# AMC: Towards Trustworthy and Explorable CRDT Applications with the Automerge Model Checker

Andrew Jeffery
University of Cambridge
Cambridge, United Kingdom
andrew.jeffery@cst.cam.ac.uk

Richard Mortier
University of Cambridge
Cambridge, United Kingdom
richard.mortier@cst.cam.ac.uk

## Abstract

Conflict-free Replicated Data Types (CRDTs) enable local-first operations and asynchronous collaboration without the need for always-on centralised services. CRDTs can have a high overhead, so implementations need to be optimised, but this optimisation can lead to bugs despite the use of test suites and fuzzing. Furthermore, using CRDTs in applications is complex, observing unexpected conflict resolution, issues synchronising documents and difficulties implementing appropriate data models. Automerge is a library, exposing a JSON CRDT, that sees users having difficulties in modelling their problems, understanding their edge cases and implementing applications correctly. We introduce the Automerge Model Checker (AMC), empowering application developers to check properties about their implementations and explore them dynamically. AMC can check a range of applications as well as being able to check properties about the core of Automerge itself, helping to make more trustworthy Automerge applications.

AMC is available open-source at github.com/jeffa5/automerge-model-checker.

***CCS Concepts:*** • **Software and its engineering** → *Formal software verification*; • **Computer systems organization** → *Peer-to-peer architectures.*

***Keywords:*** conflict-free replicated data types, model checking, distributed systems, eventual consistency

## 1 Introduction

Conflict-free Replicated Data Types (CRDTs) [35] are data structures that enable concurrent operation with consistent merge results. There are many CRDTs available, representing counters, maps, sets, lists [35], rich-text [28], trees [29], filesystems [39, 42], and JSON documents [21]. Several libraries are available that provide a core CRDT and functionality to handle synchronisation and persistence, primarily Automerge [22, 36] and Yjs [32, 33, 38] but many others are available [2–10, 13, 40].

Compared to raw documents, CRDTs incur an overhead to store the history of edits and information to perform merge operations between divergent histories. This leads to new data structures [41], storage formats [19] and processing methods [12, 41] being developed to provide performant and efficient implementations. These new developments pose risks for the correctness of the implementations, requiring extensive testing and fuzzing separately from formal verification of the abstract CRDT itself [21]. Whilst mechanized verification of CRDTs [11] is available it operates on constrained DSLs and cannot aid more optimized implementations, or applications built on them.

Despite the emphasis on performant, efficient, and correct CRDT implementations, the libraries can still be challenging to use due to the complexity in understanding how sequences of CRDT operations will be resolved. This manifests as challenges in predicting conflict resolution, correctly handling synchronisation, and correctly implementing data models on top of the CRDT library. These can lead to applications, and users, losing data along with users seeing confusing and unexpected behaviour, such as interleaving [23].

Automerge is a library, modelling JSON with an operation-based CRDT, that has undergone a rewrite to attain higher performance and efficiency. The use of fuzzing and a thorough test suite aid the correctness of the library but do not prevent complex bugs from arising [14]. Additionally, these do not solve issues experienced by developers such as writes disappearing [18], issues synchronising across peers [30] and persistence [34].

This paper introduces the Automerge Model Checker (AMC), a library for building model checked Automerge applications in Rust. AMC deterministically explores the behaviour of the Automerge application to cover both common and edge case behaviour. AMC provides a system for adding this functionality by application developers without deep knowledge of Automerge itself, instead focusing on the desired application's business logic and properties. The key contributions of this work are that AMC:

1. can check user-defined properties of applications built on Automerge; three such applications are presented based on counters, moving items in a list, and todos, §3.
2. enables developers to dynamically explore the conflict-resolution behaviour of their Automerge applications, §4.
3. can check the Automerge library *implementation*, rather than just a high level description of the CRDT, §5.

## 2 Automerge Model Checker

***Automerge application architecture.*** Each Automerge instance represents a JSON document, capturing operations performed on it and providing extra functionality such as

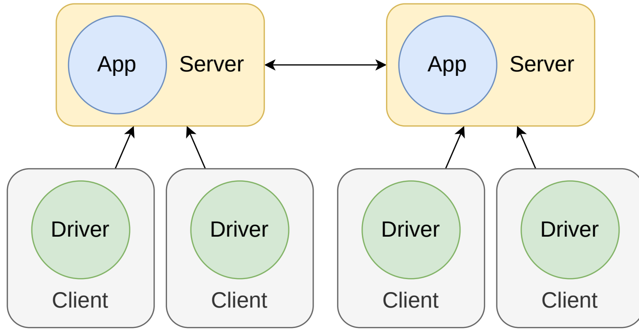


**Figure 1.** High level architecture of AMC. Developers implement the Drivers and the App.

historical views, efficient storage and logic for synchronisation. The document stores a JSON object at the root, termed a *map* in Automerge, which has string keys and values of either a map, an ordered list, text, or some scalar type such as `String`, `Bytes`, `Counter`, `Int`, `Boolean` or `Null`. Lists store an ordered collection of maps, lists or scalars. Text is a specialised object for use in collaborative text editing.

Applications leveraging Automerge primarily target local-first operation, enabling users to work offline and synchronise their changes once they reconnect. To synchronise, there can either be a central server, able to synchronise changes to clients who are not online at the same time, or clients can synchronise directly with each other in a peer to peer fashion. AMC can represent both strategies since the former is a peer in the latter strategy that performs no operations.

Internally, Automerge stores operations in *changes* which encapsulate the operation along with metadata and a list of hashes of parent changes. These changes are synchronised and applied to other documents to share the operations. Changes are arranged in a directed acyclic graph based on their hash which is used for synchronisation and analysing divergence between peers.

***Overview of AMC.*** We will use a shared counter being incremented and decremented as a running example. The initial document looks like `{"value":0}`.

To model an Automerge application, AMC separates the logic interacting with the Automerge document from the triggering of that logic. In our example the increment and decrement actions operate on the document, and we would have some other logic to *trigger* those actions. This enables actions to be interleaved in the model checker, enumerating different execution orderings. Application logic executes atomically on the document, preventing concurrent interactions such as receiving a synchronisation message.

AMC comprises four components, visualised in Figure 1. Developers using AMC implement the *application* (app), which they can reuse for their real application, and the *driver*, specifically for AMC. These components are written in Rust. The application interacts with the Automerge document, being triggered by *inputs* to mutate and inspect the document before producing *outputs*. For our example application the inputs would be either `Increment` or `Decrement`, and our outputs could be the new value. The driver creates the inputs to trigger application functions. The driver can be simple, only sending events and ignoring responses, or complex, handling responses too and conditionally sending more inputs. A driver for our example application would emit one of the inputs, `Increment` or `Decrement`, and ignore responses. Multiple instances of the application are spawned during the model checking, each having multiple drivers to exercise different input orderings.

The *server* and *client* are implemented by AMC itself and wrap the application and driver, respectively. The server handles synchronising with peers through different strategies: sending raw changes, using the built-in Automerge sync protocol, or saving the document and sending it in its entirety. Each of these has different characteristics and use cases, so each can be checked by AMC in separate runs. AMC can also explore all combinations of when to synchronise, adding the potential for batches of changes to be sent at once. Servers can also be restarted during checking to ensure that state is correctly persisted, including synchronisation state. The client simply wraps the driver into AMC. Internally, AMC connects the client and server with a reliable, ordered connection. This means that operations from a client will happen in series but may be interleaved with those from other clients. This network also connects servers to each other, meaning that synchronisation messages will be delivered in order.

***Adding properties.*** Developers will want to express properties over application behaviour to ensure properties hold. To support this, AMC provides the ability to define properties as functions over the model, including history and configuration, and the current state of the network. This enables developers to, for example, iterate over the Automerge document of each server and assert facts on them, leveraging the power of the Rust language, rather than a constrained DSL. This can be aided by the recording of application inputs and outputs to build a history for checking properties that rely on the path of operations currently performed.

Properties can be expressed as either *always* properties, holding for every state, or *sometimes* properties, holding at least once during the run of the model checking. During checking, these properties are evaluated at every step and any discoveries are recorded: counter-examples for always properties and examples for sometimes properties. Running executes until the state space is exhausted or all properties have discoveries found.

When writing properties it is common to consider the case when all peers are synchronised. To aid in writing these properties, AMC provides a utility for checking that all peers

```
|_model, state| {
  if !syncing_done(state) { return true; }
  let mut expected = 0;
  for msg in state.history {
    match msg.input() {
      Some(Msg::Increment) => { expected += 1; }
      Some(Msg::Decrement) => { expected -= 1; }
      None => {}
    }
  }
  if let Server(s) = state.actor_states.first() {
    let actual = s.document().get(ROOT, "value");
    return actual.to_i64() == expected;
  }
  panic!("Could not find a server!");
}
```

**Listing 1.** Implementation of the correctness property for a counter, syntax simplified for brevity.

have finished synchronising any local changes. This is implemented as a check that no server has outstanding changes to propagate and that the network has delivered any pending synchronisation messages.

For our example application, once peers have synchronised they should always have the correct value according to the number of increments and decrements applied at that point. The messages applied to the applications are recorded for model checking, making them available in properties. This property an *always* property, Listing 1 shows how it might look when implemented in Rust.

***Running the checker.*** To run the model checker the model needs to be built into a binary using an AMC library which handles setting up the CLI interface. During execution, AMC leverages the Stateright [31] model checker to perform the search. AMC builds the user-supplied model into an instance of the *actor model* in Stateright, treating clients and servers as actors sending messages to each other over a FIFO ordered reliable network. AMC additionally sets timers to trigger synchronisation and server restarts, made possible through our upstream contributions to Stateright for differentiable timers [17]. This separation aids in the reliability of the checking as Stateright can be used in other projects and improvements to Stateright will in turn improve AMC. Whilst AMC currently only works with applications built in Rust, it would be feasible to extend this to other languages that use Automerge bindings, such as JavaScript.

AMC can use different strategies to explore the state space. Breadth-first search is useful for finding the shortest paths for counter-examples but incurs a large memory overhead. Depth-first search quickly checks deep, complex interactions but discoveries are unlikely to be the shortest, this mode uses

```
Property Always "correct value" FAILED
counterexample, Path[6]:
- Deliver{src:2, dst:0, msg:Input(Increment)}
- Deliver{src:4, dst:1, msg:Input(Increment)}
- Timeout(0, Server(Synchronise))
- Deliver{src:0, dst:1, msg:SyncChange([...])}
- Timeout(1, Server(Synchronise))
- Deliver{src:1, dst:0, msg:SyncChange([...])}
```

**Listing 2.** Failed counter property, syntax simplified and content elided (...) for brevity.

considerably less memory. Iterative depth-first search combines these approaches, performing depth-first searches with increasing maximum depths to find the shortest discoveries with low memory usage but increased time. This was made possible through our upstream contributions to add depth tracking and limiting in Stateright [16]. Large application models can lead to large state spaces, making these searching methods impractical due to resource constraints; a stateless checker may be more suitable for these complex applications.

Timers can be used for triggering synchronisation and restarts at servers. Timeouts for these timers are injected at every possible place by Stateright during checking. The use of timers enables checking paths where servers have not eagerly tried to synchronise, instead batching changes, and restarting nodes at different points.

Running our example with an `Int` value type, we get a counter-example. The output gives a path of messages that triggered the counter-example, shown in Listing 2. Changing the datatype to an Automerge `Counter` enables tracking of intent, fixing this issue when also coupled with consistent initialization logic, expanded on later. This example is included in the results table for comparison, Table 1.

## 3 Checking Automerge applications

AMC aids developers of Automerge applications to catch bugs arising from the complex interactions of their application behaviour. We present three applications, highlighting how AMC can aid developers to catch bugs by efficiently finding the issues. These models assume correctness of Automerge, which will be explored separately in §5.

Table 1 shows the states, depth and duration resulting from AMC runs for the versions of each application presented. All model checking runs presented were executed a single time using iterative depth-first search. The machine was an Azure Cloud VM (Standard D8s v3) with 8 cores and 32 GiB of RAM, running Ubuntu Linux 20.04 and the programs were compiled with Rust version 1.66.1.

***Counter.*** Continuing our example from the previous section we still had a failure of the property which stems from the issue of Automerge not merging unrelated histories. In this particular example the `Counter` type is not the same

**Table 1.** Model checking results. All runs used iterative DFS, results are presented for the deepest traversal only, using 3 servers eagerly syncing with changes, no restarts. CI = Consistent initialization. LoC includes library and CLI code.

| App | Version | Total states | Unique states | Max depth | Duration (s) | Correct | LoC |
|---|---|---|---|---|---|---|---|
| Counter | Initial | 10,919 | 5,684 | 8 | 0.071 | ✗ | 386 |
| | Counter type | 10,963 | 5,697 | 8 | 0.078 | ✗ | |
| | CI | 10,779 | 5,618 | 8 | 0.080 | ✗ | |
| | Counter type & CI | 34,528,681 | 10,576,193 | 19 | 110.796 | ✓ | |
| List moves | Initial | 10,949 | 5,694 | 8 | 0.091 | ✗ | 271 |
| Todos | Initial | 23,735 | 13,779 | 8 | 0.176 | ✗ | 554 |
| | Random IDs | 23,304 | 14,128 | 8 | 0.193 | ✗ | |
| | CI | 22,263 | 13,027 | 8 | 0.178 | ✗ | |
| | Random IDs & CI | 25,653,499 | 7,826,621 | 19 | 125.461 | ✓ | |
| Automerge | Maps | 1,259,686 | 389,187 | 19 | 30.495 | ✓ | 1179 |
| | Lists | 1,497,109 | 473,715 | 19 | 37.124 | ✓ | |
| | Text | 1,497,109 | 473,715 | 19 | 36.171 | ✓ | |

instance between peers, so increments applied to it end up applying to separate instances. These semantics are a topic of debate around Automerge [18]. Adding some logic to consistently initialise the counter on each peer gives us the fully working solution. Having setup logic for consistent initialisation is common practice to ensure a common starting point, similar to schema migrations in databases. However, extending this to schema changes during the application life cycle is more complicated, requiring solutions such as Cambria [27].

***List moves.*** The second application is moving items within a list. This application starts with a list of three items and tries to move one item within the list. Without a native move operation one is typically implemented as a deletion followed by an insertion, as done here. The property to check is that after moving items within the list the same items exist as before. Checking this model with AMC finds a path to an invalid state where items have been duplicated in the list, shown in Figure 2. This is a known issue that requires native support in the CRDT [20] so there is currently no fix. However, it does serve to show the efficiency of checking for the issue, aiding library developers when adding the operation to the libraries.

***Todos.*** The third application is managing a group of todo-list items. As demonstrated from the list moves application, storing items in a list can lead to unintended side-effects. Due to this, our todo application chooses to give each todo an ID and store them in a map keyed by this ID. Presenting these todos to the user could then be handled by a wrapper that orders the entries by some criteria. Drivers can try to create new todos, update existing todos, toggle their completion status, and remove them. For checking, only creation and removal are enabled. The property for our model is that, after synchronisation, all peers should have the same number

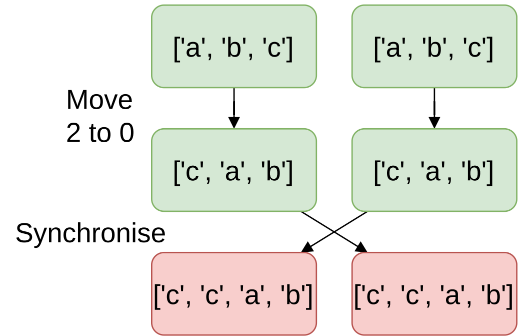


**Figure 2.** Example of a problematic trace in amc-moves. List items should not be being duplicated.

of todos as if the operations were performed on a single application instance, i.e. they should not clash. To implement this property, recorded history of application inputs and outputs is replayed, tracking how many todos would be present at each point.

The initial version starts out with using sequential IDs for each todo. When a client asks for a new todo to be created, the application generates an ID by taking the highest ID and incrementing it by 1. Checking this model quickly finds that sequential IDs are not suitable in this instance: concurrently creating todos leads to a clash in IDs (Figure 3).

Changing from sequential IDs to random IDs, using a seeded RNG for checking, contributes to a fix. Adding consistent initialization logic is also required to ensure todos are added to the same map within Automerge. This model has more functionality than needed to produce the initial failures. After small runs have succeeded, other combinations can be checked by enabling updates to existing todos and toggling of completion status. This will increase the state

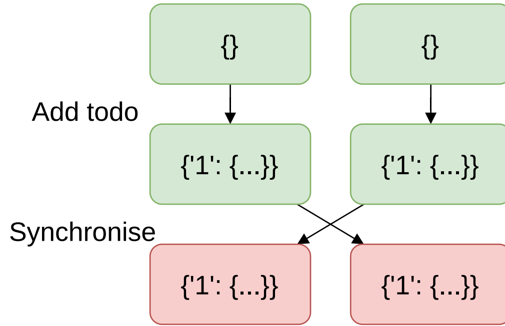


**Figure 3.** Example of a trace in amc-todo with the initial version. The IDs should not clash.

space to search and so the time required to check, making it important to work with smaller models first.

## 4 Dynamically exploring models

Once developers integrate their application's business logic with AMC, they may not want to define, or may not fully know, the properties they expect. Exploration of the interactions between peers with the custom logic is designed to help with this, as well as debugging failing properties. Rather than a DFS, BFS or iterative DFS, the exploration mode uses an *on-demand* checker, contributed to Stateright as part of this work [15]. This checker is lazy until asked to check new states by the web UI, shown in Figure 4. This means that models can be explored without the need of powerful machines whilst still using the full original model, inheriting properties, following through actions, and exploring different pathways with the ability to retrace steps.

The ability to work through the model on-demand is coupled with the ability to efficiently jump to a specific path. When a broken property is found during checking, the checker outputs the path to the final state for passing to the explorer. The explorer then iterates through the path, calculating the states on-demand, to reach the desired state. This can be particularly useful for developers exploring paths for violations that may be found on more powerful machines.

## 5 Checking Automerge

For checking Automerge itself, the application exposes a simplified version of the Automerge API. There is one driver per function in this simplified API. The application functions are split by object type: for map objects drivers can either put or delete values, for lists and text they can insert or delete. When the values are counters increment operations are also added. For checking, the documents are initialised with the required object at a preset location and results presented in Table 1 are for string values. Splitting checker runs by types of objects enables us to limit the state space to search, making checking of particular areas feasible. This is safe as operations on an object do not affect other objects.

Each driver sends only an initial message, avoiding handling of outputs. This means that operations can have no effect when they reach the application if they act on an invalid key or index. For instance, indices that drivers use for list operations are pre-programmed rather than having to perform a get or length call followed by the operation. This helps to keep the checking efficient and avoid the document changing between two dependent actions, such as getting the length of a list and acting on that, emphasising the importance of atomically working on the document. Additionally, to keep the state space small, our checking uses one key/index to focus on conflicts. We expect that this could be expanded to larger scenarios with more optimisations.

***Properties.*** Automerge has a few high-level safety properties that should hold during execution, explained here.

One of the main properties of a CRDT is that it is convergent, that is, after synchronisation has finished all peers should have the same view of the data. This is checked with an *always* property that ensures all documents have the same values observed from walking the JSON document.

Saving and subsequently loading a document is a key property used for Automerge applications, focusing on local-first, offline operation. To ensure this behaviour during checking an *always* property is added that checks, at every state, that the document can be saved and loaded to reconstruct the same document.

During execution, an application could reach a state of internal failure when using Automerge. To detect this another *always* property is added to check that no errors were produced. This allows errors to be easily reproduced, rather than aborting execution directly with a stack trace.

In addition to interacting with and checking the latest state of the document, historical versions of the document can be inspected. This enables checking more complex internals of Automerge for reading historical states. This works by means of an *always* check that, for each document, constructs historical documents from the observed changes and compares the corresponding historical view of the current document with each historical document.

## 6 Related work

Katara [25] is a tool for synthesizing verified CRDT designs from sequential data type implementations. Whilst the synthesized CRDT designs are verified, ensuring correctness, they lack an implementation, leading the verified design to drift from an implementation. AMC instead aims to check that existing implementations, which use complex logic for performance, are correct. AMC could potentially be applied to the implementations of these generated CRDTs.

VeriFX [11] provides a framework for describing the semantics of CRDTs at a high level and checking properties about the model. Additionally, these can synthesise implementations of the CRDTs into executable code. This approach

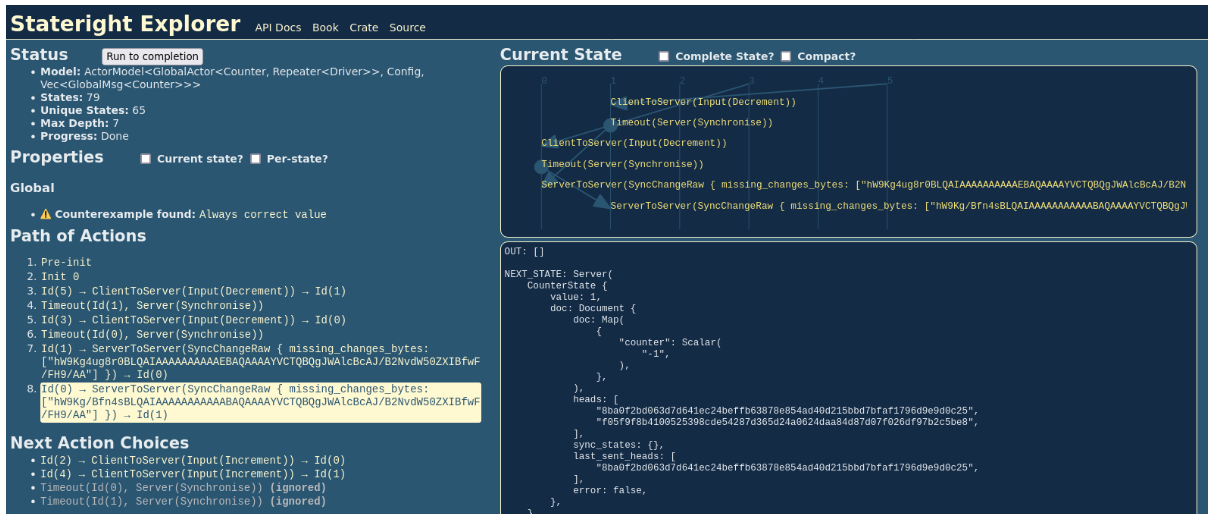


**Figure 4.** Web UI for the explore mode, showing a failure in the initial counter application.

is useful for creating new CRDTs however most CRDT libraries still have lots of other functionality that would remain to be implemented. Moreover, this approach does not aid application developers using these synthesised CRDT libraries, leaving them to instead work through an abstract model in an unfamiliar language to understand the edge-cases. AMC instead provides a more dynamic and direct way for developers to explore the behaviour of their applications.

Fuzz testing using tools like libFuzzer [37] and AFL [1, 26] are increasingly prevalent for checking implementations of CRDTs, yet rarely focus on the higher-level aspects, instead focusing on checking parsing logic of serialized data. These approaches also do not aid application developers using the libraries in understanding the behaviour they experience. AMC focuses efforts on the high-level aspects of the CRDT library and applications built on-top, as well as providing functionality for checking lower-level properties.

Implementing CRDTs and their components efficiently has many challenging aspects and work to improve the storage and memory usage [19], synchronisation [24], and processing speed [12] is widespread. It is due to this added implementation complexity that more rigorous testing approaches are being seen. As the implementations get more performant and efficient, checking them, such as with AMC, becomes more practical as a larger state space can be explored in the same time with the same resources. With this larger state space, AMC can check more complex interactions, providing more trust in the implementation and optimisations.

## 7 Conclusions

This paper presented AMC, the Automerge Model Checker, to aid in exploring and checking the properties of applications built on the Automerge CRDT library and the core library itself. AMC provides developers with a UI to dynamically explore the interactions of their application logic. AMC can aid developers in finding failing properties in their application implementations and then check the fixed implementations for correctness. Future work could extend this to extract statistical properties of the interactions as well as improving the efficiency of checking to enable more complex models. This approach could be extended to checking applications written in other languages that use Automerge as well as other CRDT implementations, easing the exploration of their features. Overall, AMC helps developers of Automerge applications to have more trust in, and understanding of, their applications.

## Acknowledgments

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

## References

[1] American fuzzy lop plus plus (afl++). https://aflplus.plus/. (accessed: 2023.02.23).

[2] Client-server communication framework based on optimistic ui, crdt, and log. https://logux.io/. (accessed: 2023.02.24).

[3] Crdts in clojure(script) with edn serialization. https://github.com/aredington/schism. (accessed: 2023.02.24).

[4] Dart implementation of conflict-free replicated data types (crdts). https://github.com/cachapa/crdt. (accessed: 2023.02.24).

[5] Live information sharing. https://m-ld.org/. (accessed: 2023.02.24).

[6] Minimal peer-to-peer database, based on kappa architecture. https://github.com/kappa-db/kappa-core. (accessed: 2023.02.24).

[7] An open source cybersecurity protocol for syncing decentralized graph data. https://github.com/amark/gun. (accessed: 2023.02.24).

[8] Replicated data types in akka. https://doc.akka.io/docs/akka/2.6.3/typed/distributed-data.html#replicated-data-types. (accessed: 2023.02.24).

[9] Replicated object notation. http://replicated.cc/. (accessed: 2023.02.24).

[10] Yorkie is a document store for collaborative applications. https://github.com/yorkie-team/yorkie. (accessed: 2023.02.24).

[11] Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. Verifx: Correct replicated data types for the masses, 2022.

[12] Seph Gentle. 5000x faster crdts: An adventure in optimization. https://josephg.com/blog/crdts-go-brrr. (accessed: 2023.02.23).

[13] Seph Gentle. The world's fastest crdt. wip. https://github.com/josephg/diamond-types. (accessed: 2023.02.23).

[14] Orion Henry. rework how skip works to push the logic into node. https://github.com/automerge/automerge/pull/531. (accessed: 2023.02.23).

[15] Andrew Jeffery. Add an on-demand checker in stateright. https://github.com/stateright/stateright/pull/30. (accessed: 2023.04.04).

[16] Andrew Jeffery. Add depth tracking and limiting in stateright. https://github.com/stateright/stateright/pull/38. (accessed: 2023.04.04).

[17] Andrew Jeffery. Named timers in stateright. https://github.com/stateright/stateright/pull/42. (accessed: 2023.04.04).

[18] Martin Kleppmann. Behaviour of concurrently created objects under the same key. https://github.com/automerge/automerge-classic/issues/4. (accessed: 2023.02.23).

[19] Martin Kleppmann. Crdts: The hard parts. https://martin.kleppmann.com/2020/07/06/crdt-hard-parts-hydra.html. (accessed: 2023.02.23).

[20] Martin Kleppmann. Moving elements in list crdts. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Martin Kleppmann and Alastair R. Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.

[22] Martin Kleppmann and Alastair R Beresford. Automerge: Real-time data sync between edge devices. In *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK 2018). https://mobiuk.org/abstract/S4-P5-Kleppmann-Automerge. pdf*, pages 101–105, 2018.

[23] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. Interleaving anomalies in collaborative text editors. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Martin Kleppmann and Heidi Howard. Byzantine eventual consistency and the fundamental limits of peer-to-peer databases, 2020.

[25] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: Synthesizing crdts with verified lifting, 2022.

[26] lcamtuf. american fuzzy lop. https://lcamtuf.coredump.cx/afl/. (accessed: 2023.02.23).

[27] Geoffrey Litt, Peter van Hardenberg, and Orion Henry. Cambria: Schema evolution in distributed systems with edit lenses. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. Peritext: A crdt for collaborative rich text editing. *Proc. ACM Hum.-Comput. Interact.*, 6(CSCW2), nov 2022.

[29] Stéphane Martin, Mehdi Ahmed-Nacer, and Pascal Urso. Abstract unordered and ordered trees crdt, 2012.

[30] Andy Matuschak. Trade-offs: sparse document structure / syncing many documents? https://github.com/automerge/automerge-classic/issues/342. (accessed: 2023.02.23).

[31] Jon Nadal. A model checker for implementing distributed systems. https://github.com/stateright/stateright. (accessed: 2023.02.23).

[32] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In *Proceedings of the 15th International Conference on Engineering the Web in the Big Data Era - Volume 9114*, ICWE 2015, page 675–678, Berlin, Heidelberg, 2015. Springer-Verlag.

[33] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 2016 ACM International Conference on Supporting Group Work*, GROUP '16, page 39–49, New York, NY, USA, 2016. Association for Computing Machinery.

[34] Roger Qiu. Efficiently persisting on every change. https://github.com/automerge/automerge-classic/issues/331. (accessed: 2023.02.23).

[35] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400. Springer LNCS volume 6976, October 2011.

[36] Automerge team. Rust implementation of automerge. https://github.com/automerge/automerge. (accessed: 2023.02.23).

[37] LLVM team. libfuzzer – a library for coverage-guided fuzz testing. https://www.llvm.org/docs/LibFuzzer.html. (accessed: 2023.02.23).

[38] Yjs team. Shared data types for building collaborative software. https://github.com/yjs/yjs. (accessed: 2023.02.23).

[39] Romain Vaillant, Dimitrios Vasilas, Marc Shapiro, and Thuy Linh Nguyen. Crdts for truly concurrent file systems. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '21, page 35–41, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. Legion: Enriching internet services with peer-to-peer interactions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 283–292, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[41] Peter van Hardenberg. Automerge 2.0. https://automerge.org/blog/automerge-2/. (accessed: 2023.02.23).

[42] Elena Yanakieva, Michael Youssef, Ahmad Hussein Rezae, and Annette Bieniusa. Access control conflict resolution in distributed file systems using crdts. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '21, New York, NY, USA, 2021. Association for Computing Machinery.