dt) => this.addDownstream(e, dt)
    case RmvMsg(e) => this.removeDownstream(e) }
  def lookup(e: V) = this.elements.getOrElse(e, 0) > 0
  def add(e: V): SetMsg[V] = {
    val count = this.elements.getOrElse(e, 0)
    val dt = if (count <= 0) (count * -1) + 1 else 1
    new AddMsg(e, dt) }
  def addDownstream(e: V, dt: Int): MWSSet[V] = {
    val count = this.elements.getOrElse(e, 0)
    new MWSSet(this.elements.add(e, count + dt)) }
  def preRemove(e: V) = this.lookup(e)
  def remove(e: V): SetMsg[V] = new RmvMsg(e)
  def removeDownstream(e: V): MWSSet[V] = {
    val kPrime = ??? // undefined in Specification 3
    new MWSSet(this.elements.add(e, kPrime - 1)) } }
object MWSSet extends CmRDTProof1[SetOp,SetMsg,MWSSet]
```

Listing 13 Computing k' at the source.

```
def remove(e: V): Tuple[V, Int] =
  new Tuple(e, this.elements.getOrElse(e, 0))
def removeDown(tup: Tuple[V, Int]): MWSSet[V] = {
  val e = tup.fst; val kPrime = tup.snd
  new MWSSet(this.elements.add(e, kPrime - 1)) }
```

Listing 14 Computing k' downstream.

```
def remove(e: V): V = e
def removeDown(e: V): MWSSet[V] = {
  val kPrime = this.elements.getOrElse(e, 0)
  new MWSSet(this.elements.add(e, kPrime - 1)) }
```