Proceedings of the of the 17th Conference on Computer Science and Intelligence Systems pp. 849–853
DOI: 10.15439/2022F99
ISSN 2300-5963 ACSIS, Vol. 30

Extensible Conflict-Free Replicated Datatypes for Real-time Collaborative Software Engineering

Istvan David, Eugene Syriani, Constantin Masson
Department of Computer Science and Operations Research (DIRO) – Université de Montréal, Canada
Email: istvan.david@umontral.ca, syriani@iro.umontreal.ca

***Abstract*—Real-time collaboration has become a prominent feature of nowadays' software engineering practices. Conflict-free replicated data types (CRDT) offer efficient mechanisms for implementing real-time collaborative environments. However, the lack of extensibility of CRDT limits their applicability. This is a particularly important problem in settings relying on complex, non-linear data types. In this paper, we report our results in augmenting primitive CRDT with extension mechanisms. We demonstrate our technique through an example from the realm of model-driven engineering, where graph types are prevalent.**

***Index Terms*—CRDT, Collaborative software engineering, Strong eventual consistency, Concurrency control**

I. Introduction

TODAY'S software engineering is often carried out in distributed settings [1], necessitating advanced coordination mechanisms, such as real-time collaboration. A key challenge in real-time collaboration is to guarantee the convergence of the stakeholders' local replicas while ensuring timely execution and smooth user experience [2]. Traditional mechanisms that implement locking [3] fall short of addressing this challenge. A more appropriate consistency model for real-time collaboration is strong eventual consistency (SEC) [4]. SEC ensures that (i) the updates will eventually be observed by each stakeholder, and (ii) the local replicas that received the same updates will be in the same state.

Conflict-free replicated Data Types (CRDT) [5] provide an efficient implementation of the SEC model. While CRDT have been successful in supporting real-time collaboration over linear data types, such as text and source code, some applications require more complex data types. For example, graph models are frequently employed in Model-Driven Software Engineering (MDSE) [6]. However, the lack of appropriate extension mechanisms in current CRDT frameworks renders the definition of more complex data types a challenging task.

In this paper, we report on our experiments with extensible CRDT using our prototype framework, CollabServer[1]. The contributions include (i) a collection of CRDT primitives; (ii) an extension mechanism for the customization of CRDT; and (iii) performance assessment of the approach. We demonstrate our approach on an illustrative case for Mind map editors, which represents typical modeling environments that require graph semantics to represent models.

C. Masson is currently with Ubisoft, Paris.
[1]https://github.com/geodes-sms/collabserver-framework

II. Background

Collaborative Model-Driven Software Engineering: The challenges of distributed software engineering are substantially exacerbated by the complexity of the engineered system that necessitates collaboration between stakeholders of highly diverse expertise. Model-driven software engineering (MDSE) [6] allows stakeholders to reason at higher levels of abstraction and by that, enables aligning the work of diverse stakeholders. Combining the benefits of collaborative software engineering with MDSE, collaborative MDSE [7] has become a prominent paradigm in software engineering practice. Due to the often disparate vocabularies of stakeholders, identifying overlaps between the stakeholders' concerns is not trivial [8]. This, in turn, renders the detection of conflicts a challenging task. Recent studies [9], [10] show that only one-third of real-time collaborative MDSE solutions provide explicit conflict resolution mechanisms. State-of-the-art tools mostly rely on version control systems to facilitate collaborative MDSE [11]. Other approaches define consistency in terms of bijective correspondence, e.g., by linking elements through correspondence graphs [12], processes [13], or semantic links [14]. However, these techniques do not support real-time collaboration.

Real-time collaboration and CRDT: Sun et al. [2] define four requirements for effective real-time collaboration: (i) convergence of replicas; (ii) user intention preservation; (iii) causality preservation of updates; and (iv) timely execution of updates. CRDT support real-time collaboration by eliminating conflicts between the distributed stakeholders' operations, without the need for a costly consensus mechanism, while showcasing excellent fault tolerance and reliability properties. Notable open-source CRDT frameworks include Automerge[2] and Yjs[3]. Automerge enables real-time collaboration in JavaScript-based systems, based on the JSON format. Yjs uses linked lists as its foundational data type, but the internal representation can be extended to achieve collaboration over complex data types. However, this extension is not trivial.

III. The CollabServer Framework

A. Design principles

1) Operation-based CRDT: There are two equivalent approaches to implementing CRDT. State-based CRDT are

[2]https://github.com/automerge/automerge
[3]https://github.com/yjs/yjs

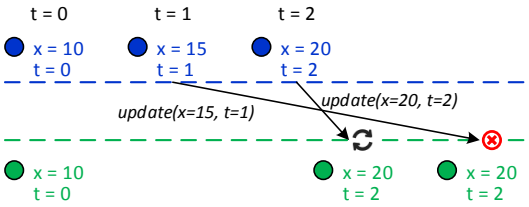

Fig. 1. Total order of updates in the LWW paradigm.

structured in a way that they adhere to a monotonic semi-lattice. Operation-based CRDT require that concurrent operations commute, i.e., irrespective of the order of operations, local replicas converge. We have opted for the operation-based CRDT scheme. This approach, as opposed to state-based CRDT, requires less network bandwidth, because only operations have to be sent through the network. In addition, an operation-based approach is more suitable for integration with external components, such as databases and user interfaces.

2) Last-Writer-Wins (LWW): Automated reconciliation between replicas can be achieved at the application level or at the data level [15]. The LWW paradigm [16] implements the latter and has been widely adopted for operation-based conflict resolution. In LWW, conflicted operations are resolved by timestamps; the data with the more recent timestamp prevails. Fig. 1 shows an example resolution scenario under LWW. User A and User B initially have their local replicas in consistent states. At $t = 1$, User A executes the update $x = 15$. The updated value and the timestamp are propagated to User B. However, before the message arrives, User A executes another update: $x = 20$, at time $t = 2$. The update is propagated to User B. Networks usually do not guarantee order preservation. Thus, the second update may arrive at User B earlier than the first. Upon receiving the update, User B will reconcile this new value with his local replica. User B has $x = 10$ timestamped with $t = 0$; and an update that says $x = 20$ timestamped with $t = 2$. Under the LWW paradigm, the latter value is accepted. Eventually, the first update reaches User B. User B has $x = 20$ timestamped with $t = 2$; and an update of $x = 15$ timestamped with $t = 1$. Under the LWW paradigm, the former value is accepted, leaving the replicas in consistent states.

3) CRDT tombstone metadata: To ensure that operations commute, data is never deleted, only flagged as removed (i.e., soft delete). This information is captured in the tombstone metadata with boolean semantics. In an alternative approach by Shapiro et al. [5], dedicated partitions of the specific datatypes are reserved for storing deleted elements (LWW-element-Set). The benefit of our approach is that it reduces the number of elementary data operations upon changes.

4) Commutativity and idempotence: Operation-based CRDT assume that operations commute (i.e., $x \circ y = y \circ x$) and are idempotent (i.e., $x \circ x = x$). Traditionally, these properties are ensured by the communication protocol [5]. We have implemented the base type system of CollabServer in a way that commutativity and idempotence are guaranteed by design. As a consequence, CRDT that extend base

TABLE I
COLLABSERVER BASIC TYPES AND THEIR METHODS

Data type	Methods
LWWRegister	`query()` `update(value, timestamp)`

LWWSet	`query(key)`
	`add(key, timestamp)`
	`clear(timestamp)`
	`remove(key, timestamp)`

LWWMap	`query(key)`
	`add(key, value, timestamp)`
	`clear(timestamp)`
	`remove(key, timestamp)`

LWWGraph	`queryVertex(vertexID)`
	`addVertex(vertexID, timestamp)`
	`removeVertex(vertexID, timestamp)`
	`addEdge(source, target, timestamp)`
	`removeEdge(source, target, timestamp)`
	`clearVertices(timestamp)`

CollabServer types are expected to satisfy these properties without further development effort. Commutativity and idempotence are achieved by the combination of *timestamps* and *tombstones*. Timestamps ensure that each replica will order the update operations in the same way. Tombstone metadata ensures that no information is lost.

B. CollabServer CRDT primitives

Table I summarizes the CRDT provided by CollabServer. Every CRDT is equipped with atomic create, read, update, and delete (CRUD) operations. More complex operations can be implemented in specific applications. CollabServer is implemented in C++, using the Standard Template Library (STL) [17]. At the source code level, CollabServer CRDT are implemented as C++ templates, allowing easy extensibility and customization. More information is available in [18] and from the GitHub repository of the project[1]. In the following, we briefly discuss the CRDT provided by CollabServer.

1) LWWRegister: The `LWWRegister` is the simplest primitive implemented in the CollabServer framework. It stores an atomic value, its timestamp ts, and its tombstone metadata. The `query` method returns the value stored in the register. The `update` method changes this value, as shown in Algorithm 1.

2) LWWSet: The `LWWSet` is a monotonically increasing data structure with the usual set semantics. That is, a value can exist in the set only once. The `LWWSet` is implemented as a C++ HashMap, with the values stored in the key set, and the associated value set storing the metadata (timestamp and tombstone). The `query` method (Algorithm 2) returns the respective key of the hashmap if it exists and is not marked as removed. The `add` method (Algorithm 3) inserts an element into the set. If the element already exists in the set, its timestamp is updated. If the element does not exist in the set, it is added to the set, along with the required metadata. The `remove` method (Algorithm 4) flags an element as deleted if its `timestamp` is higher than the current timestamp. In case the element is already deleted, its timestamp is updated, and a `false` value is returned. If the requested element is not present in the set, it is added with tombstone metadata that designates a deleted state. The `clear` method executes the `remove` method on every element in the set.

Algorithm 1: `lwwregister_update`(value, ts)

```
if value, ts > current_timestamp then
    current_value = value
    current_timestamp = value, ts
    return true
else
    return false
```

Algorithm 2: `lwwset_query`(key)

```
element = hashmap[key]
if element is not None AND is not element.value.isRemoved then
    return element.key
else
    return None
```

3) LWWMap: The `LWWMap` stores key-value pairs of data with the keys being stored in an `LWWSet` and the associated value being stored in an `LWWRegister`. By reusing the previously defined LWW types, the API methods of the `LWWMap` can be reduced to the ones defined in Algorithms 1–4.

IV. Customizing CRDT

In this section, we demonstrate the extensibility of CollabServer CRDT by (i) constructing a custom physical CRDT, the `LWWGraph` (Section IV-A); and (ii) based on this custom type, constructing a domain-specific type (Section IV-B). For the latter, we use the example of a Mind map editor, providing domain-specific operations for constructing and manipulating a Mind map, such as adding and removing topics; and placing a marker on a topic. Fig. 2 show the extended type system.

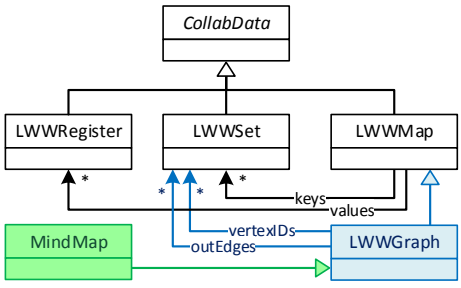

Fig. 2. Type system with the customized types highlighted

A. Constructing custom CRDT: LWWGraph

The `LWWGraph` is a directed graph, represented by its adjacency list, stored in an `LWWMap`. Vertex IDs are stored as keys and the vertices are stored as values. There are no restrictions on the type of the vertex ID, it only depends on the specific implementation, and its typing is deferred to the developer. A vertex is defined as a tuple $(content, edges)$, describing the content of the vertex and an `LWWSet` of the outgoing edges from this vertex. Each key is the ID of the target vertex. The `queryVertex` method invokes the `query` method on the `LWWMap` storing the adjacency list of vertices, and returns the vertex if exists. Similarly, the `addVertex` method invokes the

Algorithm 3: `lwwset_add`(key, ts)

```
element = hashmap[key]
if element is not None then
    if ts > element.value.timestamp then
        element.value.timestamp = ts
        if element.value.isRemoved then
            element.value.isRemoved = false
            return true
    return false
else
    element = {key, {ts, false}}
    hashmap.add(element)
    if element.value.timestamp <= lastclear_timestamp then
        element.value.timestamp = lastclear_timestamp
        element.value.isRemoved = true
        return false
    else
        return true
```

Algorithm 4: `lwwset_remove`(key, ts)

```
element = hashmap[key]
if element is not None then
    if ts > element.value.timestamp then
        element.value.timestamp = ts
        if not element.value.isRemoved then
            element.value.isRemoved = true
            return true
    return false
else
    element = {key, {ts, true}}
    hashmap.add(element)
    return false
```

`add` method on the `LWWMap` storing the adjacency list. The `removeVertex` method (Algorithm 5) removes a vertex from the graph, and all edges connected to it. If the vertex does not exist yet, it is added in the adjacency list, and the `remove` method of the `LWWMap` is invoked. The `addEdge` method (Algorithm 6) creates a new edge connecting the source vertex with the target vertex. We distinguish between three scenarios. First, we apply `addEdge` on two existing vertices. In this case, the edge is added to the underlying graph; or if the edge already exists, its timestamp is updated. Second, we apply `addEdge` when one or two vertices do not exist. This case occurs when the `addEdge` operation is observed before the `addVertex` operation. To ensure the commutativity of operations, CollabServer implements the `addEdge` method in a way that it also applies `addVertex` on the source and target vertices. Missing vertices are then simply added along with the edge. Receiving a later `addVertex` operation will simply update the timestamps. Third, we apply `addEdge` with the source and/or the target vertex that has already been deleted. This case occurs when the `addEdge` operation is received with the source and/or the target vertex already deleted. The case where `removeVertex` is older than `addEdge` is a trivial one, since `addEdge` also applies `addVertex` as seen earlier. The opposite case (`removeVertex` older than `addEdge`) requires additional steps. First, the edge is created as discussed before.

Algorithm 5: `lwwgraph_removeVertex`(vertexID,ts)

```
removed = adj.remove(vertexID, ts)
vertex = adj.queryCRDT(vertexID)
vertex.edges.clear(ts)
for vertex in adj.iteratorCRDT do
    if vertex.edges.has(vertedID) then
        vertex.edges.remove(vertexID, ts)
return removed
```

Algorithm 6: `lwwgraph_addEdge`(src, trgt, ts)

```
info = {'srcAdded': false, 'trgtAdded': false, 'edgeAdded': false}
info['srcAdded'] = adj.add(src, ts)
info['trgtAdded'] = adj.add(trgt, ts)
vertexSrc = adj.queryCRDT(src)
info['edgeAdded'] = vertexSrc.edges.add(trgt, ts)
if not vertexSrc.edges.queryCRDT(trgt).isRemoved then
    vertexTrgt = adj.queryCRDT(trgt)
    if vertexSrc.isRemoved OR vertexTrgt.isRemoved then
        t = max(vertexSrc.ts, vertexTrgt.ts)
        vertexSrc.edges.remove(trgt, t)
        info.edgeAdded = false
        return info
return info
```

Then, we check if the newly created edge is dangling and remove it. This way, `addEdge` is commutative. Note that, as shown in Algorithm 6, this operation returns more information since additional actions can be performed on the edge or the vertices. The `removeEdge` method (Algorithm 7) removes an edge from the graph. This operation may encounter a situation where the source vertex does not exist yet in the graph. Since all operations are required to be commutative, the source vertex is created with the smallest timestamp and the `isRemoved` flag is set to true. The `clearVertices` method removes all vertices and their associated edges from the graph.

B. Constructing domain-specific CRDT

The construction of custom CRDT is achieved by extending the `CollabData` base type and defining custom operations while ensuring their commutative and idempotent properties. We demonstrate the extensibility of primitive CollabServer data types by constructing the *MindMap* CRDT for the `MindmapEditor` application. The *MindMap* type is a graph; each topic and marker of the *MindMap* is a vertex of the graph; edges of the graph connect topics to their parent topic, and markers to the topics they mark. Constructing the *MindMap* requires extending the *LWWGraph* primitive type and augmenting it with domain-specific methods that are commutative and idempotent. The outline of the MindMap CRDT is shown in Listing 1. The full implementation is available from the GitHub repository of the project.[1]

The methods in Listing 1 reuse the API of the *LWWGraph*; thus, they inherit CRDT properties. The *MindMap* type extends the *LWWGraph* by introducing the notion of *attributes*, for example, for storing the *name* of the MindMap. As shown

Algorithm 7: `lwwgraph_removeEdge`(src, trgt, ts)

```
adj.remove(src, Timestamp.MIN)
if src ! = trgt then
    adj.remove(trgt, Timestamp.MIN)
vertex = adj.queryCRDT(src)
return vertex.edges.remove(trgt, ts)
```

in Listing 2, attributes are added to LWWGraph-derivatives by invoking the `add` method of the LWWMap that allows storing key-value pairs. The built-in CRDT can be readily reused to construct custom data types. This has been demonstrated in this example, and also in the definition of the *LWWMap* and *LWWGraph* that reuse more primitive CollabServer CRDT.

Listing 1. MindMap CRDT with domain-specific operations

```
class Topic: Vertex{...}
class Marker: Vertex{...}

class MindMap: LWWGraph{
  void addTopic(const UUID& topicId){
    LWWGraph::addVertex(topicId,Timestamp::now())
    notifyOperationBroadcaster()
  }
  void removeTopic(const UUID& id){}
  void addMarker(const UUID& id){}
  void removeMarker(const UUID& id){}
  void connectTopics(const UUID& t1,const UUID& t2){
    LWWGraph::addEdge(t1, t2, Timestamp::now())
    notifyOperationBroadcaster()
  }
  void putMarker(const UUID& m, const UUID& t){}
  ...
}
```

Listing 2. API for adding attributes to the MindMap

```
class MindMap: LWWGraph{
  void addAttribute(
    const UUID& id,
    const std::string& name,
    const std::string& value) {
      //calls the LWWMap super class
      LWWGraph::add(name, value, Timestamp::now())
      notifyOperationBroadcaster()
    }
}
```

V. Performance evaluation

Although performance was not our primary concern in this exploratory project, we provide a performance evaluation of the framework. As the performance of CRDT is determined by the number of objects present in the application [19], we assess the performance by simulating a scenario in which new vertices and edges are added to a shared model.

Experimental setup: We used the following sequence as a test scenario: `add topic`$_i$ → `add topic`$_j$ → `connectTopics topic`$_i$, `topic`$_j$ The scenario was executed 50.000 times. That is, 100.000 topics (graph vertices) and 50.000 relationships (edges) were generated. We have executed the test scenario with one, two, and four parallel users, and measured the change in response times. In the case of two and four parallel simulated users, each user carried out

TABLE II
RESPONSE TIMES OF THE 1/2/4 USER CASES

Users	Response times Min [ms]	μ [ms]	σ	Max [ms]
1	0.11	0.94	0.22	17.4
2	0.13	2.06	0.33	17.4
4	0.37	5.92	0.49	17.4

this sequence, adding topics (vertices) and connecting them (by adding edges) to the same shared mind map (graph). The measurements have been executed on a VMWare virtual machine running a 64-bit Ubuntu 20.04.1. OS, with 4GB of memory, and with 4 CPU cores allocated, checked at 2.6 GHz.

Results: To assess the *scalability* of the framework, response times were measured at the local replicas, defined as the time difference between issuing a command in the editor and getting a response. To filter noise, we clipped the sample at $\mu \pm 3\sigma$ (0.2% of the cases). The mean response time in the one-user case shows linear scaling with the number of objects. (Linear regression statistic: $p = 22E - 17$.) We have observed the same linear increase in response times in the two and four-user cases. We have also observed increasing response times with the increasing number of users. Table II shows the mean response time in one, two, and four user cases. The mean response time increased by a factor of 2.2 and 2.8 as the number of users doubled from one to two, and from two to four, respectively. A statistically significant difference is observed in the mean response time of the three cases, as confirmed by a t-test ($\alpha = 0.05$, p =2e-11).

We observed a linear increase in the *memory heap*. The majority of memory consumption is due to C++ node iterators and hashtable objects the `LWWGraph` relies on.

Discussion: We observe a linearly increasing response time and a linearly increasing memory footprint. This is in line with the observation of Sun et al. [20]. We conclude that this performance profile is characteristic of CRDT implementations, and can be effectively treated by suitable garbage collection mechanisms [21]. We consider these results adequate (i) considering the benefits in extensibility CollabServer CRDT provide; and (ii) considering that performance was not the primary objective of the current solution.

VI. CONCLUSION

In this paper, we presented an approach for augmenting CRDT with extension mechanisms and demonstrated that the performance repercussions of extensibility are manageable. We provided a family of concurrency control algorithms, ensuring strong eventual consistency and allowing for efficient real-time collaboration. Our algorithms and data types show linear scaling of response time and memory footprint with the number of objects in memory and with users. This is a characteristic performance profile of CRDT. Our results suggest that CRDT can be used in disciplines where customizability is a key factor, such as collaborative modeling using domain-specific modeling languages. In future work, we will investigate garbage collection mechanisms to achieve the scalability collaborative engineering tools require. We used the takeaways of this exploratory project in the development of our real-time collaborative modeling framework lowkey [22].

REFERENCES

[1] J. Whitehead, "Collaboration in software engineering: A roadmap," in *Future of Software Engineering*. IEEE, 2007. doi: 10.1007/978-3-642-10294-3_1 pp. 214–225.

[2] C. Sun *et al.*, "Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, p. 63–108, 1998.

[3] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, 1981. doi: 10.1145/356842.356846

[4] V. B. Gomes *et al.*, "Verifying strong eventual consistency in distributed systems," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017. doi: 10.1145/3133933

[5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Symposium on Self-Stabilizing Systems*. Springer, 2011. doi: 10.1007/978-3-642-24550-3_29 pp. 386–400.

[6] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006. doi: 10.1109/MC.2006.58

[7] H. Muccini, J. Bosch, and A. van der Hoek, "Collaborative modeling in software engineering," *IEEE Software*, vol. 35, no. 6, pp. 20–24, 2018.

[8] I. David *et al.*, "Engineering process transformation to manage (in)consistency," in *Proceedings of the 1st Intl. Workshop on Collaborative Modelling in MDE*, vol. 1717. CEUR-WS.org, 2016, pp. 7–16.

[9] ——, "Collaborative Model-Driven Software Engineering: A Systematic Update," in *Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. ACM, 2021. doi: 10.1109/MODELS50736.2021.00035 pp. 273–284.

[10] M. Franzago, D. D. Ruscio, I. Malavolta, and H. Muccini, "Collaborative model-driven software engineering: A classification framework and a research map," vol. 44, no. 12, 2018. doi: 10.1109/TSE.2017.2755039 pp. 1146–1175.

[11] S. Kelly, "Collaborative modelling with version control," in *Federation of Intl. Conferences on Software Technologies: Applications and Foundations*. Springer, 2017. doi: 10.1007/978-3-319-74730-9_3 pp. 20–29.

[12] C. Adourian and H. Vangheluwe, "Consistency between geometric and dynamic views of a mechanical system," in *Proceedings of the 2007 Summer Computer Simulation Conference*. Society for Computer Simulation International, 2007.

[13] I. David *et al.*, "Towards inconsistency management by process-oriented dependency modeling," in *Proceedings of the 9th International Workshop on Multi-Paradigm Modeling, 2015*, ser. CEUR Workshop Proceedings, vol. 1511. CEUR-WS.org, 2015, pp. 32–41.

[14] K. Vanherpen *et al.*, "Ontological reasoning for consistency in the design of cyber-physical systems," in *1st International Workshop on Cyber-Physical Production Systems*. IEEE, 2016, pp. 1–8.

[15] C. Meiklejohn and P. Van Roy, "Lasp: A language for distributed, coordination-free programming," in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. ACM, 2015. doi: 10.1145/2790449.2790525 p. 184–195.

[16] P. R. Johnson and R. Thomas, *RFC0677: Maintenance of duplicate databases*. RFC Editor, 1975.

[17] A. Stepanov and M. Lee, *The standard template library*. HP Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995, vol. 1501.

[18] C. Masson, "Framework for Real-time collaboration on extensive Data Types using Strong Eventual Consistency," Master's thesis, Université de Montréal, Canada, December 2018.

[19] D. Sun and C. Sun, "Operation Context and Context-based Operational Transformation," in *Proceedings of the 2006 20th Conference on Computer Supported Cooperative Work*. ACM, 2006. doi: 10.1145/1180875.1180918 pp. 279–288.

[20] D. Sun, C. Sun, A. Ng, and W. Cai, "Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors," *Proc. ACM Hum.-Comput. Interact.*, vol. 4, 2020.

[21] J. Bauwens and E. Gonzalez Boix, "Memory Efficient CRDTs in Dynamic Environments," in *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL 2019. ACM, 2019. doi: 10.1145/3358504.3361231 p. 48–57.

[22] I. David and E. Syriani, "Real-time Collaborative Multi-Level Modeling by Conflict-Free Replicated Data Types," *Softw. Syst. Model.*, 2022.