On the Complexity of Checking Consistency for Replicated Data Types

Ranadeep Biswas[1(✉)], Michael Emmi[2], and Constantin Enea[1]

[1] Université de Paris, IRIF, CNRS, 75013 Paris, France
{ranadeep,cenea}@irif.fr
[2] SRI International, New York, NY, USA
michael.emmi@sri.com

Abstract. Recent distributed systems have introduced variations of familiar abstract data types (ADTs) like counters, registers, flags, and sets, that provide high availability and partition tolerance. These *conflict-free replicated data types* (CRDTs) utilize mechanisms to resolve the effects of concurrent updates to replicated data. Naturally these objects weaken their consistency guarantees to achieve availability and partition-tolerance, and various notions of *weak consistency* capture those guarantees.

In this work we study the tractability of CRDT-consistency checking. To capture guarantees precisely, and facilitate symbolic reasoning, we propose novel logical characterizations. By developing novel reductions from propositional satisfiability problems, and novel consistency-checking algorithms, we discover both positive and negative results. In particular, we show intractability for replicated flags, sets, counters, and registers, yet tractability for replicated growable arrays. Furthermore, we demonstrate that tractability can be redeemed for registers when each value is written at most once, for counters when the number of replicas is fixed, and for sets and flags when the number of replicas and variables is fixed.

1 Introduction

Recent distributed systems have introduced variations of familiar abstract data types (ADTs) like counters, registers, flags, and sets, that provide high availability and partition tolerance. These *conflict-free replicated data types* (CRDTs) [33] efficiently resolve the effects of concurrent updates to replicated data. Naturally they weaken consistency guarantees to achieve availability and partition-tolerance, and various notions of *weak consistency* capture such guarantees [8,11,29,35,36].

In this work we study the tractability of CRDT consistency checking; Fig. 1 summarizes our results. In particular, we consider *runtime verification*: deciding

This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 678177).

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11562, pp. 324–343, 2019.
https://doi.org/10.1007/978-3-030-25543-5_19

Data Types	Complexity
Add-Wins Set, Remove-Wins Set	NP-complete
Enable-Wins Flag, Disable-Wins Flag	NP-complete
Sets & Flags — with bounded domains	PTIME
Last-Writer-Wins Register (LWW)	NP-complete
Multi-Value Register (MVR)	NP-complete
Registers – with unique values	PTIME
Replicated Counters	NP-complete
Counters – with bounded replicas	PTIME
Replicated Growable Array (RGA)	PTIME

Fig. 1. The complexity of consistency checking for various replicated data types. We demonstrate intractability and tractability results in Sects. 3 and 4, respectively.

whether a given execution of a CRDT is consistent with its ADT specification. This problem is particularly relevant as distributed-system testing tools like Jepsen [25] are appearing; without efficient, general consistency-checking algorithms, such tools could be limited to specialized classes of errors like node crashes.

Our setting captures executions across a set of replicas as per-replica sequences of operations called *histories*. Roughly speaking, a history is *consistent* so long as each operation's return value can be justified according to the operations that its replica has observed so far. In the setting of CRDTs, the determination of a replica's observations is essentially an implementation choice: replicas are only obliged to observe their own operations, and the predecessors of those it has already observed. This relatively-weak constraint on replicas' observations makes the CRDT consistency checking problem unique.

Our study proceeds in three parts. First, to precisely characterize the consistency of various CRDTs, and facilitate symbolic reasoning, we develop novel logical characterizations to capture their guarantees. Our logical models are built on a notion of *abstract execution*, which relates the operations of a given history with three separate relations: a *read-from* relation, governing the observations from which a given operation constitutes its own return value; a *happens-before* relation, capturing the causal relationships among operations; and a *linearization* relation, capturing any necessary arbitration among non-commutative effects which are executed concurrently, e.g., following a *last-writer-wins* policy. Accordingly, we capture data type specifications with logical axioms interpreted over the read-from, happens-before, and linearization relations of abstract executions, reducing the consistency problem to: does there exist an abstract execution over the given history which satisfies the axioms of the given data type?

Second, we demonstrate the intractability of several replicated data types by reduction from propositional satisfiability (SAT) problems. In particular, we consider the 1-in-3 SAT problem [19], which asks for a truth assignment to

the variables of a given set of clauses such that exactly one literal per clause is assigned true. Our reductions essentially simulate the existential choice of a truth assignment with the existential choice of the read-from and happens-before relations of an abstract execution. For a given 1-in-3 SAT instance, we construct a history of replicas obeying carefully-tailored synchronization protocols, which is consistent exactly when the corresponding SAT instance is positive.

Third, we develop tractable consistency-checking algorithms for individual data types and special cases: replicated growing arrays; multi-value and last-writer-wins registers, when each value is written only once; counters, when replicas are bounded; and sets and flags, when their sizes are also bounded. While the algorithms for each case are tailored to the algebraic properties of the data types they handle, they essentially all function by constructing abstract executions incrementally, processing replicas' operations in prefix order.

The remainder of this article is organized around our three key contributions:

1. We develop novel logical characterizations of consistency for the replicated register, flag, set, counter, and array data types (Sect. 2);
2. We develop novel reductions from propositional satisfiability problems to consistency checking to demonstrate intractability for replicated flags, sets, counters, and registers (Sect. 3); and
3. We develop tractable consistency-checking algorithms for replicated growable arrays, registers, when written values are unique, counters, when replicas are bounded, and sets and flags, when their sizes are also bounded (Sects. 4–6).

Section 7 overviews related work, and Sect. 8 concludes.

2 A Logical Characterization of Replicated Data Types

In this section we describe an axiomatic framework for defining the semantics of replicated data types. We consider a set of method names $\mathbb{M}$, and that each method $\mathsf{m} \in \mathbb{M}$ has a number of arguments and a return value sampled from a data domain $\mathbb{D}$. We will use operation labels of the form $\mathsf{m}(a) \overset{i}{\Rightarrow} b$ to represent the call of a method $\mathsf{m} \in \mathbb{M}$, with argument $a \in \mathbb{D}$, and resulting in the value $b \in \mathbb{D}$. Since there might be multiple calls to the same method with the same arguments and result, labels are tagged with a unique identifier i. We will ignore identifiers when unambiguous.

The interaction between a data type implementation and a client is represented by a *history* $h = \langle \mathsf{Op}, \mathsf{ro} \rangle$ which consists of a set of operation labels Op and a partial *replica order* ro ordering operations issued by the client on the same replica. Usually, ro is a union of sequences, each sequence representing the operations issued on the same replica, and the *width* of ro, i.e., the maximum number of mutually-unordered operations, gives the number of replicas in a given history.

To characterize the set of histories $h = \langle \mathsf{Op}, \mathsf{ro} \rangle$ admitted by a certain replicated data type, we use *abstract executions* $e = \langle \mathsf{rf}, \mathsf{hb}, \mathsf{lin} \rangle$, which include:

- a *read-from* binary relation rf over operations in Op, which identifies the set of updates needed to "explain" a certain return value, e.g., a write operation explaining the return value of a read,
- a strict partial *happens-before* order hb, which includes ro and rf, representing the causality constraints in an execution, and
- a strict total *linearization* order lin, which includes hb, used to model conflict resolution policies based on timestamps.

In this work, we consider replicated data types which satisfy *causal consistency* [26], i.e., updates which are related by cause and effect relations are observed by all replicas in the same order. This follows from the fact that the happens-before order is constrained to be a partial order, and thus transitive (other forms of weak consistency don't pose this constraint). Some of the replicated data types we consider in this paper do *not* consider resolution policies based on timestamps and in those cases, the linearization order can be ignored.

$\underline{\textsc{ReadFrom}(R)}$

$\forall o_1, o_2.\ \mathsf{rf}(o_1, o_2) \Rightarrow R(o_1, o_2)$

$\underline{\textsc{ReadFromMaximal}(R)}$

$\forall o_1, o_2, o_3.\ \mathsf{rf}(o_1, o_2) \wedge R(o_3, o_2) \Rightarrow$
$\neg\mathsf{hb}(o_1, o_3) \vee \neg\mathsf{hb}(o_3, o_2)$

$\underline{\textsc{ReadAllMaximals}(R)}$

$\forall o_1, o_2.\ \mathsf{hb}(o_1, o_2) \wedge R(o_1, o_2)$
$\Rightarrow \exists o_3.\ \mathsf{hb}^*(o_1, o_3) \wedge \mathsf{rf}(o_3, o_2)$

$\underline{\textsc{ClosedRF}(R)}$

$\forall o_1, o_2, o_3.\ R(o_1, o_2) \wedge \mathsf{hb}(o_1, o_3)$
$\wedge\ \mathsf{rf}(o_3, o_2) \Rightarrow \mathsf{rf}(o_1, o_2)$

$\underline{\textsc{RetvalSet}(X, v, Y)}$

$\forall o_1.\ \mathsf{meth}(o_1) = X \wedge \mathsf{ret}(o_1) = v$
$\Leftrightarrow \exists o_2.\ \mathsf{rf}(o_2, o_1) \wedge \mathsf{meth}(o_2) = Y$
$\wedge\ \mathsf{arg}(o_1) = \mathsf{arg}(o_2)$

$\underline{\textsc{RetvalCounter}}$

$\forall o_1.\ \mathsf{meth}(o_1) = \mathsf{read}$
$\Rightarrow \mathsf{ret}(o_1) = |\{o_2 : \mathsf{meth}(o_2) = \mathsf{inc} \wedge \mathsf{rf}(o_2, o_1)\}|$
$- |\{o_2 : \mathsf{meth}(o_2) = \mathsf{dec} \wedge \mathsf{rf}(o_2, o_1)\}|$

$\underline{\textsc{LinLWW}}$

$\forall o_1, o_2, o_3.\ \mathsf{rf}(o_1, o_2) \wedge \mathsf{meth}(o_3) = \mathsf{write}$
$\wedge\ \mathsf{arg}_1(o_3) = \mathsf{arg}(o_2) \wedge \mathsf{hb}(o_3, o_2) \Rightarrow \mathsf{lin}(o_3, o_1)$

$\underline{\textsc{RetvalReg}}$

$\forall o_1, v.\mathsf{meth}(o_1) = \mathsf{read} \wedge v \in \mathsf{ret}(o_1) \Rightarrow \exists! o_2.\mathsf{rf}(o_2, o_1) \wedge \mathsf{meth}(o_2) = \mathsf{write} \wedge \mathsf{arg}_2(o_2) = v$

Fig. 2. The axiomatic semantics of replicated data types. Quantified variables are implicitly distinct, and $\exists! o$ denotes the existence of a unique operation o.

A *replicated data type* is defined by a set of first-order axioms Φ characterizing the relations in an abstract execution. A history h is *admitted* by a data type when there exists an abstract execution e such that $\langle h, e\rangle \models \Phi$. The satisfaction relation $\models$ is defined as usual in first order logic. The *admissibility problem* is the problem of checking whether a history h is admitted by a given data type.

In the following, we define the replicated data types with respect to which we study the complexity of the admissibility problem. The axioms used to

define them are listed in Figs. 2 and 3. These axioms use the function symbols meth-od, arg-ument, and ret-urn interpreted over operation labels, whose semantics is self-explanatory.

2.1 Replicated Sets and Flags

The Add-Wins Set and Remove-Wins Set [34] are two implementations of a replicated set with operations $\mathsf{add}(x)$, $\mathsf{remove}(x)$, and $\mathsf{contains}(x)$ for adding, removing, and checking membership of an element x. Although the meaning of these methods is self-evident from their names, the result of conflicting concurrent operations is not evident. When concurrent $\mathsf{add}(x)$ and $\mathsf{remove}(x)$ operations are delivered to a certain replica, the Add-Wins Set chooses to keep the element x in the set, so every subsequent invocation of $\mathsf{contains}(x)$ on this replica returns *true*, while the Remove-Wins Set makes the dual choice of removing x from the set.

The formal definition of their semantics uses abstract executions where the read-from relation associates sets of $\mathsf{add}(x)$ and $\mathsf{remove}(x)$ operations to $\mathsf{contains}(x)$ operations. Therefore, the predicate $\mathsf{ReadOk}(o_1, o_2)$ is defined by

$$\mathsf{meth}(o_1) \in \{\mathsf{add}, \mathsf{remove}\} \wedge \mathsf{meth}(o_2) = \mathsf{contains} \wedge \mathsf{arg}(o_1) = \mathsf{arg}(o_2)$$

and the Add-Wins Set is defined by the following set of axioms:

$$\textsc{ReadFrom}(\mathsf{ReadOk}) \wedge \textsc{ReadFromMaximal}(\mathsf{ReadOk}) \wedge$$
$$\textsc{ReadAllMaximals}(\mathsf{ReadOk}) \wedge \textsc{RetvalSet}(\mathsf{contains}, \mathit{true}, \mathsf{add})$$

ReadFromMaximal says that every operation read by a $\mathsf{contains}(x)$ is maximal among its hb-predecessors that add or remove x while ReadAllMaximals says that all such maximal hb-predecessors are read. The RetvalSet instantiation ensures that a $\mathsf{contains}(x)$ returns *true* iff it reads-from at least one $\mathsf{add}(x)$.

The definition of the Remove-Wins Set is similar, except for the parameters of RetvalSet, which become $\textsc{RetvalSet}(\mathsf{contains}, \mathit{false}, \mathsf{remove})$, i.e., a $\mathsf{contains}(x)$ returns *false* iff it reads-from at least one $\mathsf{remove}(x)$.

The Enable-Wins Flag and Disable-Wins Flag are implementations of a set of flags with operations: $\mathsf{enable}(x)$, $\mathsf{disable}(x)$, and $\mathsf{read}(x)$, where $\mathsf{enable}(x)$ turns the flag x to true, $\mathsf{disable}(x)$ turns x to false, while $\mathsf{read}(x)$ returns the state of the flag x. Their semantics is similar to the Add-Wins Set and Remove-Wins Set, respectively, where $\mathsf{enable}(x)$, $\mathsf{disable}(x)$, and $\mathsf{read}(x)$ play the role of $\mathsf{add}(x)$, $\mathsf{remove}(x)$, and $\mathsf{contains}(x)$, respectively. Their axioms are defined as above.

2.2 Replicated Registers

We consider two variations of replicated registers called Multi-Value Register (MVR) and Last-Writer-Wins Register (LWW) [34] which maintain a set of registers and provide $\mathsf{write}(x,v)$ operations for writing a value v on a register x and $\mathsf{read}(x)$ operations for reading the content of a register x (the domain of values is kept unspecified since it is irrelevant). While a $\mathsf{read}(x)$ operation of

MVR returns *all* the values written by concurrent writes which are maximal among its happens-before predecessors, therefore, leaving the responsibility for solving conflicts between concurrent writes to the client, a $\mathsf{read}(x)$ operation of LWW returns a single value chosen using a conflict-resolution policy based on timestamps. Each written value is associated to a timestamp, and a read operation returns the most recent value w.r.t. the timestamps. This order between timestamps is modeled using the linearization order of an abstract execution.

Therefore, the predicate $\mathsf{ReadOk}(o_1, o_2)$ is defined by

$$\mathsf{meth}(o_1) = \mathsf{write} \wedge \mathsf{meth}(o_2) = \mathsf{read} \wedge \mathsf{arg}_1(o_1) = \mathsf{arg}(o_2) \wedge \mathsf{arg}_2(o_1) \in \mathsf{ret}(o_2)$$

(we use $\mathsf{arg}_1(o_1)$ to denote the first argument of a write operation, i.e., the register name, and $\mathsf{arg}_2(o_1)$ to denote its second argument, i.e., the written value) and the MVR is defined by the following set of axioms:

$$\textsc{ReadFrom}(\mathsf{ReadOk}) \wedge \textsc{ReadFromMaximal}(\mathsf{ReadOk}) \wedge$$
$$\textsc{ReadAllMaximals}(\mathsf{ReadOk}) \wedge \textsc{RetvalReg}$$

where RetvalReg ensures that a $\mathsf{read}(x)$ operation reads from a $\mathsf{write}(x,v)$ operation, for each value v in the set of returned values[1].

LWW is obtained from the definition of MVR by replacing ReadAllMaximals with the axiom LinLWW which ensures that every $\mathsf{write}(x,\_)$ operation which happens-before a $\mathsf{read}(x)$ operation is linearized before the $\mathsf{write}(x,\_)$ operation from where the $\mathsf{read}(x)$ takes its value (when these two write operations are different). This definition of LWW is inspired by the "bad-pattern" characterization in [6], corresponding to their causal convergence criterion.

2.3 Replicated Counters

The replicated counter datatype [34] maintains a set of counters interpreted as integers (the counters can become negative). This datatype provides operations $\mathsf{inc}(x)$ and $\mathsf{dec}(x)$ for incrementing and decrementing a counter x, and $\mathsf{read}(x)$ operations to read the value of the counter x. The semantics of the replicated counter is quite standard: a $\mathsf{read}(x)$ operation returns the value computed as the difference between the number of $\mathsf{inc}(x)$ operations and $\mathsf{dec}(x)$ operations among its happens-before predecessors. The axioms defined below will enforce the fact that a $\mathsf{read}(x)$ operation reads-from all its happens-before predecessors which are $\mathsf{inc}(x)$ or $\mathsf{dec}(x)$ operations.

Therefore, the predicate $\mathsf{ReadOk}(o_1, o_2)$ is defined by

$$\mathsf{meth}(o_1) \in \{\mathsf{inc}, \mathsf{dec}\} \wedge \mathsf{meth}(o_2) = \mathsf{read} \wedge \mathsf{arg}(o_1) = \mathsf{arg}(o_2)$$

and the replicated counter is defined by the following set of axioms:

$$\textsc{ReadFrom}(\mathsf{ReadOk}) \wedge \textsc{ClosedRF}(\mathsf{ReadOk}) \wedge \textsc{RetvalCounter}.$$

[1] For simplicity, we assume that every history contains a set of write operations writing the initial values of variables, which precede every other operation in replica order.

ReadFromRGA

$$\forall o_2.\ \mathsf{meth}(o_2) = \mathsf{addAfter} \Rightarrow \mathsf{arg}_1(o_2) = \circ \vee \exists o_1.\ \mathsf{meth}(o_1) = \mathsf{addAfter} \wedge \mathsf{arg}_2(o_1) = \mathsf{arg}_1(o_2) \wedge \mathsf{rf}(o_1, o_2)$$
$$\wedge\ \mathsf{meth}(o_2) = \mathsf{remove} \Rightarrow \exists o_1.\ \mathsf{meth}(o_1) = \mathsf{addAfter} \wedge \mathsf{arg}_2(o_1) = \mathsf{arg}(o_2) \wedge \mathsf{rf}(o_1, o_2)$$
$$\wedge\ \mathsf{meth}(o_2) = \mathsf{read} \Rightarrow \forall v \in \mathsf{ret}(o_2)\ \exists o_1.\mathsf{meth}(o_1) = \mathsf{addAfter} \wedge \mathsf{arg}_2(o_1) = v \wedge \mathsf{rf}(o_1, o_2)$$

RetvalRGA

$$\forall o_1, o_2.\ \mathsf{meth}(o_1) = \mathsf{read} \wedge \mathsf{meth}(o_2) = \mathsf{addAfter} \wedge \mathsf{hb}(o_2, o_1) \wedge \mathsf{arg}_2(o_2) \notin \mathsf{ret}(o_1) \Rightarrow \exists o_3.\ \mathsf{meth}(o_3) = \mathsf{remove} \wedge \mathsf{arg}(o_3) = \mathsf{arg}_2(o_2) \wedge \mathsf{rf}(o_3, o_1)$$

LinRGA

$$\forall o_1, o_2.\ \big(\mathsf{meth}(o_1) = \mathsf{meth}(o_2) = \mathsf{addAfter} \wedge \mathsf{arg}_1(o_1) = \mathsf{arg}_1(o_2) \wedge \exists o_3, o_4, o_5.\ \mathsf{meth}(o_3) = \mathsf{meth}(o_4) = \mathsf{addAfter} \wedge \mathsf{rf}^*_{\mathsf{addAfter}}(o_1, o_3) \wedge \mathsf{rf}^*_{\mathsf{addAfter}}(o_2, o_4) \wedge \mathsf{meth}(o_5) = \mathsf{read} \wedge \mathsf{arg}_2(o_4) <_{o_5} \mathsf{arg}_2(o_3)\big) \Rightarrow \mathsf{lin}(o_1, o_2)$$

Fig. 3. Axioms used to define the semantics of RGA.

2.4 Replicated Growable Array

The Replicated Growing Array (RGA) [32] is a replicated list used for text-editing applications. RGA supports three operations: addAfter(a,b) which adds the character b immediately after the occurrence of the character a assumed to be present in the list, remove(a) which removes a assumed to be present in the list, and read() which returns the list contents. It is assumed that a character is added at most once[2]. The conflicts between concurrent addAfter operations that add a character immediately after the same character is solved using timestamps (i.e., each added character is associated to a timestamp and the order between characters depends on the order between the corresponding timestamps), which in the axioms below are modeled by the linearization order.

Figure 3 lists the axioms defining RGA. ReadFromRGA ensures that:

- every addAfter(a,b) operation reads-from the addAfter(_,a) adding the character a, except when $a = \circ$ which denotes the "root" element of the list[3],
- every remove(a) operation reads-from the operation adding a, and
- every read operation returning a list containing a reads-from the operation addAfter(_,a) adding a.

Then, RetvalRGA ensures that a read operation o_1 happening-after an operation adding a character a reads-from a remove(a) operation when a doesn't occur in the list returned by o_1 (the history must contain a remove(a) operation because otherwise, a should have occurred in the list returned by the read).

Finally, LinRGA models the conflict resolution policy by constraining the linearization order between addAfter(a,_) operations adding some character

[2] In a practical context, this can be enforced by tagging characters with replica identifiers and sequence numbers.

[3] This element is not returned by read operations.

immediately after the same character a. As a particular case, LinRGA enforces that $\mathsf{addAfter}(a,b)$ is linearized before $\mathsf{addAfter}(a,c)$ when a read operation returns a list where c precedes b ($\mathsf{addAfter}(a,b)$ results in the list $a \cdot b$ and applying $\mathsf{addAfter}(a,c)$ on $a \cdot b$ results in the list $a \cdot c \cdot b$). However, this is not sufficient: assume that the history contains the two operations $\mathsf{addAfter}(a,b)$ and $\mathsf{addAfter}(a,c)$ along with two operations $\mathsf{remove}(b)$ and $\mathsf{addAfter}(b,d)$. Then, a read operation returning the list $a \cdot c \cdot d$ must enforce that $\mathsf{addAfter}(a,b)$ is linearized before $\mathsf{addAfter}(a,c)$ because this is the only order between these two operations that can lead to the result $a \cdot c \cdot d$, i.e., executing $\mathsf{addAfter}(a,b)$, $\mathsf{addAfter}(b,d)$, $\mathsf{remove}(b)$, $\mathsf{addAfter}(a,c)$ in this order. LinRGA deals with any scenario where arbitrarily-many characters can be removed from the list: $\mathsf{rf}^*_{\mathsf{addAfter}}$ is the reflexive and transitive closure of the projection of rf on $\mathsf{addAfter}$ operations and $<_{o_5}$ denotes the order between characters in the list returned by the read operation o_5.

3 Intractability for Registers, Sets, Flags, and Counters

In this section we demonstrate that checking the consistency is intractable for many widely-used data types. While this is not completely unexpected, since some related consistency-checking problems like sequential consistency are also intractable [20], this contrasts recent tractability results for checking strong consistency (i.e., linearizability) of common non-replicated data types like sets, maps, and queues [15]. In fact, in many cases we show that intractability even holds if the number of replicas is fixed.

Our proofs of intractability follow the general structure of Gibbons and Korach's proofs for the intractability of checking sequential consistency (SC) for atomic registers with read and write operations [20]. In particular, we reduce a specialized type of NP-hard propositional satisfiability (SAT) problem to checking whether histories are admitted by a given data type. While our construction borrows from Gibbons and Korach's, the adaptation from SC to CRDT consistency requires a significant extension to handle the consistency relaxation represented by abstract executions: rather than a direct sequencing of threads' operations, CRDT consistency requires the construction of three separate relations: read-from, happens-before, and linearization.

Technically, our reductions start from the 1-in-3 SAT problem [19]: given a propositional formula $\bigwedge_{i=1}^{m}(\alpha_i \vee \beta_i \vee \gamma_i)$ over variables $x_1, \ldots, x_n$ with only positive literals, i.e., $\alpha_i, \beta_i, \gamma_i \in \{x_1, \ldots, x_n\}$, does there exist an assignment to the variables such that exactly one of $\alpha_i, \beta_i, \gamma_i$ per clause is assigned *true*? The proofs of Theorems 1 and 2 reduce 1-in-3 SAT to CRDT consistency checking.

Theorem 1. *The admissibility problem is NP-hard when the number of replicas is fixed for the following data types: Add-Wins Set, Remove-Wins Set, Enable-Wins Flag, Disable-Wins Flag, Multi-Value Register, and Last-Writer-Wins Register.*

	Replica 0	Replica 1	Replica 2
Round 0	Enable(x_1) ... Enable(x_n)	Disable(x_1) ... Disable(x_n)	
Barrier 1	Enable(y_0) Read(y_1) = *true* Read(y_2) = *true*	Enable(y_1) Read(y_0) = *true* Read(y_2) = *true*	Enable(y_2) Read(y_0) = *true* Read(y_1) = *true*
Round 1	Read(α_1) = *true* Read(β_1) = *false* Read(γ_1) = *false* Disable(α_1) Enable(β_1)	Read(α_1) = *false* Read(β_1) = *true* Read(γ_1) = *false* Disable(β_1) Enable(γ_1)	Read(α_1) = *false* Read(β_1) = *false* Read(γ_1) = *true* Disable(γ_1) Enable(α_1)
Barrier 2	Disable(y_0) Read(y_1) = *false* Read(y_2) = *false*	Disable(y_1) Read(y_0) = *false* Read(y_2) = *false*	Disable(y_2) Read(y_0) = *false* Read(y_1) = *false*
	...	...	...
Round m	Read(α_m) = *true* Read(β_m) = *false* Read(γ_m) = *false* Disable(α_m) Enable(β_m)	Read(α_m) = *false* Read(β_m) = *true* Read(γ_m) = *false* Disable(β_m) Enable(γ_m)	Read(α_m) = *false* Read(β_m) = *false* Read(γ_m) = *true* Disable(γ_m) Enable(α_m)

Fig. 4. The encoding of a 1-in-3 SAT problem $\bigwedge_{i=1}^{m}(\alpha_i \vee \beta_i \vee \gamma_i)$ over variables $x_1, \ldots, x_n$ as a 3-replica history of a flag data type. Besides the flag variable x_j for each propositional variable x_j, the encoding adds per-replica variables y_j for synchronization barriers.

Proof. We demonstrate a reduction from the 1-in-3 SAT problem. For a given problem $p = \bigwedge_{i=1}^{m}(\alpha_i \vee \beta_i \vee \gamma_i)$ over variables $x_1, \ldots, x_n$, we construct a 3-replica history h_p of the flag data type — either enable- or disable-wins — as illustrated in Fig. 4. The encoding includes a flag variable x_j for each propositional variable x_j, along with a per-replica flag variable y_j used to implement synchronization barriers. Intuitively, executions of h_p proceed in $m + 1$ rounds: the first round corresponds to the assignment of a truth valuation, while subsequent rounds check the validity of each clause given the assignment. The reductions to sets and registers are slight variations on this proof, in which the Read, Enable, and Disable operations are replaced with Contains, Add, and Remove, respectively, and Read and Writes of values 1 and 0, respectively.

It suffices to show that the constructed history h_p is admitted if and only if the given problem p is satisfiable. Since the flag data type does not constrain the linearization relation of its abstract executions, we regard only the read-from and happens-before components. It is straightforward to verify that the happens-before relations of h_p's abstract executions necessarily order:

1. every pair of operations in distinct rounds — due to barriers; and
2. every operation in a given round, over all replicas, without interleaving the operations of distinct replicas within the same round — since a replica's reads in a given round are only consistent with the other replicas' after the re-enabling and -disabling of flag variables.

In other words, replicas appear to execute atomically per round, in a round-robin fashion. Furthermore, since all operations in a given round happen before the operations of subsequent rounds, the values of flag variables are consistent across rounds —i.e., as read by the first replica to execute in a given round — and determined in the initial round either by conflict resolution — i.e., enable- or disable-wins — or by happens-before, in case conflict resolution would have been inconsistent with subsequent reads.

In the "if" direction, let $\boldsymbol{r} \in \{0,1,2\}^m$ be the positions of positively-assigned variables in each clause, e.g., $r_i = 0$ implies $\alpha_i = \mathit{true}$ and $\beta_i = \gamma_i = \mathit{false}$. We construct an abstract execution $e_{\boldsymbol{r}}$ in which the happens-before relation sequences the operations of replica r_i before those of $r_i + 1 \bmod 3$, and in turn before $r_i + 2 \bmod 3$. In other words, the replicas in round i appear to execute in left-to-right order from starting with the replica r_i, whose reads correspond to the satisfying assignment of $(\alpha_i \vee \beta_i \vee \gamma_i)$. The read-from relation of $e_{\boldsymbol{r}}$ relates each $\mathrm{Read}(x_j) = \mathit{true}$ operation to the most recent $\mathrm{Enable}(x_j)$ operation in happens-before order, which is unique since happens-before sequences the operations of all rounds; the case for $\mathrm{Read}(x_j) = \mathit{false}$ and $\mathrm{Disable}(x_j)$ is symmetric. It is then straightforward to verify that $e_{\boldsymbol{r}}$ satisfies the axioms of the enable- or disable-wins flag, and thus h_p is admitted.

In the "only if" direction, let e be an abstract execution of h_p, and let $\boldsymbol{r} \in \{0,1,2\}^m$ be the replicas first to execute in each round according to the happens-before order of e. It is straightforward to verify that the assignment in which a given variable is set to true iff the replica encoding its positive assignment in some clause executes first in its round, i.e.,

$$x_j = \begin{cases} \mathit{true} & \text{if } \exists i.(r_i = 0 \wedge \alpha_i = x_j) \vee (r_i = 1 \wedge \beta_i = x_j) \vee (r_i = 2 \wedge \gamma_i = x_j) \\ \mathit{false} & \text{otherwise,} \end{cases}$$

is a satisfying assignment to p. □

Theorem 1 establishes intractability of consistency for the aforementioned sets, flags, and registers, independently from the number of replicas. In contrast, our proof of Theorem 2 for counter data types depends on the number of replicas, since our encoding requires two replicas per propositional variable. Intuitively, since counter increments and decrements are commutative, the initial round in the previous encoding would have fixed all counter values to zero. Instead, the next encoding isolates initial increments and decrements to independent replicas. The weaker result is indeed tight since checking counter consistency with a fixed number of replicas is polynomial time, as Sect. 5 demonstrates.

Theorem 2. *The admissibility problem for the Counter data type is NP-hard.*

Proof. We demonstrate a reduction from the 1-in-3 SAT problem. For a given problem $p = \bigwedge_{i=1}^{m}(\alpha_i \vee \beta_i \vee \gamma_i)$ over variables $x_1, \ldots, x_n$, we construct a history h_p of the counter data type over $2n + 3$ replicas, as illustrated in Fig. 5.

Besides the differences imposed due to the commutativity of counter increments and decrements, our reduction follows the same strategy as in the proof of

Theorem 1: the happens-before relation of h_p's abstract executions order every pair of operations in distinct rounds (of Replicas 0–2), and every operation in a given (non-initial) round. As before, Replicas 0–2 appear to execute atomically per round, in a round-robin fashion, and counter variables are consistent across rounds. The key difference is that here abstract executions' happens-before relations only relate the operations of either Replica $2j+1$ or $2j+2$, for each $j = 1, \ldots, n$, to operations in subsequent rounds: the other's operations are never observed by other replicas. Our encoding ensures that exactly one of each is observed by ensuring that the counter y is incremented exactly n times — and relying on the fact that every variable appears in some clause, so that a read that observed neither or both would yield the value zero, which is inconsistent with h_p. Otherwise, our reasoning follows the proof of Theorem 1, in which the read-from relation selects all increments and decrements of the same counter variable in happens-before order. □

4 Polynomial-Time Algorithms for Registers and Arrays

We show that the problem of checking consistency is polynomial time for RGA, and even for LWW and MVR under the assumption that each value is written at most once, i.e., for each value v, the input history contains at most one write operation $\mathsf{write}(x,v)$. Histories satisfying this assumption are called *differentiated*. The latter is a restriction motivated by the fact that practical implementations of these datatypes are data-independent [38], i.e., their behavior doesn't depend on the concrete values read or written and any potential buggy behavior can be exposed in executions where each value is written at most once. Also, in a testing environment, this restriction can be enforced by tagging each value with a replica identifier and a sequence number.

In all three cases, the feature that enables polynomial time consistency checking is the fact that the read-from relation becomes fixed for a given history, i.e., if the history is consistent, then there exists exactly one read-from relation rf that satisfies the ReadFrom_ and Retval_ axioms, and rf can be derived syntactically from the operation labels (using those axioms). Then, our axiomatic characterizations enable a consistency checking algorithm which roughly, consists in instantiating those axioms in order to compute an abstract execution.

The consistency checking algorithm for RGA, LWW, and MVR is listed in Algorithm 1. It computes the three relations rf, hb, and lin of an abstract execution using the datatype's axioms. The history is declared consistent iff there exist satisfying rf and hb relations, and the relations hb and lin computed this way are acyclic. The acyclicity requirement comes from the definition of abstract executions where hb and lin are required to be partial/total orders. While an abstract execution would require that lin is a total order, this algorithm computes a partial linearization order. However, any total order compatible with this partial linearization would satisfy the axioms of the datatype.

ComputeRF computes the read-from relation rf satisfying the ReadFrom_ and Retval_ axioms. In the case of LWW and MVR, it defines rf as the set

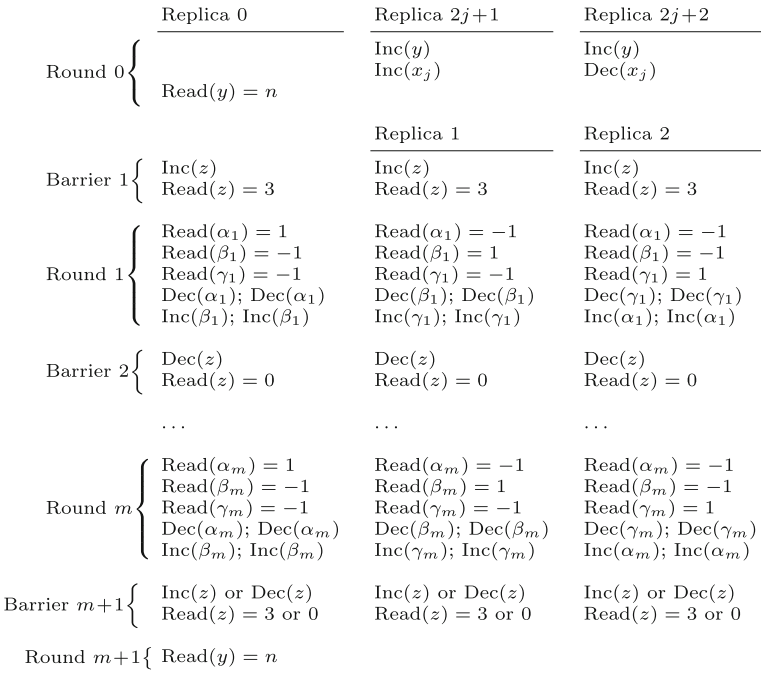

Fig. 5. The encoding of a 1-in-3 SAT problem $\bigwedge_{i=1}^{m}(\alpha_i \vee \beta_i \vee \gamma_i)$ over variables $x_1, \ldots, x_n$ as the history of a counter over $2n+3$ replicas. Besides the counter variables x_j encoding propositional variables x_j, the encoding adds a variable y encoding the number of initial increments and decrements, and a variable z to implement synchronization barriers.

of all pairs formed of $\mathsf{write}(x,v)$ and $\mathsf{read}(x)$ operations where v belongs to the return value of the read. By RETVAL_, each $\mathsf{read}(x)$ operation must be associated to at least one $\mathsf{write}(x,\_)$ operation. Also, the fact that each value is written at most once implies that this rf relation is uniquely defined, e.g., for LWW, it is not possible to find two write operations that could be rf related to the same read operation. In general, if there exists no rf relation satisfying these axioms, then $\mathsf{ComputeRF}$ returns a distinguished value $\perp$ to signal a consistency violation. Note that the computation of the read-from for LWW and MVR is quadratic time[4] since the constraints imposed by the axioms relate only to the operation labels, the methods they invoke or their arguments. The case of RGA is slightly more involved because the axiom RETVALRGA introduces more read-from constraints based on the happens-before order which includes ro and the rf itself. In this case, the computation of rf relies on a fixpoint computation, which converges in at most quadratic time (the maximal size of rf), described in Algorithm 2. Essentially, we use the axiom READFROMRGA to populate the

[4] Assuming constant time lookup/insert operations (e.g., using hashmaps), this complexity is linear time.

Input: A differentiated history $h = \langle \mathsf{Op}, \mathsf{ro} \rangle$ and a datatype T.
Output: *true* iff h satisfies the axioms of T.

```
1 rf ← ComputeRF(h,ReadFrom[T],Retval[T] );
2 if rf = ⊥ then return false;
3 hb ← (ro ∪ rf)^+;
4 if hb is cyclic or ⟨h, rf, hb⟩ ⊭ ReadFromMaximal[T] ∧ ReadAllMaximals[T]
  then
5 |   return false;
6 lin ← hb;
7 lin ← LinClosure(hb,Lin[T]);
8 if lin is cyclic then return false;
9 return true;
```

Algorithm 1. Consistency checking for RGA, LWW, and MVR. Re. . . [T] refers to an axiom of T, or *true* when T lacks such an axiom. The relation R^+ denotes the transitive closure of R.

read-from relation and then, apply the axiom RetvalRGA iteratively, using the read-from constraints added in previous steps, until the computation converges.

After computing the read-from relation, our algorithm defines the happens-before relation hb as the transitive closure of ro union rf. This is sound because none of the axioms of these datatypes enforce new happens-before constraints, which are not already captured by ro and rf. Then, it checks whether the hb defined this way is acyclic and satisfies the datatype's axioms that constrain hb, i.e., ReadFromMaximal and ReadAllMaximals (when they are present).

Finally, in the case of LWW and RGA, the algorithm computes a (partial) linearization order that satisfies the corresponding Lin_ axioms. Starting from an initial linearization order which is exactly the happens-before, it computes new constraints by instantiating the universally quantified axioms LinLWW and LinRGA. Since these axioms are not "recursive", i.e., they don't enforce linearization order constraints based on other linearization order constraints, a standard instantiation of these axioms is enough to compute a partial linearization order such that any extension to a total order satisfies the datatype's axioms.

Theorem 3. *Algorithm 1 returns true iff the input history is consistent.*

The following holds because Algorithm 1 runs in polynomial time — the rank depends on the number of quantifiers in the datatype's axioms. Indeed, Algorithm 1 represents a least fixpoint computation which converges in at most a quadratic number of iterations (the maximal size of rf).

Corollary 1. *The admissibility problem is polynomial time for RGA, and for LWW and MVR on differentiated histories.*

Input: A history $h = \langle \mathsf{Op}, \mathsf{ro} \rangle$ of RGA.
Output: An rf satisfying ReadFromRGA $\wedge$ RetvalRGA, if exists; $\perp$ o/w

$\mathsf{rf} \leftarrow \{(o_1, o_2) : \mathsf{meth}(o_1) = \mathsf{addAfter}, \mathsf{meth}(o_2) \in \{\mathsf{addAfter}, \mathsf{remove}, \mathsf{read}\}, \mathsf{arg}_2(o_1) = \mathsf{arg}_1(o_2) \vee \mathsf{arg}_2(o_1) \in \mathsf{ret}(o_2)\}$;
if $\langle h, \mathsf{rf} \rangle \not\models$ ReadFromRGA **then return** $\perp$;
while *true* **do**
  $\mathsf{rf}_1 \leftarrow \emptyset$;
  foreach $o_1, o_2 \in \mathsf{Op}$ s.t. $\langle o_2, o_1 \rangle \in (\mathsf{rf} \cup \mathsf{ro})^+$ and $\mathsf{meth}(o_1) = \mathsf{read}$ and $\mathsf{meth}(o_2) = \mathsf{addAfter}$ and $\mathsf{arg}_2(o_2) \notin \mathsf{ret}(o_1)$ **do**
    if $\exists o_3 \in \mathsf{Op}$ s.t. $\mathsf{meth}(o_3) = \mathsf{remove}$ and $\mathsf{arg}(o_3) = \mathsf{arg}_2(o_2)$ **then**
      $\mathsf{rf}_1 \leftarrow \mathsf{rf}_1 \cup \{\langle o_3, o_1 \rangle\}$;
    else
      return $\perp$;
  if $\mathsf{rf}_1 \subseteq \mathsf{rf}$ **then break**;
  else $\mathsf{rf} \leftarrow \mathsf{rf} \cup \mathsf{rf}_1$;
return rf;

Algorithm 2. The procedure ComputeRF for RGA.

5 Polynomial-Time Algorithms for Replicated Counters

In this section, we show that checking consistency for the replicated counter datatype becomes polynomial time assuming the number of replicas in the input history is fixed (i.e., the width of the replica order ro is fixed). We present an algorithm which constructs a valid happens-before order (note that the semantics of the replicated counter doesn't constrain the linearization order) incrementally, following the replica order. At any time, the happens-before order is uniquely determined by a *prefix mapping* that associates to each replica a *prefix* of the history, i.e., a set of operations which is downward-closed w.r.t. replica order (i.e., if it contains an operation it contains all its ro predecessors). This models the fact that the replica order is included in the happens-before and therefore, if an operation o_1 happens-before another operation o_2, then all the ro predecessors of o_1 happen-before o_2. The happens-before order can be extended in two ways: (1) adding an operation issued on the replica i to the prefix of replica i, or (2) "merging" the prefix of a replica j to the prefix of a replica i (this models the delivery of an operation issued on replica j and all its happens-before predecessors to the replica i). Verifying that an extension of the happens-before is valid, i.e., that the return values of newly-added read operations satisfy the RetvalCounter axiom, doesn't depend on the happens-before order between the operations in the prefix associated to some replica (it is enough to count the inc and dec operations in that prefix). Therefore, the algorithm can be seen as a search in the space of prefix mappings. If the number of replicas in the input history is fixed, then the number of possible prefix mappings is polynomial in the size of the history, which implies that the search can be done in polynomial time.

Let $h = (\mathsf{Op}, \mathsf{ro})$ be a history. To simplify the notations, we assume that the replica order is a union of sequences, each sequence representing the operations

Input: History $h = (\mathsf{Op}, \mathsf{ro})$, prefix map m, and set *seen* of invalid prefix maps
Output: *true* iff there exists read-from and happens-before relations rf and hb such that $m \subseteq \mathsf{hb}$, and $\langle h, \mathsf{rf}, \mathsf{hb}\rangle$ satisfies the counter axioms.

```
1  if m is complete then return true;
2  foreach replica i do
3  |  foreach replica j ≠ i do
4  |  |  m' ← m[i ← m(i) ∪ m(j)];
5  |  |  if m' ∉ seen and checkCounter(h, m', seen) then
6  |  |  |  return true;
7  |  |  seen ← seen ∪ {m'};
8  |  if ∃o1. ro^1(last_i(m), o1) then
9  |  |  if meth(o1) = read and
   |  |  arg(o1) = x ∧ ret(o1) ≠ |{o ∈ m[i] | o = inc(x)}| − |{o ∈ m[i] | o = dec(x)}|
   |  |  then
10 |  |  |  return false;
11 |  |  m' ← m[i ← m(i) ∪ {o1}];
12 |  |  if m' ∉ seen and checkCounter(h, m', seen) then
13 |  |  |  return true;
14 |  |  seen ← seen ∪ {m'};
15 return false;
```

Algorithm 3. The procedure checkCounter, where ro^1 denotes immediate ro-successor, and $f[a \leftarrow b]$ updates function f with mapping $a \mapsto b$.

issued on the same replica. Therefore, each operation $o \in \mathsf{Op}$ is associated with a replica identifier $\mathsf{rep}(o) \in [1..n_h]$, where n_h is the number of replicas in h.

A *prefix* of h is a set of operation $\mathsf{Op}' \subseteq \mathsf{Op}$ such that all the ro predecessors of operations in Op' are also in Op', i.e., $\forall o \in \mathsf{Op}.\ \mathsf{ro}^{-1}(o) \in \mathsf{Op}$. Note that the union of two prefixes of h is also a prefix of h. The *last operation* of replica i in a prefix Op' is the ro-maximal operation o with $\mathsf{rep}(o) = i$ included in Op'. A prefix Op' is called *valid* if $(\mathsf{Op}', \mathsf{ro}')$, where ro' is the projection of ro on Op', is admitted by the replicated counter.

A *prefix map* is a mapping m which associates a prefix of h to each replica $i \in [1..n_h]$. Intuitively, a prefix map defines for each replica i the set of operations which are "known" to i, i.e., happen-before the last operation of i in its prefix. Formally, a prefix map m is *included* in a happens-before relation hb, denoted by $m \subseteq \mathsf{hb}$, if for each replica $i \in [1..n_h]$, $\mathsf{hb}(o, o_i)$ for each operation in $o \in m(i) \setminus \{o_i\}$, where o_i is the last operation of i in $m(i)$. We call o_i the *last operation* of i in m, and denoted it by $\mathsf{last}_i(m)$. A prefix map m is *valid* if it associates a valid prefix to each replica, and *complete* if it associates the whole history h to each replica i.

Algorithm 3 lists our algorithm for checking consistency of replicated counter histories. It is defined as a recursive procedure checkCounter that searches for a sequence of valid extensions of a given prefix map (initially, this prefix map is empty) until it becomes complete. The axiom RetvalCounter is enforced whenever extending the prefix map with a new read operation (when the last

operation of a replica i is "advanced" to a read operation). The following theorem states of the correctness of the algorithm.

Theorem 4. checkCounter$(h, \emptyset, \emptyset)$ *returns true iff the input history is consistent.*

When the number of replicas is fixed, the number of prefix maps becomes polynomial in the size of the history. This follows from the fact that prefixes are uniquely defined by their ro-maximal operations, whose number is fixed.

Corollary 2. *The admissibility problem for replicated counters is polynomial-time when the number of replicas is fixed.*

6 Polynomial-Time Algorithms for Sets and Flags

While Theorem 1 shows that the admissibility problem is NP-complete for replicated sets and flags even if the number of replicas is fixed, we show that this problem becomes polynomial time when additionally, the number of values added to the set, or the number of flags, is also fixed. Note that this doesn't limit the number of operations in the input history which can still be arbitrarily large. In the following, we focus on the Add-Wins Set, the other cases being very similar.

We propose an algorithm for checking consistency which is actually an extension of the one presented in Sect. 5 for replicated counters. The additional complexity in checking consistency for the Add-Wins Set comes from the validity of contains(x) return values which requires identifying the maximal predecessors in the happens-before relation that add or remove x (which are not necessarily the maximal hb-predecessors all-together). In the case of counters, it was enough just to count happens-before predecessors. Therefore, we extend the algorithm for replicated counters such that along with the prefix map, we also keep track of the hb-maximal add(x) and remove(x) operations for each element x and each replica i. When extending a prefix map with a contains operation, these hb-maximal operations (which define a witness for the read-from relation) are enough to verify the RetValSet axiom. Extending the prefix of a replica with an add or remove operation (issued on the same replica), or by merging the prefix of another replica, may require an update of these hb-maximal predecessors.

When the number of replicas and elements are fixed, the number of read-from maps is polynomial in the size of the history — recall that the number of operations associated by a read-from map to a replica and set element is bounded by the number of replicas. Combined with the number of prefix maps being polynomial when the number of replicas is fixed, we obtain the following result.

Theorem 5. *Checking whether a history is admitted by the Add-Wins Set, Remove-Wins Set, Enable-Wins Flag, or the Disable-Wins Flag is polynomial time provided that the number of replicas and elements/flags is fixed.*

7 Related Work

Many have considered consistency models applicable to CRDTs, including causal consistency [26], sequential consistency [27], linearizability [24], session consistency [35], eventual consistency [36], and happens-before consistency [29]. Burckhardt et al. [8,11] propose a unifying framework to formalize these models. Many have also studied the complexity of verifying data-type agnostic notions of consistency, including serializability, sequential consistency and linearizability [1,2,4,18,20,22,30], as well as causal consistency [6]. Our definition of the replicated LWW register corresponds to the notion of causal convergence in [6]. This work studies the complexity of the admissibility problem for the replicated LWW register. It shows that this problem is NP-complete in general and polynomial time when each value is written only once. Our NP-completeness result is stronger since it assumes a fixed number of replicas, and our algorithm for the case of unique values is more general and can be applied uniformly to MVR and RGA. While Bouajjani et al. [5,14] consider the complexity for individual linearizable collection types, we are the first to establish (in)tractability of individual replicated data types. Others have developed effective consistency checking algorithms for sequential consistency [3,9,23,31], serializability [12,17,18,21], linearizability [10,16,28,37], and even weaker notions like eventual consistency [7] and sequential happens-before consistency [13,15]. In contrast, we are the first to establish precise polynomial-time algorithms for runtime verification of replicated data types.

8 Conclusion

By developing novel logical characterizations of replicated data types, reductions from propositional satisfiability checking, and tractable algorithms, we have established a frontier of tractability for checking consistency of replicated data types. As far as we are aware, our results are the first to characterize the asymptotic complexity consistency checking for CRDTs.

References

1. Alur, R., McMillan, K.L., Peled, D.A.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. **160**(1–2), 167–188 (2000). https://doi.org/10.1006/inco.1999.2847
2. Bingham, J.D., Condon, A., Hu, A.J.: Toward a decidable notion of sequential consistency. In: Rosenberg, A.L., auf der Heide, F.M. (eds.) SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, (part of FCRC 2003), 7–9 June 2003, pp. 304–313. ACM (2003). https://doi.org/10.1145/777412.777467
3. Bingham, J., Condon, A., Hu, A.J., Qadeer, S., Zhang, Z.: Automatic verification of sequential consistency for unbounded addresses and data values. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 427–439. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_33

4. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 290–309. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_17
5. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: On reducing linearizability to state reachability. Inf. Comput. **261**(Part), 383–400 (2018). https://doi.org/10.1016/j.ic.2018.02.014
6. Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, 18–20 January 2017, pp. 626–638. ACM (2017). http://dl.acm.org/citation.cfm?id=3009888
7. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 285–296. ACM (2014). https://doi.org/10.1145/2535838.2535877
8. Burckhardt, S.: Principles of eventual consistency. Found. Trends Program. Lang. **1**(1–2), 1–150 (2014). https://doi.org/10.1561/2500000011
9. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, 10–13 June 2007, pp. 12–21. ACM (2007). https://doi.org/10.1145/1250734.1250737
10. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, 5–10 June 2010, pp. 330–340. ACM (2010). https://doi.org/10.1145/1806596.1806634
11. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 271–284. ACM (2014). https://doi.org/10.1145/2535838.2535848
12. Cohen, A., O'Leary, J.W., Pnueli, A., Tuttle, M.R., Zuck, L.D.: Verifying correctness of transactional memories. In: Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2007, Austin, Texas, USA, 11–14 November 2007, pp. 37–44. IEEE Computer Society (2007). https://doi.org/10.1109/FAMCAD.2007.40
13. Emmi, M., Enea, C.: Monitoring weak consistency. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018, Part I. LNCS, vol. 10981, pp. 487–506. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_26
14. Emmi, M., Enea, C.: Sound, complete, and tractable linearizability monitoring for concurrent collections. PACMPL **2**(POPL), 25:1–25:27 (2018). https://doi.org/10.1145/3158113
15. Emmi, M., Enea, C.: Weak-consistency specification via visibility relaxation. PACMPL **3**(POPL), 60:1–60:28 (2019). https://dl.acm.org/citation.cfm?id=3290373

16. Emmi, M., Enea, C., Hamza, J.: Monitoring refinement via symbolic reasoning. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 260–269. ACM (2015). https://doi.org/10.1145/2737924.2737983
17. Emmi, M., Majumdar, R., Manevich, R.: Parameterized verification of transactional memories. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, 5–10 June 2010, pp. 134–145. ACM (2010). https://doi.org/10.1145/1806596.1806613
18. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_8
19. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York (1979)
20. Gibbons, P.B., Korach, E.: Testing shared memories. SIAM J. Comput. **26**(4), 1208–1244 (1997). https://doi.org/10.1137/S0097539794279614
21. Guerraoui, R., Henzinger, T.A., Jobstmann, B., Singh, V.: Model checking transactional memories. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June 2008, pp. 372–382. ACM (2008). https://doi.org/10.1145/1375581.1375626
22. Hamza, J.: On the complexity of linearizability. In: Bouajjani, A., Fauconnier, H. (eds.) NETYS 2015. LNCS, vol. 9466, pp. 308–321. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26850-7_21
23. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Verifying sequential consistency on shared-memory multiprocessor systems. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 301–315. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_27
24. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
25. Kingsbury, K.: Jepsen: Distributed systems safety research (2016). https://jepsen.io
26. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563
27. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439
28. Lowe, G.: Testing for linearizability. Concurr. Comput. Pract. Exp. **29**(4) (2017). https://doi.org/10.1002/cpe.3928
29. Manson, J., Pugh, W., Adve, S.V.: The java memory model. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, 12–14 January 2005, pp. 378–391. ACM (2005). https://doi.org/10.1145/1040305.1040336
30. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979). https://doi.org/10.1145/322154.322158
31. Qadeer, S.: Verifying sequential consistency on shared-memory multiprocessorsby model checking. IEEE Trans. Parallel Distrib. Syst. **14**(8), 730–741 (2003). https://doi.org/10.1109/TPDS.2003.1225053

32. Roh, H., Jeon, M., Kim, J., Lee, J.: Replicated abstract data types: building blocks for collaborative applications. J. Parallel Distrib. Comput. **71**(3), 354–368 (2011). https://doi.org/10.1016/j.jpdc.2010.12.006
33. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 386–400. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24550-3_29
34. Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Convergent and commutative replicated data types. Bull. EATCS **104**, 67–88 (2011). http://eatcs.org/beatcs/index.php/beatcs/article/view/120
35. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M., Theimer, M., Welch, B.B.: Session guarantees for weakly consistent replicated data. In: Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 1994), Austin, Texas, USA, 28–30 September 1994, pp. 140–149. IEEE Computer Society (1994). https://doi.org/10.1109/PDIS.1994.331722
36. Terry, D.B., Theimer, M., Petersen, K., Demers, A.J., Spreitzer, M., Hauser, C.: Managing update conflicts in bayou, a weakly connected replicated storage system. In: Jones, M.B. (ed.) Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, 3–6 December 1995, pp. 172–183. ACM (1995). https://doi.org/10.1145/224056.224070
37. Wing, J.M., Gong, C.: Testing and verifying concurrent objects. J. Parallel Distrib. Comput. **17**(1–2), 164–182 (1993). https://doi.org/10.1006/jpdc.1993.1015
38. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic. In: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986, pp. 184–193. ACM Press (1986). https://doi.org/10.1145/512644.512661