ReDunT: Automatically Deriving Redundancy Relations for Pure Op-Based CRDTs

Dina Borrego*
Universidade NOVA de Lisboa
Vrije Universiteit Brussel
Portugal, Belgium
d.borrego@campus.fct.unl.pt

Afonso Vilalonga*
Universidade NOVA de Lisboa
Portugal
j.vilalonga@campus.fct.unl.pt

Henrique Domingos
Universidade NOVA de Lisboa
Portugal
hj@fct.unl.pt

Nuno Preguiça
Universidade NOVA de Lisboa
Portugal
nuno.preguica@fct.unl.pt

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Belgium
egonzale@vub.be

Carla Ferreira
Universidade NOVA de Lisboa
Portugal
carla.ferreira@fct.unl.pt

Abstract

Conflict-free Replicated Data Types (CRDTs) are a family of data structures that incorporate conflict resolution mechanisms to ensure state convergence and ease the development of highly available distributed systems. Pure operation-based CRDTs provide a unified framework for defining CRDTs based on a partially ordered log of operations. The framework eases the implementation of CRDTs and reduces the risk of introducing bugs. However, developers are still required to manually define the CRDT semantics and compaction functions, which can be complex and error-prone. This paper proposes ReDunT, a framework for the automatic synthesis of compaction functions for pure op-based CRDTs using a rewrite-based semantic approach. We evaluate ReDunT by applying it to a portfolio of pure operation-based CRDTs, including sets and flags, and comparing the obtained results with the manually optimised specifications. We also demonstrate the feasibility of the framework by implementing it as an automatic tool in K semantic framework.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; • **Computing methodologies** → **Distributed computing methodologies**.

Keywords: CRDTs, program synthesis, optimisations

ACM Reference Format:
Dina Borrego, Afonso Vilalonga, Henrique Domingos, Nuno Preguiça, Elisa Gonzalez Boix, and Carla Ferreira. 2025. ReDunT: Automatically Deriving Redundancy Relations for Pure Op-Based CRDTs. In *12th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3721473.3722145

1 Introduction

Modern distributed applications require high availability, low latency, fault tolerance, and scalability. However, as the CAP theorem [3] demonstrates, systems operating under unavoidable network partitions face an inherent trade-off between consistency and availability. Highly available systems weaken consistency, allowing operations to execute independently and propagate asynchronously to other replicas. While this can lead to temporary divergence, a mechanism is needed to ensure eventual convergence, guaranteeing that all replicas reach a consistent state.

Replicated Data Types (RDTs) [1, 4, 5, 7, 11] have emerged as a fundamental abstraction for simplifying the development of highly available distributed systems. They enable replicas to converge without coordination, eliminating costly synchronisation and making them well-suited for scalable systems such as collaborative editing, edge computing, etc.

While RDTs are a promising technique for building highly scalable systems, designing new RDTs remains challenging. This challenge stems from the need to reason about all possible orders of concurrent operations to ensure that conflicts are correctly detected and resolved. Prior research has investigated automatic techniques for simplifying RDT construction, including methods to synthesise Conflict-free Replicated Data Types (CRDTs) [7, 8], and derive RDTs from sequential data types [5, 6, 9]. These approaches have made substantial progress in formalising correctness conditions to enable automatic synthesis. However, their primary focus has been on ensuring correctness rather than producing optimised designs. On the other hand, pure operation-based CRDTs [1] provide a principled and practical approach to designing optimised CRDTs [11]. This approach structures CRDTs around a Partially Ordered Log (PO-Log) of operations, ensuring efficient replication and conflict resolution. The approach exposes causal information from the underlying broadcast middleware that can be leveraged to minimise resource usage through redundancy relations and causal stability. However, developers must still manually encode the CRDT semantics and the compaction functions, which can be complex and error-prone.

In this paper, we propose ReDunT, a rewrite-based framework that, given a non-optimised specification of a pure

*Both authors contributed equally to this research.

PaPoC '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1558-7/2025/03
https://doi.org/10.1145/3721473.3722145

Algorithm 1 (Simplified) distributed algorithm for replica i showing the interaction between the RCB middleware and the pure op-based CRDT framework (from [2]).

state: $s_i := \emptyset$
on operation$_i(o)$
  broadcast$_i(o)$
on deliver$_i(t, o)$
  $s_i := s_i \setminus \{(t', o') \mid (t', o') \in s_i \cdot (t', o')\ \mathsf{R\_}\ (t, o)\}$
  $\cup\ \{(t, o) \mid (t, o)\ \not\mathsf{R}\ s_i\}$

operation-based CRDT, automatically synthesises an equivalent optimised version of the CRDT. By automating this process, our framework simplifies CRDT design while ensuring correctness and optimised storage performance. We applied ReDunT to the portfolio of non-commutative pure op-based CRDTs from [1], showcasing its ability to derive automatically the optimised versions of the CRDTs. We did not apply ReDunT to commutative data types, as they can be implemented without a log. Additionally, we present a prototype implementation of ReDunT using K [10], a rewrite-based executable semantic framework.

We compared our results with manually optimised implementations from [1], finding that for some CRDTs, we obtained an equivalent specification. In contrast, for others, ReDunT produced a less optimised version due to its inability to leverage operations semantics knowledge encoded in manual optimisations. While still a work in progress, ReDunT provides an initial foundation for automatically transforming non-optimised CRDTs into their optimised counterparts.

2 Background

In this section, we provide the necessary background on the pure operation-based (op-based) CRDT framework [1] to understand the contributions of this work. The pure op-based CRDT framework supports the design of CRDTs by using a partially ordered log of operations (PO-Log) constructed using causality information. The framework is built on top of Reliable Causal Broadcast (RCB) middleware, which ensures causal delivery and automatically tags each operation with causality metadata, eliminating the need for manual encoding of this metadata in the CRDT definition.

Algorithm 1 outlines the interaction between the RCB middleware and the pure op-based CRDT framework. Each replica maintains a PO-Log, denoted as s_i, which is initially empty. When a user invokes an operation o on replica i, the operation$_i$ event on that replica is triggered, broadcasting the operation to all replicas via the RCB middleware, which tags a timestamp to the operation. Upon delivery of an operation o tagged with timestamp t to replica i, the deliver$_i$ event is triggered, integrating the operation into the log if needed.

The framework introduces the concepts of *causal redundancy* to keep the log compact and *causal stabilisation* to

$$\mathsf{eval}_i(\mathsf{rd}, s) \quad = \quad \{v \mid (t, [\mathsf{wr}, v]) \in s \wedge \forall (t', \_) \in s \cdot t \not< t'\}$$

Figure 1. MVRegister Concurrent Semantics (from [1]).

$$(t, o)\ \mathsf{R}\ s \iff o[0] = \mathsf{clear}$$
$$(t', o')\ \mathsf{R\_}\ (t, o) \iff t' < t$$
$$\mathsf{eval}_i(\mathsf{rd}, s) \quad = \quad \{v \mid (\_, [\mathsf{wr}, v]) \in s\}$$

Figure 2. (Simplified) Compact MVRegister CRDT (from [1])

trim causal information from the log entries once all replicas have observed an operation. Our work focuses on deriving the compacting relations for causal redundancy.

The idea of causal redundancy is to compact the log by removing operations whose effects do not affect the state of the data type (and thus do not impact the output of query operations). Specifically, the framework introduces two binary relations, R and R_, defining the conditions that make operations causal redundant. R defines whether to store a newly arriving operation in the log, and R_ defines whether an arriving operation renders existing log entries redundant. A concrete CRDT implementation built on the framework must define these relations to ensure correct PO-Log compaction.

We now discuss the implementation of a Multi-Value Register (MVRegister) pure op-based CRDT. We assume two update operations: wr, which writes a value to the register, and clear, which removes all values from the set. In the pure op-based framework, the concurrency semantics of non-commutative data types, such as the MVRegister, are determined by the output of query operations, considering all operations kept in the log. The eval query function takes a query and the current log state as input, returning a result based on the stored operations. Figure 1 shows the query operation for the MVRegister, which states that a read (rd) operation returns a set of values v for which the corresponding write (wr) operations are not causally succeeded by any other wr or clear operation.

Figure 2 presents the compact MVRegister CRDT design from [1]. The R relation defines clear operations as redundant, meaning they are not added to the PO-Log, as they only affect causally precedent operations that have already been delivered. The R_ relation defines that an arriving operation o renders a stored operation o' redundant if and only if o' causally precedes o. This means that wr operations are removed from the PO-Log if a later wr or clear operation exists in their causal future. As a result, the PO-Log maintains the concurrent wr operations. Finally, the rd operation returns all values with wr operations in the PO-Log. While the R and R_ relations in figure 2 were manually defined, this work aims to derive them automatically from the concurrency semantics in figure 1.

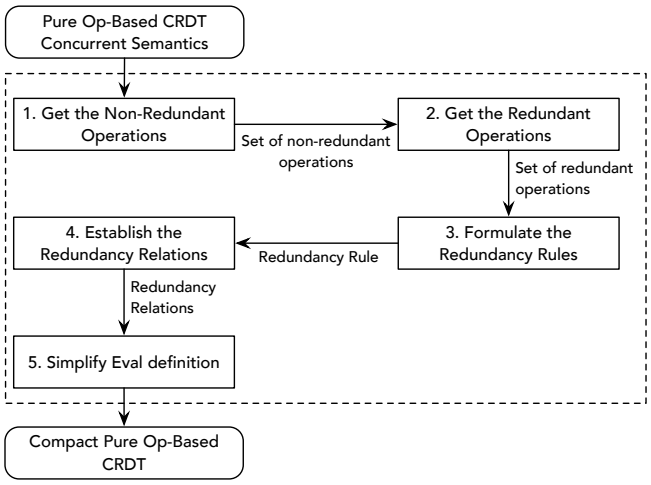

Figure 3. The ReDunT workflow.

3 The ReDunT Approach

We aim to derive a compact pure op-based CDRT from its non-optimised design. To this end, we introduce ReDunT, a five-step methodology that automatically synthesises the redundancy relations of a pure op-based CRDT given its concurrent semantics expressed in the eval function.

Figure 3 illustrates ReDunT's five steps. The process starts by identifying non-redundant operations—those needed for state evaluation. Next, ReDunT identifies the redundant operations—those not impacting the CRDT's state and that can be removed from the PO-Log without loss of information. It then defines the redundancy rules, specifying the conditions under which an operation is considered redundant. These rules form the basis for establishing redundancy relations, which determine how operations are added or removed from the log. Finally, ReDunT simplifies the eval function based on the derived relations. We now detail each step using the MVRegister CRDT discussed in section 2.

1. Get the non-redundant operations. Generating redundancy relations begins by identifying the set of non-redundant operations. In this step, ReDunT analyses the eval definition of a given pure op-based CRDT to extract the operations that influence state computation. In a nutshell, this step converts the eval definition into its corresponding set of operations. These operations, referred to as non-redundant operations, form the minimal set necessary to reconstruct the state while maintaining correctness. Starting from figure 1, ReDunT identifies that, for an MVRegister CRDT, the non-redundant operations are the write operations not causally succeeded by any other operation, i.e., $\{(t, [\mathsf{wr}, v]) \in s \mid \forall (t', \_) \in s \cdot t \not\prec t'\}$.

2. Get the redundant operations. Starting from the set of non-redundant operations, ReDunT derives the set of redundant operations, which consists of the operations that do not affect the CRDT state and can, therefore, be safely removed from the PO-Log. From a purely mathematical perspective, the set of redundant operations corresponds to the complement of the set of non-redundant operations. However, the complement may classify as redundant certain operations that, while not directly contributing to state computation, play an indirect role in defining the CRDT's behaviour—particularly in specifying its semantics. We discuss this issue further, with an example, in section 4. To prevent the incorrect removal of non-redundant operations, ReDunT derives the redundant operations in two steps. First, it computes the complement of the non-redundant operations. Then, it filters out operations required for defining the CRDT's concurrency semantics and conflict resolution. For example, in the MVRegister CRDT, the complement of the non-redundant operation is $\{(t, [\mathsf{wr}, v]) \in s \mid \exists (t', \_) \in s \cdot t \prec t'\} \cup \{(t, [\mathsf{clear}]) \in s\}$. In this case, no operations need to be filtered out, as both writes and clears only make preceding operations redundant. However, if the eval function specified that the clear operation (for example) also rendered concurrent writes redundant, then $\{(t, [\mathsf{clear}]) \in s\}$ would need to be filtered from the complement, as it would define concurrency semantics.

3. Formulate the redundancy rules. A redundancy rule captures the conditions under which an operation becomes redundant due to the presence of another one. Unlike redundancy relations, which determine whether a specific operation—either an existing entry in the log or a newly received one—is redundant, our redundancy rule establishes the general criteria for redundancy, serving as the foundation for deriving these relations. A redundancy rule evaluates two tagged operations, (o_r, t_r) and (o, t). The first, (o_r, t_r), represents the operation rendered redundant, while the second, (o, t), represents the operation responsible for causing the redundancy. Each redundancy rule consists of a set of conditions that define when o_r is redundant. These conditions are expressed as logical conjunctions, ensuring that all specified criteria must hold for the redundancy to apply.

In this step, ReDunT derives redundancy rules from the set of redundant operations. This set defines the conditions under which an operation becomes redundant and may consist of multiple subsets, as shown above. Each subset represents a specific redundancy condition, leading to the formulation of a redundancy rule. For the MVRegister, ReDunT derives the following two rules from the identified redundant operations: i) $o_r[0] = \mathsf{wr} \wedge o[0] = \_ \wedge t_r \prec t$, and ii) $o_r[0] = \mathsf{clear}$.

4. Establish the redundancy relations. ReDunT derives the two redundancy relations by applying symbol substitutions to the redundancy rules. In both R and R_ relations, (o, t) represents the newly delivered operation, while (o', t') corresponds to an existing operation in the log. The R relation defines the conditions under which a newly received operation is made redundant. Therefore, ReDunT derives this relation by replacing (o_r, t_r) and (o, t) in the redundancy

rules with (o, t) and (o', t'), respectively. Conversely, the R_ relation defines the conditions under which an operation in the log is rendered redundant. Thus, to define the R_ relation, ReDunT applies the opposite substitutions: (o_r, t_r) and (o, t) are replaced with (o', t') and (o, t), respectively.

This step processes the redundancy rules iteratively, applying symbol substitutions based on the relation:

$$\begin{aligned}(t,o)\ \mathsf{R}\ (t',o') &\iff (o[0] = \mathsf{wr}) \wedge o'[0] = \_ \wedge t \prec t') \vee \\ &\quad (o[0] = \mathsf{clear}) \\ (t,o)\ \mathsf{R\_}\ (t',o') &\iff (o'[0] = \mathsf{wr} \wedge o[0] = \_ \wedge t' \prec t) \vee \\ &\quad (o'[0] = \mathsf{clear})\end{aligned}$$

Once substitutions are applied, ReDunT verifies that the derived properties do not introduce contradictions. While redundancy rules are agnostic to which operation is rendered redundant, redundancy relations must respect causality.

By the properties of causal delivery, a newly delivered operation (o, t) can only be sequential or concurrent with operations already in the log. An example of a contradiction can be observed in the MVRegister as $t \prec t'$ in the derived R relation contradicts the causal delivery assumption. To resolve this, ReDunT removes the conflicting rule from the relation, yielding the following R relation: $(t, o)\ \mathsf{R}\ (t', o') \iff o[0] = \mathsf{clear}$. The R_ relation has no contradictions related to causality and, therefore, remains unchanged.

5. Simplify eval's definition. Recall that the starting eval function determines the CRDT state, assuming the log contains all operations. However, once redundancy relations are established, the system removes redundant operations from the log whenever a new operation is delivered. Therefore, the eval function can be simplified, as redundant operations no longer need to be considered. The final step of our methodology consists of refining the eval function by eliminating constraints that are now always satisfied due to the absence of redundant operations. For the MVRegister, this step simplifies the original eval function shown in figure 1 to $\{v \mid (t, [\mathsf{wr}, v]) \in s\}$ as we know from the redundancy relations that $\forall (t', \_) \in s \cdot t \not\prec t'$ always holds.

4 Use Case: Remove-Wins Set CRDT

We now illustrate how ReDunT applies to a more complex data type, a Remove-Wins Set (RWSet) CRDT. Starting from the original specification from [1], we explain how to derive a compact design, detailing each step and presenting the results obtained at each stage. The process starts with the RWSet concurrent semantics expressed by its eval function:

$$\begin{aligned}\mathsf{eval}_i(\mathsf{elems}, s) = \{v \mid\ & (t, [\mathsf{add}, v]) \in s \\ & \wedge \forall (t', [\mathsf{rmv}, v]) \in s \cdot t' \prec t \\ & \wedge \forall (t'', [\mathsf{clear}]) \in s \cdot t \not\prec t''\}\end{aligned}$$

The elems operation returns all values that satisfy three conditions. First, an add operation for the value exists in the log. Second, all rmv operations for the value have occurred before the corresponding add operation. Finally, no clear operations took place after the add operation.

Step 1. ReDunT starts by identifying the non-redundant operations. It analyses the eval definition to extract operations that influence the state. This process yields the following set of non-redundant operations for the RWSet:

$$\begin{aligned}&\{(t, [\mathsf{add}, v]) \in s \mid \\ &\wedge \forall (t', [\mathsf{rmv}, v]) \in s \cdot t' \prec t \wedge \forall (t'', [\mathsf{clear}]) \in s \cdot t \not\prec t''\}\end{aligned}$$

Step 2. ReDunT derives the set of redundant operations from the set of non-redundant operations in two steps.

Step 2.1. ReDunT computes the complement of the non-redundant operations. In this process, quantifiers and constraints are negated: universal quantifiers ($\forall$) become existential quantifiers ($\exists$), and temporal constraints are inverted, replacing $\prec$ with $\not\prec$, and vice versa. Furthermore, since rmv and clear operations serve only to filter add operations, they are inherently included in the complement. Given that the universal set consists of add, rmv, and clear operations, the complement of the non-redundant operations for the RWSet is defined as follows:

$$\begin{aligned}&\{(t, [\mathsf{add}, v]) \in s \mid \exists (t', [\mathsf{rmv}, v]) \in s \cdot t' \not\prec t\} \cup \\ &\{(t, [\mathsf{add}, v]) \in s \mid \exists (t'', [\mathsf{clear}]) \in s \cdot t \prec t''\} \cup \\ &\quad \{(t, [\mathsf{rmv}, v]) \in s\}\ \cup \{(t, [\mathsf{clear}, v]) \in s\}\end{aligned}$$

Step 2.2. ReDunT filters the complement result to exclude operations needed for conflict resolution. For example, in the RWSet, ReDunT needs to filter the rmv operations from the complement, as these are not redundant and must remain in the PO-Log. Making rmv operations redundant would lead to an incorrect CRDT specification. Specifically, when a new rmv operation arrives, all prior or concurrent add operations for the same value would be removed from the log. Additionally, if the rmv itself were classified as redundant, it would not be added to the log. Up to this point, the process may seem consistent. However, if a concurrent add operation were later delivered, it would be inserted into the log, as the absence of the concurrent rmv prevents it from being marked as redundant. Evaluating the state would incorrectly include the element in the set, violating the remove-wins semantics.

In contrast, clear operations are redundant. Unlike rmvs, which contribute to defining conflict resolution, clears only filter out preceding adds during state evaluation. As a result, clear operations can be safely considered redundant.

After filtering the complement set, ReDunT obtains the following set of redundant operations:

$$\begin{aligned}&\{(t, [\mathsf{add}, v]) \in s \mid \exists (t', [\mathsf{rmv}, v]) \in s \cdot t' \not\prec t\} \cup \\ &\{(t, [\mathsf{add}, v]) \in s \mid \exists (t'', [\mathsf{clear}]) \in s \cdot t \prec t''\} \cup \\ &\quad \{(t, [\mathsf{clear}, v]) \in s\}\end{aligned}$$

The formula defines the redundant operations as follows:

- all add operations that did not happen after a rmv operation for a given value v (i.e., the rmv is either concurrent with or happened after the add),
- all add operations that happened before a clear, or
- all clear operations.

These conditions align with the semantics of a remove-wins set, confirming that the identified operations are redundant and can be safely removed from the log.

Step 3. ReDunT formulates the redundancy rules by transforming the definition of redundant operations into an equivalent formulation that explicitly considers two operations: one that becomes redundant and another that makes the first redundant. As explained, each subset of redundant operations generates a corresponding redundancy rule. Consequently, in this example, ReDunT generates three redundancy rules, each addressing a distinct redundancy scenario.

$$o_r[0] = \mathsf{add} \land o[0] = \mathsf{rmv} \land o_r[1] = o[1] \land t \not\prec t_r \quad (1)$$

$$o_r[0] = \mathsf{add} \land o[0] = \mathsf{clear} \land t_r \prec t \quad (2)$$

$$o_r[0] = \mathsf{clear} \quad (3)$$

Step 4. ReDunT derives the redundancy relations by substituting the symbols representing operations and validating the resulting formulas to ensure no contradictions arise. Thus, for the RWSet, the R relation is given by:

$$\begin{aligned}(t,o)\ \mathsf{R}\ (t',o') \iff & (o[0] = \mathsf{clear}) \lor \\ & (o[0] = \mathsf{add} \land o'[0] = \mathsf{rmv} \land \\ & o[1] = o'[1] \land t \parallel t')\end{aligned}$$

The relation does not fully align with the redundancy rules. The first rule states that $t \not\prec t_r$ (equivalent to $t \succ t_r \lor t \parallel t_r$). However, in pure operation-based CRDTs, causal delivery ensures that an operation in the log cannot be a successor of a newly received operation. As a result, in the R relation, this rule only holds for $t \parallel t_r$. The second rule stipulates that the operation being made redundant must be a predecessor of the clear. This condition does not hold in R, as the newly received operation cannot be a predecessor of an operation already in the log due to the causal delivery property.

Conversely, the substitutions to the redundancy rule to generate R_ do not introduce contradictions. Therefore, the resulting formula fully matches the redundancy rules:

$$\begin{aligned}(t',o')\ \mathsf{R\_}\ (t,o) \iff & (o'[0] = \mathsf{clear}) \lor \\ & (o'[0] = \mathsf{add} \land o[0] = \mathsf{rmv} \land \\ & o'[1] = o[1] \land t \not\prec t') \lor \\ & (o'[0] = \mathsf{add} \land o[0] = \mathsf{clear} \land t' < t)\end{aligned}$$

Step 5. ReDunT simplifies the eval definition considering the established redundancy relations. As previously explained, eliminating redundant operations from the log allows for a more concise eval function, as those operations no longer need to be considered for computing the state. In particular, all add operations that occurred before a clear operation are redundant. Consequently, any add operation in the log must have occurred after a clear. Similarly, all add operations in the log must have occurred after a rmv since any rmv operation that is concurrent with or follows an add renders it redundant. As a result, the two quantified formulas in the eval will always evaluate to *true* when applying the redundancy relations. Since these formulas no longer provide meaningful constraints, we can simplify the eval function by removing them, yielding the following simplified definition:

$$\mathsf{eval}_i(\mathsf{elems}, s) = \{v \mid (t, [\mathsf{add}, v]) \in s\}$$

5 Portfolio of Pure Op-based CRDTs

We validated our approach by deriving the compact version of the portfolio of non-commutative data types presented in [1]. Table 1 overviews the concrete CRDTs. For each data type, the table includes the redundancy relations and eval function derived using ReDunT, and the (manually optimised) compact specification from [1].

When comparing both specifications, ReDunT does not always generate identical designs to those in [1]. In particular, ReDunT successfully derives an equivalent simplification for the multi-value register (see section 3). Despite structural differences, the formulas are semantically equivalent, and further simplifications could adjust our formula with [1].

For the remaining CRDTs, the manually optimised specifications eliminate more operations than the automatically derived ones with ReDunT. For instance, in the add-wins and remove-wins sets, ReDunT cannot remove sequential adds. This is because the eval formula that ReDunT takes as input provides no information about the semantics between sequential add operations. In contrast, the specifications in [1] were manually crafted by a programmer who understands the semantics of the operations, a level of knowledge that ReDunT lacks and, therefore, cannot derive automatically.

The remove-wins set and the disable-wins flag exhibit another difference compared to the original specification. Unlike in [1], where only sequential operations are marked as redundant, ReDunT identifies both sequential and concurrent operations as redundant. As a result, using ReDunT's synthesised relations, some operations can be removed from the log earlier than with the handcrafted design.

6 Implementation

We implemented the ReDunT approach for synthesising redundancy relations in pure op-based CRDTs using K [10]. K is a rewrite-based semantic framework to specify languages, type systems, and formal analysis tools. It defines execution semantics using configurations to represent program states and rewrite rules to model state transitions. We formally specified ReDunT in K by defining a transformation system for pure op-based CRDTs. Our approach captures the logical

Table 1. Comparison between the designs obtained using ReDunT and those presented in [1].

	ReDunT Specification	Original Compact Specification (from [1])
MVRegister	$(t,o)\ \mathsf{R}\ (t',o') \iff o[0] = \text{clear}$ $(t',o')\ \mathsf{R}\_\ (t,o) \iff (o'[0] = \text{wr} \land t' < t) \lor (o'[0] = \text{clear})$ $\text{eval}_i(\text{rd}, s) = \{v \mid (t, [\text{wr}, v]) \in s\}$	$(t,o)\ \mathsf{R}\ (t',o') \iff o[0] = \text{clear}$ $(t',o')\ \mathsf{R}\_\ (t,o) \iff t' < t$ $\text{eval}_i(\text{rd}, s) = \{v \mid (\_, [\text{wr}, v]) \in s\}$
Add-Wins Set	$(t,o)\ \mathsf{R}\ (t',o') \iff (o[0] = \text{clear}) \lor (o[0] = \text{rmv})$ $(t',o')\ \mathsf{R}\_\ (t,o) \iff (o'[0] = \text{clear}) \lor (o'[0] = \text{rmv}) \lor$ $(o'[0] = \text{add} \land o[0] = \text{rmv} \land o'[1] = o[1] \land t' < t) \lor$ $(o'[0] = \text{add} \land o[0] = \text{clear} \land t' < t)$ $\text{eval}_i(\text{elems}, s) = \{v \mid (t, [\text{add}, v]) \in s$	$(t,o)\ \mathsf{R}\ (t',o') \iff (o[0] = \text{clear}) \lor (o[0] = \text{rmv})$ $(t',o')\ \mathsf{R}\_\ (t,o) \iff t' < t \land (o[0] = \text{clear} \lor o[1] = o'[1])$ $\text{eval}_i(\text{elems}, s) = \{v \mid (\_, [\text{add}, v]) \in s\}$
Rmv-Wins Set	$(t,o)\ \mathsf{R}\ (t',o') \iff (o[0] = \text{clear}) \lor$ $(o[0] = \text{add} \land o'[0] = \text{rmv} \land o[1] = o'[1] \land t \parallel t')$ $(t',o')\ \mathsf{R}\_\ (t,o) \iff (o'[0] = \text{clear}) \lor$ $(o'[0] = \text{add} \land o[0] = \text{rmv} \land o'[1] = o[1] \land t \not< t') \lor$ $(o'[0] = \text{add} \land o[0] = \text{clear} \land t' < t)$ $\text{eval}_i(\text{elems}, s) = \{v \mid (t, [\text{add}, v]) \in s\}$	$(t,o)\ \mathsf{R}\ (t',o') \iff o[0] = \text{clear}$ $(t',o')\ \mathsf{R}\_\ (t,o) \iff t' < t \land (o[0] = \text{clear} \lor o[1] = o'[1])$ $\text{eval}_i(\text{elems}, s) = \{v \mid (\_, [\text{add}, v]) \in s \land (\_, [\text{rmv}, v]) \notin s\}$
Enable-Wins Flag	$(t,o)\ \mathsf{R}\ (t',o') \iff (o[0] = \text{clear}) \lor (o[0] = \text{disable})$ $(t',o')\ \mathsf{R}\_\ (t,o) \iff (o'[0] = \text{clear}) \lor (o'[0] = \text{disable}) \lor$ $(o'[0] = \text{enable} \land o[0] = \text{disable} \land t' < t) \lor$ $(o'[0] = \text{enable} \land o[0] = \text{clear} \land t' < t)$ $\text{eval}_i(\text{read}, s) = (t, [\text{enable}]) \in s$	$(t,o)\ \mathsf{R}\ (t',o') \iff (o[0] = \text{clear}) \lor (o[0] = \text{disable})$ $(t',o')\ \mathsf{R}\_\ (t,o) \iff t' < t$ $\text{eval}_i(\text{read}, s) = (\_, [\text{enable}]) \in s$
Disable-Wins Flag	$(t,o)\ \mathsf{R}\ (t',o') \iff (o[0] = \text{clear}) \lor$ $(o[0] = \text{enable} \land o'[0] = \text{disable} \land t' \parallel t)$ $(t',o')\ \mathsf{R}\_\ (t,o) \iff (o'[0] = \text{clear}) \lor$ $(o'[0] = \text{enable} \land o[0] = \text{disable} \land t \not< t') \lor$ $(o'[0] = \text{enable} \land o[0] = \text{clear} \land t' \prec t)$ $\text{eval}_i(\text{read}, s) = (t, [\text{enable}]) \in s$	$(t,o)\ \mathsf{R}\ (t',o') \iff o[0] = \text{clear}$ $(t',o')\ \mathsf{R}\_\ (t,o) \iff t' < t$ $\text{eval}_i(\text{read}, s) = (\_, [\text{enable}]) \in s \land (\_, [\text{disable}]) \notin s$

structure of the eval definition and uses formal rewrite rules to compute the redundancy relations.

The ReDunT implementation in K is still a work in progress, and it is not extensible to all CRDTs used to evaluate the methodology. Currently, it only supports sets. Nevertheless, it provides a structured foundation for synthesising optimisations in replicated data types.

7 Related Work

Prior research has focused on synthesising CRDTs [7, 8] and deriving RDTs from sequential data types [5, 6, 9].

Mergeable Data Types [7] automatically derive correct distributed data types by leveraging merge functions to resolve conflicts in a principled way. Katara [8] is a synthesis-based system that transforms sequential data type implementations into verified CRDT designs, lifting annotated C/C++ data types into nearly equivalent CRDT implementations.

Hamsaz [6] and Hampa [9] automatically synthesise replicated objects by coordinating unsafe operations. ECROs [5] derives RDTs from sequential data types based on a distributed specification, where application semantics are expressed through invariants over the replicated state.

While these approaches successfully synthesise RDTs, they do not optimise log storage. Particularly, mergeable data types face practical challenges in efficiently storing, computing, and retrieving the lowest common ancestor between two concurrent versions, as highlighted in [7]. ReDunT complements these works by focusing on log storage optimisation, systematically deriving redundancy relations to eliminate unnecessary operations while preserving correctness.

8 Conclusion

In this paper, we presented ReDunT, a methodology for automatically deriving the redundancy relations of pure operation-based CRDTs. Our approach allows programmers to focus solely on defining the concurrency semantics for pure operation-based CRDTs in the eval function. ReDunT then automatically synthesises redundancy compaction functions to optimise log storage. By eliminating the need for programmers to manually define these functions, ReDunT simplifies the implementation of new CRDTs and reduces the risk of introducing bugs. To the best of our knowledge, ReDunT is the first framework capable of synthesising optimisations for log storage. Currently, ReDunT is designed for pure operation-based CRDTs. In the future, we aim to explore its applicability to optimising other types of logs.

Acknowledgments

We would like to thank Carlos Baquero for his initial idea and valuable discussion. This work is partially supported by PRT/BD/154519/2022 awarded by the EUTOPIA European University Alliance, PRT/BD/154787/2023 awarded by the CMU Portugal Affiliated Ph.D. program, UID/04516/NOVA Laboratory for Computer Science and Informatics (NOVA LINCS) with the financial support of FCT.IP and the European Commission through the TaRDIS project (agreement ID 101093006).

References

[1] Carlos Baquero, Paulo S. Almeida, and Ali Shoker. 2017. Pure Operation-Based Replicated Data Types. *CoRR* abs/1710.04469 (2017). arXiv:1710.04469

[2] Jim Bauwens and Elisa Gonzalez Boix. 2023. Nested Pure Operation-Based CRDTs. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:26. https://doi.org/10.4230/LIPIcs.ECOOP.2023.2

[3] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (2012), 23–29. https://doi.org/10.1109/MC.2012.37

[4] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. 2012. Cloud Types for Eventual Consistency. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–307.

[5] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. 2021. ECROs: Building Global Scale Systems from Sequential Code. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 107 (oct 2021), 30 pages. https://doi.org/10.1145/3485484

[6] Farzin Houshmand and Mohsen Lesani. 2019. Hamsaz: replication coordination analysis and synthesis. *Proc. ACM Program. Lang.* 3, POPL (2019), 74:1–74:32. https://doi.org/10.1145/3290387

[7] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable replicated data types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 154 (oct 2019), 29 pages. https://doi.org/10.1145/3360580

[8] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. 2022. Katara: Synthesizing CRDTs with Verified Lifting. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 173 (oct 2022). https://doi.org/10.1145/3563336

[9] Xiao Li, Farzin Houshmand, and Mohsen Lesani. 2020. 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 (Lecture Notes in Computer Science, Vol. 12224)*. Springer, 324–349. https://doi.org/10.1007/978-3-030-53288-8_16

[10] Runtime Verification. 2024. K Framework - GitHub Repository. https://github.com/runtimeverification/k Accessed: 1 Feb. 2024.

[11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://inria.hal.science/inria-00555588