# Architectures and Algorithms for Real-Time Web-Based Collaboration

**Cristian Gadea**

A thesis submitted in partial fulfillment of the requirements for the
Doctorate in Philosophy
degree in
Electrical and Computer Engineering

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

*"Great things are done by a series of small things brought together."*

Vincent Van Gogh

# *Abstract*

Originating in the theory of distributed computing, the optimistic consistency control method known as Operational Transformation (OT) has been studied by researchers since the late 1980s. Algorithms were devised for managing the concurrent nature of user actions and for maintaining the consistency of replicated data as changes are introduced by multiple geographically-distributed users in real-time. Web-Based Collaborative Platforms are now essential components of modern organizations, with real-time protocols and standards such as WebSocket enabling the development of online collaboration tools to facilitate information sharing, content creation, document management, audio and video streaming, and communication among team members. Products such as Google Docs have shown that centralized web-based co-editing is now possible in a reliable way, with benefits in user productivity and efficiency. However, as the demand for effective real-time collaboration between team members continues to increase, web applications require new synchronization algorithms and architectures to resolve the editing conflicts that may appear when multiple individuals are modifying the same data at the same time. In addition, collaborative applications need to be supported by scalable distributed backend services, as can be achieved with "serverless" technologies. While much existing research has addressed problems of optimistic consistency maintenance, previous approaches have not focused on capturing the dynamic client-server interactions of OT systems by modeling them as real-time systems using Finite State Machine (FSM) theory. This thesis includes an exploration of how the principles of control theory and hierarchical FSMs can be applied to model the distributed system behavior when processing and transforming HTML DOM changes initiated by multiple concurrent users. The FSM-based OT implementation is simulated, including with random inputs, and the approach is shown to be invaluable for organizing the algorithms required for synchronizing complex data structures. The real-time feedback control mechanism is used to develop a Web-Based Collaborative Platform based on a new OT integration algorithm and architecture that brings "Virtual DOM" concepts together with state-of-the-art OT principles to enable the next generation of collaborative web-based experiences, as shown with implementations of a rich-text editor and a 3D virtual environment.

# *Acknowledgements*

This thesis was prepared at the Network Communications and Control Technologies (NCCT) Laboratory of the University of Ottawa. I am grateful for the openness and guidance of my supervisor, Dr. Dan Ionescu, who always encouraged me to tackle new challenges and never hesitated to offer help. His input and experience was invaluable throughout the realization of this work.

This thesis would not have been possible without the generous assistance of other members of the NCCT Lab. Their depth of knowledge and devotion to technical research created a positive working atmosphere that motivated me to learn about many new technologies. Bogdan Ionescu, Bogdan Solomon, Mircea Trifan, Laurentiu Checiu, Juliano Garcia and Daniel Hong all contributed to the projects mentioned in this thesis.

Finally, I would like to thank my parents and my brother for their continuous love, support and encouragement throughout this entire journey.

# Publications

Content from this thesis has previously appeared in the following publications:

## Refereed Journal and Conference Papers

1. C. Gadea, B. Ionescu, and D. Ionescu, "A Finite State Machine Approach to Collaborative Computing," in Transactions on Parallel and Distributed Systems. IEEE Computer Society, 2021, submitted.

2. _____, "A Control Loop-based Algorithm for Operational Transformation," in *SACI 2020: Proc. of 14th Int. Symp. on Applied Computational Intelligence and Informatics*. IEEE Computer Society, 2020, pp. 247-254.

3. _____, "New Algorithms and Methods for Collaborative Co-Editing Using HTML DOM Synchronization," in *CIC 2018: Proc. of 4th IEEE Int. Conf. on Collaboration and Internet Computing*. IEEE Computer Society, 2018, pp. 217-226.

4. _____, "A Hybrid FSM Rule-Based Approach for the Real-Time Control of Web-Based Collaborative Platforms," in *INES 2018: Proc. of 22nd Int. Conf. on Intelligent Engineering Systems*. IEEE Computer Society, 2018, pp. 27-32.

5. _____, "Modeling and Simulation of an Operational Transformation Algorithm using Finite State Machines," in *SACI 2018: Proc. of 12th Int. Symp. on Applied Computational Intelligence and Informatics*. IEEE Computer Society, 2018, pp. 119-124.

6. C. Gadea, D. Hong, D. Ionescu, and B. Ionescu, "An Architecture for Web-based Collaborative 3D Virtual Spaces using DOM Synchronization," in

*CIVEMSA 2016: Proc. of Int. Conf. on Computational Intelligence and Virtual Environments for Measurement Systems and Applications*. IEEE Computer Society, 2016, pp. 16.

7. C. Gadea, M. Trifan, D. Ionescu, M. Cordea, and B. Ionescu, "A Microservices Architecture for Collaborative Document Editing Enhanced with Face Recognition," in *SACI 2016: Proc. of 11th Int. Symp. on Applied Computational Intelligence and Informatics*. IEEE Computer Society, 2016, pp. 441-446.

8. C. Gadea, M. Trifan, D. Ionescu, and B. Ionescu, "A Reference Architecture for Real-Time Microservice API Consumption," in *Proc. of the 3rd Workshop on CrossCloud Infrastructures & Platforms*. ACM, 2016, p. 2.

9. B. Ionescu, D. Ionescu, C. Gadea, B. Solomon, and M. Trifan, "An Architecture and Methods for Big Data Analysis," in *SOFA 2014: Proc. of the 6th Int. Workshop Soft Computing Applications*, Advances in Intelligent Systems and Computing, vol 356. Springer International Publishing, 2016, pp. 491-514.

10. B. Ionescu, C. Gadea, B. Solomon, M. Trifan, D. Ionescu, and V. Stoicu-Tivadar, "A Chat-Centric Collaborative Environment for Web-Based Real-Time Collaboration," in *SACI 2015: IEEE 10th Jubilee Int. Symp. on Applied Computational Intelligence and Informatics*. IEEE Computer Society, 2015, pp. 105-110.

11. B. Ionescu, C. Gadea, B. Solomon, D. Ionescu, V. Stoicu-Tivadar, and M. Trifan, "A Cloud Based Real-Time Collaborative Platform for eHealth," *Studies in Health Technology and Informatics*, vol. 210, p. 919-923, 2015.

# Book Chapters

1. B. Solomon, D. Ionescu, C. Gadea, and M. Litoiu, *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, November 2012, ch. Geographically Distributed Cloud-Based Collaborative Application.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Listings

# Abbreviations

| | |
|---|---|
| **AJAX** | **A**synchronous **J**avaScript **A**nd **X**ML |
| **API** | **A**pplication **P**rogramming **I**nterface |
| **ASCII** | **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange |
| **BaaS** | **B**ackend-**a**s-**a**-**S**ervice |
| **CCCP** | **C**ommon **C**ollaborative **C**oding **P**rotocol |
| **CLOT** | **C**ontrol **L**oop-based **O**perational **T**ransformation algorithm |
| **COT** | **C**ontext-based **O**perational **T**ransformation algorithm |
| **CmRDT** | **C**ommutative **R**eplicated **D**ata **T**ype |
| **CRDT** | **C**onflict-free **R**eplicated **D**ata **T**ype |
| **CSCW** | **C**omputer **S**upported **C**ooperative **W**ork |
| **CSS** | **C**ascading **S**tyle **S**heets |
| **DOM** | **D**ocument **O**bject **M**odel |
| **dOPT** | **d**istributed **OP**erational **T**ransformation algorithm |
| **FDO** | **F**eedback **D**OM **O**bserver |
| **FSM** | **F**inite **S**tate **M**achine |
| **GROVE** | **GR**oup **O**utline **V**iewing **E**dit |
| **GWT** | **G**oogle **W**eb **T**oolkit |
| **HMD** | **H**ead **M**ounted **D**isplay |
| **HTML** | **H**yper **T**ext **M**arkup **L**anguage |
| **IBM** | **I**nternational **B**usiness **M**achines Corporation |
| **IDE** | **I**ntegrated **D**evelopment **E**nvironment |
| **JSON** | **J**ava**S**cript **O**bject **N**otation |
| **NCCT** | **N**etwork **C**omputing and **C**ontrol **T**echnologies |
| **NICE** | **N**otification-flex**I**ble **C**ollaborative **E**diting algorithm |

| | |
|---|---|
| **NoSQL** | **No S**tructured **Q**uery **L**anguage |
| **OT** | **O**perational **T**ransformation |
| **RSAD** | **R**ational **S**oftware **A**rchitect **D**esigner |
| **SAP** | **S**ystems, **A**pplications & **P**roducts in Data Processing |
| **SIP** | **S**ession **I**nitiation **P**rotocol |
| **TCP** | **T**ransmission **C**ontrol **P**rotocol |
| **TIBOT** | **T**ime **I**nterval **B**ased **O**perational **T**ransformation |
| **TP** | **T**ransformation **P**roperty |
| **TS** | **T**ime**S**tamp |
| **UAL** | **U**ML **A**ction **L**anguage |
| **UML** | **U**nified **M**odeling **L**anguage |
| **URL** | **U**niform **R**esource **L**ocator |
| **VDOM** | **V**irtual **D**ocument **O**bject **M**odel |
| **VR** | **V**irtual **R**eality |
| **W3C** | **W**orld **W**ide **W**eb **C**onsortium |
| **WebGL** | **W**eb **G**raphics **L**ibrary |
| **WYSIWYG** | **W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et |
| **XML** | e**X**tensible **M**arkup **L**anguage |

*Dedicated to my parents and my brother*

# Chapter 1

# Introduction

## 1.1 Motivation & Research Objectives

The World Wide Web now reaches billions of users who regularly rely on social collaboration products such as Wikipedia, Twitter, Facebook, and others. Once capable of hosting only simple static HTML resources, the web has now become a platform for complex interactive web applications. Users of the modern web expect the ability to share and collaborate with other people, such as friends and colleagues, online and in real-time. At the enterprise level, social networking products and virtual environments are allowing for more efficient integration of remote team members whose work has to remain synchronized with that of their colleagues. The 2020 coronavirus pandemic has highlighted the importance of this technology, as the modern workplace now increasingly depends on web-based team collaboration platforms and social enterprise tools. A recent Gallup poll showed that 59 percent of workers hope to continue working from home as much as possible even after the pandemic is under control [1].

Real-time groupware allows multiple geographically-dispersed users to be simultaneously engaged in accomplishing common tasks or goals by modifying parts of the same object or document at the same time through an interactive shared environment [2]. At the heart of real-time groupware is a set of algorithms that support a range of functions related to concurrency control of the user's actions while editing a document, and to the maintenance of the consistency of the document. Users may be residing in different geographical locations and observing high network latency as document changes are propagated between them in the

form of *operations*. The desired co-editing algorithms must avoid client usability constraints such as locked or restricted portions of the user interface while ensuring that document replicas at all user sites remain synchronized in the face of editing conflicts. Conflicts between operations must be resolved without user intervention when two or more users working in parallel change the same portion of a document. In order to ensure a high level of responsiveness for the local user's edits while also accommodating high network latencies, received operations must be *transformed* against any local operations before being applied, as the received operations may not yet account for the immediately-applied local changes. This is achieved through the use of an optimistic consistency control method such as Commutative Replicated Data Types (CmRDTs) or Operational Transformations (OT), the latter of which will be the focus of this thesis.

Known as the *dOPT* algorithm, the first OT algorithm was pioneered by Ellis and Gibbs [3] in 1989 and used a peer-to-peer approach with operations exchanged directly between clients. Their seminal work identified many of the present key requirements and challenges of OT, including real-time convergence in high latency environments where concurrent users must always be able to interact with their local text and see their own changes applied immediately (OT algorithms are therefore *optimistic*). By building on the work of Ellis and Gibbs, a *consistency model* for collaborative editing systems was identified by Sun et al. [4] as requiring *causality preservation*, *convergence*, and *intention preservation*.

The *dOPT* algorithm, however, contained a flaw now known as the *dOPT puzzle*, and early OT research focused on solving such consistency maintenance challenges for plaintext co-editing [5]. Several solutions were identified that make use of a centralized server component for ordering and broadcasting operations [6][7][8][9]. Such solutions are now of particular interest for their applicability to web-based architectures, with the objective of maintaining the consistency of document replicas within each user's local browser. Four users co-editing a document through a centralized server can be represented with the basic architecture shown in Figure 1.1. In addition to the replica at each user's site, the Jupiter algorithm of Nichols et al. [6] explicitly requires a central transformation-based server to maintain its own replica of the shared document.

Today's modern web applications are increasingly adopting "serverless" architectures [10]. In this thesis, *serverless* implies that a server exists but does not require

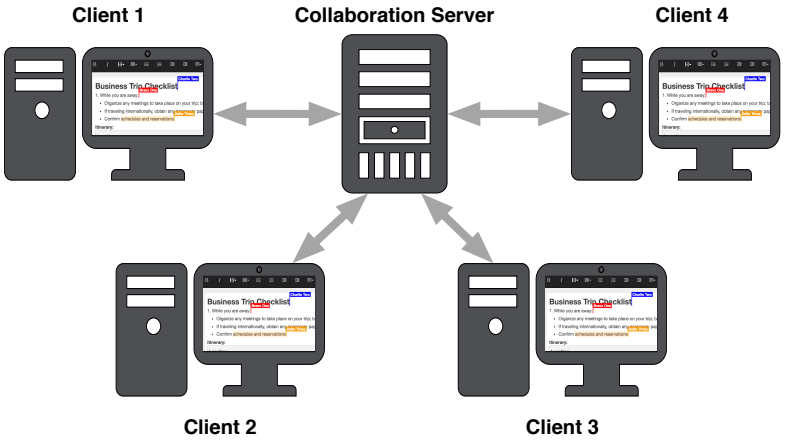



FIGURE 1.1: Centralized architecture with four clients collaboratively editing the same document.

attention for tasks such as resource allocation and management, thereby simplifying cloud-based scalability and deployment. More specifically, *serverless database* providers such as Firebase offer hierarchical key-value storage services with real-time broadcasting features [11]. However, such services are typically limited to read/write-based API functions and generally do not permit custom server-side logic such as the transformation functions required by Jupiter. The SOCT4 OT algorithm [7] revealed that a simplification of the server-side OT logic is possible through the use of a *Ticket* function to establish a continuous global order of operations, thereby enabling transformation functions to reside exclusively on the client side. As will be presented in this thesis, an algorithm that further simplifies such approaches is required to enable "serverless" deployments for browser-based real-time collaboration.

The evolution of web communication technologies and standards such as AJAX, WebSocket and WebRTC [12][13] have now made it possible to implement sophisticated concurrency control methodologies as part of web applications accessible from a regular web browser. A popular example of such an implementation is Google Docs, an online collaborative tool for editing documents in real-time [14]. Based on the centralized OT approach of Nichols et al. [6], Google Docs allows documents to be opened and edited by multiple users at the same time, where users are able to see character-by-character changes appear as other participants make edits to the same document. It is now part of the "Google Workspace" (formerly "G Suite") [15] brand of products, along with other real-time groupware tools such as Google Sheets, Google Slides and Google Drawings. A 2017 survey by BetterCloud Monitor found that Google Workspace administrators observed an

"increase in collaboration, efficiency, flexibility, mobility, and productivity among their workforce" [16], showing how organizations have seen significant benefits and competitive advantages by allowing documents such as technical reports, project plans, work proposals, team presentations, scientific publications, and financial spreadsheets to be edited by multiple users at the same time. Figure 1.2 shows a Google Trends graph that indicates the search interest for the term "google docs" between January 2006 and October 2020 [17]. The large dips in search volume occurring every summer are likely connected to the product's popularity among students, but the general upward trend is noteworthy as it implies healthy future demand for such online collaborative technologies.

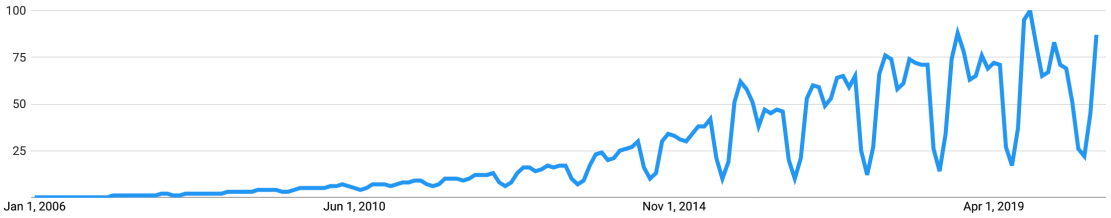



FIGURE 1.2: Google Trends graph of interest for the search term "google docs" from January 1, 2006 to October 1, 2020.

In early user feedback of their collaborative text editor *GROVE* (GRoup Outline Viewing Edit), Ellis et al. [2] observed in 1991 that "once one has experienced the flexibility and support provided by a groupware tool, one wants groupware features in all tools". However, most of today's existing wikis, content management systems, enterprise social networking platforms, and project management tools (all of which will henceforth be referred to as *content environments*) still allow only a single user to edit a document at a time. If a real-time breaking news event, meeting, or looming deadline requires multiple colleagues to work on the same document at the same time, the users are likely to encounter "locked document" errors and take turns closing the document so that it becomes available for the next person to edit. Alternatively, they are presented with complicated user interfaces for manually merging changes.

Figure 1.3 shows a collection of unpleasant error messages which hinder content creation and user adoption for products such as MediaWiki [18], XWiki [19] and Jive [20]. Since users are already familiar with the tools and their corresponding content editors, they are often reluctant to change this inefficient workflow.

The WordPress content management system and blogging platform [21] is now being used by 40 percent of websites on the Internet [22], and almost 1500 blog posts

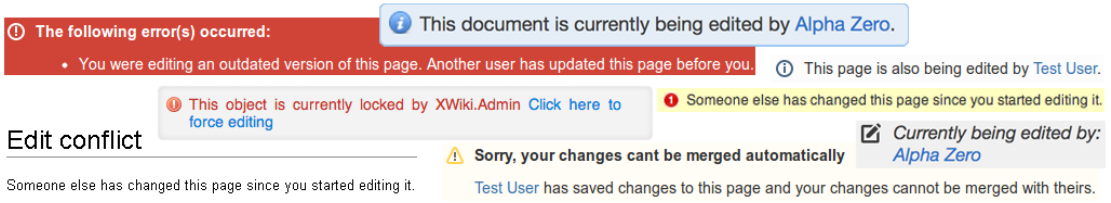



FIGURE 1.3: A collection of conflict-related error messages from various content environments across the Internet.

were being written per minute in 2016 [23]. Since a content environment such as WordPress can benefit from co-editing, there is a great demand for a framework that allows any existing rich-text editor and web application across the Internet to include (or be enhanced with) real-time co-editing functionality. Content environments typically rely on third-party "what you see is what you get" (WYSIWYG) editors to enable their content creation capabilities. For example, when examining popular blogging platforms, it was observed that WordPress and Joomla [24] make use of the TinyMCE editor [25], while Drupal [26] uses CKEditor [27]. The main purpose of reusable open source editing components such as TinyMCE and CKEditor is to produce the document as HTML content in the background while presenting the user with a familiar user interface that is not dissimilar from the user interface of popular desktop applications such as Microsoft Word. The content environment typically allows the user to modify the body of the editor, and when the user has finished, a "Save" function is normally invoked. At this point, the HTML contents of the editor are submitted to the content environment's application server and persisted in a database for later retrieval. While content environments may customize their editors to add or remove functionality (both TinyMCE and CKEditor support plugins), or change settings such as the available toolbar buttons, the editor generally still produces HTML content to be stored in a database.

Products such as Google Docs were designed to generate their own proprietary HTML Document Object Model (DOM) using a secondary annotation-based data model that simplifies the concurrency control problem [28]. A different approach is needed for a generalized solution. In addition, organizations may wish to retain full control over their internal documents and may be governed by data policies (confidentiality, compliance, retention etc.) that prevent the use of products such as Google Docs. The solution proposed in this thesis allows existing WYSIWYG editors (and other applications that modify the DOM) to create the document content as they always have, but introduces a layer that monitors the area of

the DOM in which the editor operates. The editor itself does not need to be modified, thereby eliminating the need to understand the tens of thousands of lines of code that editors such as TinyMCE and CKEditor typically consist of. In previous research, this non-invasive and reusable technique has been referred to as *transparent adaptation* [29] or *single-user to multi-user transformation* [30][31][32].

In this thesis, a feedback-based architecture is proposed that uses Finite State Machines (FSMs) to model the dynamic nature of a new client-server Operational Transformation (OT) algorithm designed for "serverless" web applications. The proposed Control Loop-based OT integration algorithm (CLOT) does not require server-side transformations or ticket-based sequencing in order to serialize operations. In the proposed approach, client sites retry to send transformed versions of local operations until they are acknowledged as accepted by the central controller. It will be shown how FSMs and a novel Virtual DOM technique can help manage the consistency maintenance and intention preservation challenges of an OT system. By modeling the real-time dynamic behavior using FSMs, a centralized OT system can be simulated step-by-step as additional operations such as node splitting are added to enable a more complete DOM-based synchronization than previous approaches. The real-time behavior is simulated in MATLAB, a widely used environment for algorithm development and prototyping. As many changes as possible that can be enacted on a standards-compliant DOM are determined based on the November 2015 DOM4 Recommendation [33] of the W3C [34] and are represented as operations. DOM-based synchronization is then implemented as a real-time control loop that conveys operations as feedback to clients, building on established OT techniques to allow HTML document contents to converge into a consistent state while preserving more user intentions than previous approaches. The algorithms are evaluated using a collaborative rich-text editor and a multi-user 3D virtual environment that were implemented on top of a Web-Based Collaborative Platform derived from the resulting feedback-loop architecture.

## 1.2 Thesis Contributions

This thesis contains a number of contributions to the fields of distributed systems and real-time collaborative systems, parts of which were previously presented in [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], and [45].

1. An architecture for distributed consistency control algorithms is proposed for DOM-based synchronization, implemented as a real-time control loop conveying operations as feedback to clients' disturbances. The architecture consists of a *Feedback DOM Observer* and a *Comparison Logic* block for DOM change detection, a *Client Content Processor* for DOM manipulation, and a *Decision Maker* containing conflict resolution logic. Finally, a *Central Controller* is used for enforcing unique timestamps, for publisher/subscriber brokering, as well as for optional key-value persistence functionality.

2. A Control Loop-based OT integration algorithm (CLOT) is introduced that makes use of "Virtual DOM" concepts and an FSM-based structure to ensure that new DOM-based operations are synchronized and transformed correctly. The Finite State Machine (FSM) approach is used to capture the dynamic client-server interactions of OT systems by modeling them based on the proposed real-time system architecture. Hierarchical FSMs are devised and employed to describe and model real-time behaviors of client and server components when generating, transforming, and applying the changes initiated by users. By applying FSM state space representations, users are shown to reach a consistent state even with some of the previously-mentioned complex DOM-based operations. While OT algorithms have been analyzed in the past using techniques such as formal algebraic proofs [46][47] and theorem provers [48], FSM-based approaches to OT integration algorithms have not been explored in previous academic or industrial publications. Where possible, state tables are used along with an extensive test suite and a random events generation component in order to demonstrate the validity of the algorithms.

3. A unique and generalized set of ten operations required for synchronization of DOM nodes (namely, element nodes and text nodes) was devised by studying the types of possible changes inline with a subset of the W3C DOM4 specification [33]. The FSM representation led to a minimal set of operations that allow for more elaborate transformations for content synchronization than any other known existing work in this field. Operations such as splitting, merging and moving of nodes, as well as operations that affect element node attributes, were determined to be necessary. From the observed patterns, the FSM models were built and a set of operations was derived for representing changes to the graph-based structure of the DOM.

Moreover, the operations obtained can be transmitted between concurrent users to maintain a consistent state across all users.

4. A Web-Based Collaborative Platform is devised such that new or existing content environments and web applications can make use of the platform's DOM synchronization functionality to create new collaborative tools and experiences. Since the proposed OT algorithm does not require any server-side transformations, the Web-Based Collaborative Platform is ideal for modern scalable "serverless" deployments [10]. Two sample applications were designed and experimented with in order to demonstrate the platform's reusability and evaluate its real-time performance. At first, a collaborative rich-text document editor was developed that includes word processing functionality commonly found in products such as Google Docs. To show that the architecture and platform have implications far beyond office applications, a collaborative 3D environment was then developed that allows multiple users to generate objects in a persistent multiplayer world. Both projects were created with the assistance of other researchers at the NCCT Lab.

## 1.3 Thesis Methodology & Organization

The methodology used to develop this thesis is based on design-oriented research [49]. The need for a Web-Based Collaborative Platform that enables simultaneous editing of HTML DOM content is first identified as a gap in existing research. To automatically resolve the conflicts that occur during such editing, the Control Loop-based OT integration algorithm (CLOT) is introduced by developing a real-time architecture based on detailed requirements. New operations and conflict resolution algorithms are defined using FSMs to ensure that all changes made to a user's local DOM can be transmitted to the other collaborators such that a consistent state is reached across all users. The components of the architecture are modeled using an iterative approach and evaluated with modern FSM simulation tools to ensure the causality, convergence, and intention preservation requirements of the OT system are met in several representative co-editing test scenarios. To address the identified functionality gaps in existing literature, a "serverless" implementation of the architecture is presented as the Web-Based Collaborative Platform and demonstrated with two different collaborative web applications.

The remainder of this thesis is organized as follows:

Chapter 2 provides the necessary background and identifies various related work and relevant literature from the domain of real-time co-editing technologies. It discusses existing collaborative platforms and state-of-the-art optimistic consistency control algorithms, including the importance of designing operations that preserve the user's intentions.

Chapter 3 derives a set of requirements for the proposed architecture and describes the high-level design. The requirements are mapped to components of a real-time feedback control system and initial models using FSMs are developed around well-established OT concepts.

Chapter 4 shows low-level details of the new CLOT algorithm and how the high-level FSM-based OT model of Chapter 3 was extended to support a "serverless" design.

Chapter 5 presents a set of operations for tree-based synchronization of DOM nodes and develops the necessary algorithms in order to support synchronization of DOM content using OT, including with more detailed FSM models.

Chapter 6 uses an advanced modeling tool to build a distributed system that adopts the proposed architecture and algorithms. For the first time, FSM models are developed that allow simulations of OT systems to be observed and evaluated.

Chapter 7 provides the results obtained after executing the simulations of the proposed design. The modeled system supports the execution and simulation of the proposed control loop architecture and FSM-based OT algorithms to ensure that they can be used as a solid foundation for the Web-Based Collaborative Platform. Integration tests and randomized inputs are used to ensure the robustness of the algorithms and that the causality, convergence, and intention preservation requirements are met.

Chapter 8 describes the design and implementation of the Web-Based Collaborative Platform. The reusability of the platform for new types of collaborative experiences is emphasized, as the platform's robust DOM synchronization capability enables many next-generation web applications to be developed. This will be shown using a rich-text co-editing application and a multi-user 3D virtual environment.

Finally, Chapter 9 presents conclusions on the benefits provided by the proposed real-time architecture. Future directions for the architecture and novel uses of the Web-Based Collaborative Platform are also described.

# Chapter 2

# Background & Related Work

Conflict resolution algorithms for real-time co-editing have been an important area of research since the eighties, and their development continues to be essential with growing demand for online team collaboration software that facilitates the completion of tasks as a group. This chapter presents the relevant background related to co-editing technologies and real-time synchronization techniques with a focus on approaches applicable to web-based collaboration and browser-based environments.

## 2.1 Groupware

The multidisciplinary research field of *Computer Supported Cooperative Work* (CSCW) aims to use technology to help users work together in shared time and space [50]. In their 1991 article entitled "Groupware: Some Issues and Experiences", Ellis et al. [2] identified *groupware* as a class of applications within CSCW. They defined *groupware* as follows:

> "computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment."

Figure 2.1 presents one method of classifying groupware systems as inspired by adaptations [2][51] of the "Time Space Matrix" of Johansen [52]. The figure shows

how collaborative applications can be categorized in terms of the locations of the collaborators and the speed of interaction between them. In the vertical *time* axis, real-time *strongly synchronous* interaction is differentiated from non-real-time *weakly synchronous* (asynchronous) interaction. The horizontal *space* axis identifies long-distance collaborations with isolated users as *strongly distributed* and co-located interactions as *weakly distributed*. This thesis will focus on the *synchronous distributed interaction* quadrant denoted by the dashed rectangle in the top right corner of Figure 2.1, whereby geographically-dispersed users are interacting with the groupware application at the same time in a real-time co-editing scenario. While it may be possible to use such an editing application from the same location and even at different times, the algorithms discussed in this thesis address co-editing conflicts and consistency maintenance challenges that are more likely to appear when the multi-user collaboration is *strongly synchronous* and *strongly distributed*.

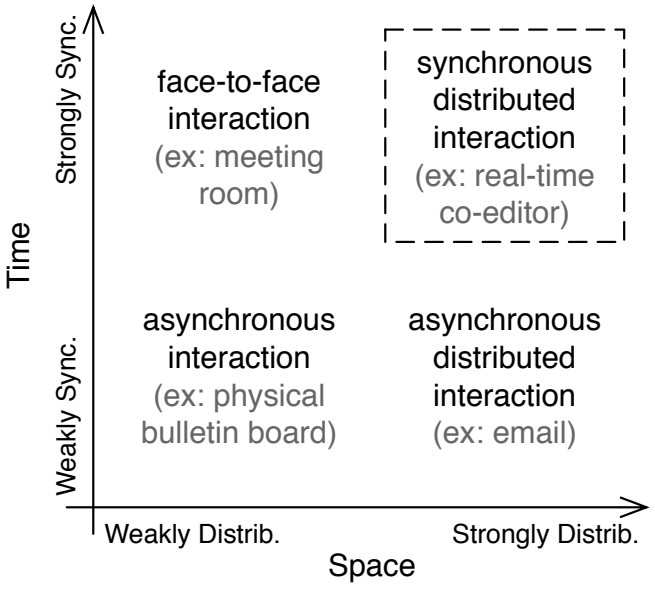



FIGURE 2.1: Classifications of Groupware Systems.

It is worthwhile to note that the word "collaborative" has been used in various online marketing materials to denote functionality ranging from basic email sending and message posting to more advanced real-time team chatting (the latter of which is offered by products such as Slack [53]). The work presented in this thesis will focus on real-time collaboration solutions that require conflict resolution algorithms between multiple concurrent users rather than the simpler types of data sharing often branded as "collaboration".

## 2.2 Optimistic Consistency Control

The design and implementation of optimistic consistency control algorithms for real-time collaboration presents significant challenges, and numerous technical and theoretical solutions have evolved over more than three decades of co-editing research. As this section will explore, the resulting research has identified two main families of approaches for optimistic consistency control with intention preservation: Operational Transformations (OT) and Commutative Replicated Data Types (CmRDTs).

### 2.2.1 Operational Transformations

Early research into collaborative text editors identified the need for a replicated architecture where the contents of a shared document are simultaneously edited locally at each site [54]. With the introduction of the *dOPT* algorithm and *GROVE* collaborative text editor in 1989, Ellis and Gibbs [3] highlighted the importance for users to immediately see the effects of their actions applied to their local text editor, thereby ensuring that the user interface feels responsive like that of a single-user editor. Once the local replica is updated, changes made to a document must be propagated as *operations* to remote users for integration into their replicas. Such propagation must accommodate high communication latency that may be exhibited by networks between participating users. A concurrency control algorithm is therefore required to maintain data consistency between multiple client sites where local changes may appear at any time.

Before making changes to data, traditional concurrency control algorithms obtain locks based on round-trip communication with a central coordinator or other sites, thereby avoiding simultaneous conflicting operations [6]. Since such *pessimistic* algorithms would impair the desired local interaction requirements, an *optimistic* replication approach was employed as part of the *dOPT* algorithm. Also known as *eventual consistency*, this approach allows replicas of shared objects to diverge temporarily while guaranteeing that they will eventually converge to a globally consistent state as long as all messages are delivered. However, conflicting updates may occur when permitting concurrent operations in this fashion, and such conflicts must be rectified using a reconciliation strategy. The *dOPT* algorithm

identifies how received operations can be *transformed* against local operations before being applied to the local replica, as the received operations may need to be adjusted to compensate for the previously-applied local changes. That is, a received remote operation $O$ is transformed into a modified version $O'$ based on the local concurrent operations such that applying $O'$ on the local replica produces the same document as the one obtained by applying $O$ on the original remote replica.

In addition to this requirement for *convergence*, a requirement for *causality preservation* was also proposed by Ellis and Gibbs [3]. A global time ordering (serialization) technique [55] was employed to identify concurrent operations from multiple sites and ensure that operations are applied in the correct order. A third requirement known as *intention preservation*, which focuses on *operation intention* and *user intention*, was later identified by Sun et al. [56] to ensure that the effect of applying (executing) an operation on a remote replica matches the intention of the operation when it was applied to the local replica. Together, the requirements of convergence, causality preservation and intention preservation form a well-accepted *consistency model* for collaborative editing systems [4], as will be explored in this section. The described class of concurrency control algorithms designed to reach consistency in the presence of concurrent user actions was named Operational Transformations (OT).

#### 2.2.1.1 "Causality Preservation" Requirement

Two operations are linked by a *causal relation* when an operation has been influenced by a previously-occurring operation. For example, if three users are co-editing a document and the first user types a word that the second user then adds a letter to, the third user should not see the added letter appear in their document before seeing the word. Operations therefore need to be executed in their *causal-effect* order [4] so that the effects of the operations are observed in the order that makes sense to all users. It is also possible that two operations follow each other but don't affect each other (and are therefore not *causally dependent*), such as when each operation affects a different leaf node (e.g. paragraph) of a tree-based document (e.g. document based on HTML DOM).

In their definition of correctness in groupware systems, Ellis and Gibbs [3] identified the causality requirement as the *Precedence Property* and defined it as follows:

> The *Precedence Property* states that if one operation, $o$, precedes another, $p$, then at each site the execution of $o$ *happens before* the execution of $p$.

To better understand this definition and what is meant by *happens before*, it is necessary to revisit the 1978 work of Leslie Lamport [55]. In this seminal paper, Lamport showed that while it is not possible to synchronize physical clocks perfectly across a distributed system, the distinct patterns of events in distributed systems, such as the presence of a forward-moving sequence when in the same process or the path taken by a message from its sending to its reception, can still reveal ordering. Namely, operation $O_a$ is said to have *happened before* operation $O_b$ if $O_a$ was generated before $O_b$ at the same site, or if $O_b$ was generated at a site after $O_a$ was received at that site, or, lastly, if there exists an operation $O_x$ such that $O_a$ *happened before* $O_x$ and $O_x$ *happened before* $O_b$ (transitivity). In this way, a *partial order relation* is established between the operations, and two operations that are not in such a relation are said to be *concurrent*. Lamport's definition used the concept of "process" instead of "site" and used "event" instead of "operation", but the words are used interchangeably here.

Figure 2.2 was adapted from Coulouris et al. [57] to illustrate Lamport's technique for basic ordering in a distributed system. With time increasing when moving down the vertical axis, it can be seen that event $a$ *happened before* event $b$ at Alice's site. Similarly, event $c$ *happened before* event $d$ at Bob's site. It can also be seen that message generation event $b$ *happened before* message reception event $c$, and similarly $d$ before $f$. By the transitivity property, event $a$ *happened before* event $f$. However, one cannot conclude that event $a$ *happened before* event $e$ (or vice versa) since there is no sequence of messages between them. Event $a$ is therefore concurrent to event $e$.

Lamport captured this *happened before* relationship numerically by having each site maintain an incrementing software counter called a *logical clock*. The logical clock is used to apply scalar timestamps (now also called *Lamport timestamps*) to events based on the following rules [55]:

1. A site increments its logical clock before each event at that site.
2. The logical clock value is included as a timestamp when any message is sent.

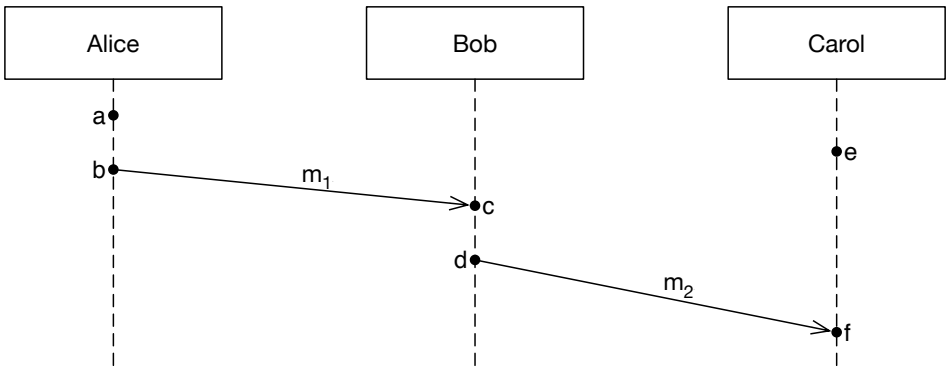



FIGURE 2.2: Ordering events in a distributed system consisting of three sites.

3. Upon receiving a message, the receiving site sets its logical clock value to the timestamp in the received message if the value is greater than its current logical clock value. The logical clock value at the receiving site is then incremented by 1.

Figure 2.3 presents a version of Figure 2.2 that includes the logical clock values obtained immediately after the corresponding event and assumes that all sites start with a logical clock value of 0. It can be inferred that $b$ has influenced $c$ and therefore *happened before* $c$, so the timestamp of event $b$ is less than the timestamp of event $c$. However, the converse is not true; an event with a lower timestamp value does not imply that it *happened before* an event with a higher timestamp value. For example, event $b$ has a larger timestamp than event $e$, but event $b$ is actually *concurrent* to event $e$ based on the *happened before* definition. More advanced techniques such as *state vectors* (i.e. *vector clocks*) [58][59] later addressed this problem by maintaining a vector (array) of timestamps at each site, where the length of the vector corresponds to the number of sites in the distributed system.

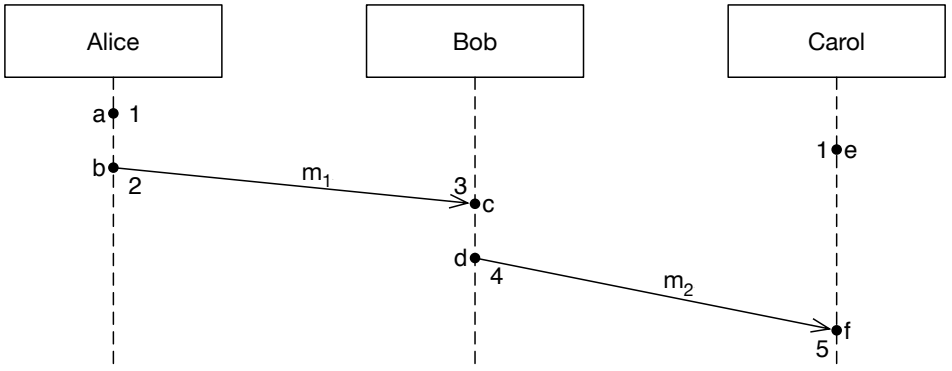



FIGURE 2.3: Lamport timestamps in the distributed system of Figure 2.2.

A variety of serialization protocols from distributed computing research have been identified as suitable for achieving totally ordered execution of operations in OT algorithms, thereby ensuring that causality is preserved since operations are always executed in their cause-effect orders. The methods used in OT literature include state vectors [3][60][5][46] and scalar timestamps [6][7][8][9], the latter of which is also essential to the Control Loop-based OT integration algorithm (CLOT) of this thesis. Lamport also proposed the use of unique numeric site identifiers to differentiate between two events that are concurrent, thereby obtaining a total order on the set of events in a distributed system. This technique was later adopted for resolving OT insert-tie scenarios [3][60][61], and it also appears in the algorithms of this thesis (for example, note the use of *userid* in Algorithm 5.7).

#### 2.2.1.2 "Convergence" Requirement

By first establishing the concept of *quiescence*, Ellis and Gibbs [3] defined the second requirement for correctness in groupware systems, the *Convergence Property*, as follows:

> A groupware session is *quiescent* iff all generated operations have been executed at all sites, that is, there are no requests in transit or waiting to be executed by a site process. The *Convergence Property* states that site objects are identical at all sites at quiescence.

User actions performed inside of a plaintext editor must be captured in such a way that they can be transmitted as operations to other users. For example, adding a character to a document may generate an *insert* operation containing the position of the insertion along with the letter to insert. Removing a character may generate a *delete* operation containing the position of the deletion. At its most basic, an operation can therefore be seen as a data structure with a *type* (*insert* or *delete*) and a *position* parameter (the location where to perform the insertion or deletion), while operations of type *insert* also require the actual character that is to be inserted. Zero is used to indicate the position before the first character, as shown in Figure 2.4(a) for the string "xyz". For example, an operation may describe the insertion of the letter $a$ at position 1. Rather than denote such a structure using the format $[insert, \text{“}a\text{”}, 1]$, operations in this thesis are defined using the simplified $insert[\text{“}a\text{”}, 1]$ pattern for legibility and historical reasons. The

result of applying the operation *insert*[“*a*”, 1] on the string “xyz” is shown in Figure 2.4(b). Similarly, *delete*[1] denotes an operation describing a deletion of the character at position 1 in the string “xyz”, namely the letter “y”. This can be equated with pressing the “delete” key on a keyboard when the cursor is at position 1, thereby obtaining “xz” as shown in Figure 2.4(c).

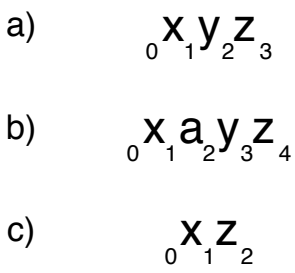



FIGURE 2.4: Plaintext position numbering (a) for the string “xyz”, (b) after applying the operation insert[“a”, 1] to “xyz”, and (c) after applying the operation delete[1] to “xyz”.

Given two concurrent operations $O_1$ and $O_2$, Ellis and Gibbs identified the need for a *transformation function* $T(O_1, O_2)$, where operation $O_1$ is transformed against operation $O_2$ to produce a transformed operation $O_1'$, which can be expressed as shown in Equation 2.1:

$$O_1' = T(O_1, O_2) \tag{2.1}$$

In this thesis, $O_1$ from Equation 2.1 is denoted as the *target* operation, while $O_2$ is denoted as the *reference* operation [61], where the target is said to be transformed *against* the reference. It follows that $O_2' = T(O_2, O_1)$ with $O_2$ as the target operation. From the client's perspective in an OT system, the target operation is typically the *remote* operation (sometimes referred to as the *incoming* or *received* operation), while the reference operation is a previously-applied *local* operation.

In order to ensure that the convergence requirement is satisfied when any two operations are executed in different orders, Ellis and Gibbs [3] identified a transformation property, later named *Transformation Property 1* (TP1) [60], that must be preserved in correct OT systems. The property ensures concurrent operations are commutative by requiring that executing $O_1'$ after $O_2$ on a document must always produce the same final document as executing $O_2'$ after $O_1$. It can be expressed as shown in Equation 2.2, where $\circ$ indicates the subsequent application of operations on the same document state $D$ and $\equiv$ indicates equivalence in the

resulting document states:

$$O'_1 \circ O_2 \circ D \equiv O'_2 \circ O_1 \circ D \tag{2.2}$$

For example, Figure 2.5 shows a scenario where users Alice and Bob both started with the equivalent document state of "xyz". Document states at each site are shown in bold, while underlined characters draw attention to insertions. Alice typed the letter "a" between the "x" and the "y" of "xyz" (position 1), thereby generating the operation $O_1 = insert[\text{“}a\text{”}, 1]$. The letter instantly appears in Alice's document (Alice sees "xayz") and $O_1$ is transmitted to Bob. Meanwhile, Bob performed $O_2 = insert[\text{“}b\text{”}, 2]$ upon his local document (Bob sees "xybz") and sent his operation $O_2$ to Alice. Once Bob receives Alice's operation $O_1$, it appears that Bob can apply Alice's $insert[\text{“}a\text{”}, 1]$ directly to get "xaybz". However, when Alice receives Bob's $O_2$, Alice cannot simply apply $insert[\text{“}b\text{”}, 2]$ to her local state, as this would result with "xabyz", and the documents will not meet the convergence requirement at quiescence when the system is idle.

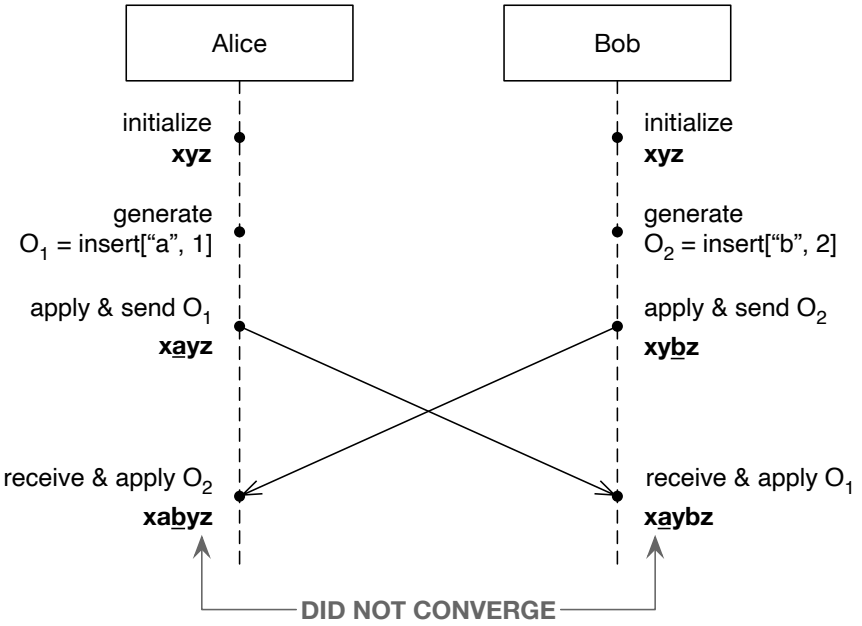



FIGURE 2.5: Naïve application of remote operations without transformations.

Using a condensed diagram structure typical of existing OT literature, Figure 2.6 shows how transformation functions can be employed to resolve the divergent states of Figure 2.5. Alice's local operation $O_1$ requires the position parameter of Bob's $O_2$ operation to be incremented by 1 to account for her previous local change. The transformed version of Bob's operation that Alice can apply is therefore denoted as $O'_2 = insert[\text{“}b\text{”}, 3]$, which was obtained by computing the

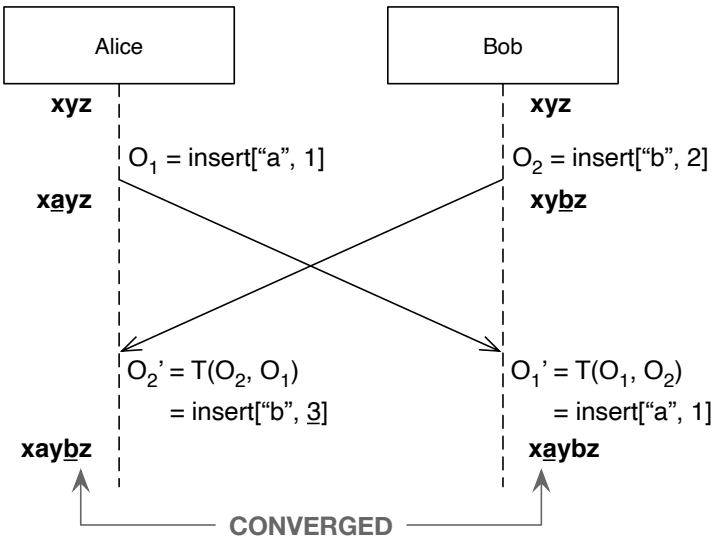



FIGURE 2.6: A simple “insert vs. insert” transformation example.

transformation $O_2' = T(O_2, O_1)$. By applying the resulting $O_2'$, Alice will now also obtain “xaybz”, which is consistent with Bob. Bob will actually have performed the transformation $O_1' = T(O_1, O_2)$ to obtain and apply $O_1' = insert[“a”, 1]$, but in this case the transformation function results with unmodified parameters for $O_1$. This equivalence between $O_1'$ and $O_1$ is why the naïve approach presented in Figure 2.5 appeared valid at first.

Since a new transformation function is required for each pair of operations, an OT system with $n$ operations requires a matrix of $n \times n$ transformation functions. For an OT system supporting the operations *insert* and *delete*, a 2x2 matrix containing the four corresponding transformation functions (TII, TDD, TID and TDI) is shown in Table 2.1.

TABLE 2.1: Matrix of transformation functions for an OT system with an insert operation and a delete operation.

| **target op** \ **reference op** | **insert** | **delete** |
|---|---|---|
| **insert** | TII | TID |
| **delete** | TDI | TDD |

Algorithm 2.1 to Algorithm 2.4 show the pseudocode for the four example transformation functions. The algorithms have been adapted from early single-character OT research by Ellis et al. [3] and Sun [61], where *c1* is used to denote the *target*

operation character and *p1* the *target* position, while *c2* and *p2* are used to denote the *reference* operation character and position, respectively. These four basic transformation algorithms will be revisited and revised in Chapter 5 to support operations acting on DOM-based text nodes (Algorithm 5.7 to Algorithm 5.10).

A transformation function for two insert operations (TII) is shown in Algorithm 2.1. In the example of Figure 2.6 above, it can be seen how user Alice reached line 5 of Algorithm 2.1 to return a transformed operation with an incremented position, while user Bob resulted with an unmodified operation by reaching line 3. The algorithm requires a unique numeric site or user identifier (*u1* and *u2*) to be included with each operation in order to resolve insert-tie scenarios, as previously mentioned in Section 2.2.1.1 and discussed further in Section 2.2.1.3.

**Algorithm 2.1** Transformation function for insert vs. insert.

```
1: function TII(insert[c1, p1], insert[c2, p2])
2:     if p1 < p2 or (p1 = p2 and u1 > u2) then
3:         return insert[c1, p1];
4:     else
5:         return insert[c1, p1 + 1];
6:     end if
7: end function
```

Algorithm 2.2 shows the transformation function for two delete operations (TDD). Line 5 shows how the position of the target operation must be adjusted if the deletion in the target operation occurred after the reference operation (the position of the incoming target operation must be decremented for the local deletion before it). To prevent an unnecessary deletion, the algorithm returns an identity operation when both the target and reference operations performed a deletion at the same position (line 7). Applying an identity operation to a document simply leaves the document unmodified.

**Algorithm 2.2** Transformation function for delete vs. delete.

```
1: function TDD(delete[p1], delete[p2])
2:     if p1 < p2 then
3:         return delete[p1];
4:     else if p1 > p2 then
5:         return delete[p1 − 1];
6:     else
7:         return identity;
8:     end if
9: end function
```

Algorithm 2.3 shows the insert vs. delete (TID) transformation algorithm, where line 5 contains the logic for decreasing the position parameter of the target operation in order to account for a deletion occurring before it in the reference operation.

**Algorithm 2.3** Transformation function for insert vs. delete.

```
1: function TID(insert[c1, p1], delete[p2])
2:     if p1 ≤ p2 then
3:         return insert[c1, p1];
4:     else
5:         return insert[c1, p1 − 1];
6:     end if
7: end function
```

Finally, Algorithm 2.4 shows the delete vs. insert (TDI) transformation logic. The position of the target operation must be incremented to include the character inserted by the reference operation, as shown in line 5.

**Algorithm 2.4** Transformation function for delete vs. insert.

```
1: function TDI(delete[p1], insert[c2, p2])
2:     if p1 < p2 then
3:         return delete[p1];
4:     else
5:         return delete[p1 + 1];
6:     end if
7: end function
```

The complexity of the system grows significantly if more than two users are to be supported. By using a three-dimensional state space graph of consistent global states, the work of Ressel et al. [60] identified that an additional property, *Transformation Property 2* (TP2), is required for convergence when more than two sites are involved and operations can be executed in arbitrary orders. The property is shown in Equation 2.3 for operations $O_1$, $O_2$ and $O_3$ defined on the same state, and note that for any $O_1$, $O_2$ and $O_3$, $T(O_1, O_2 \circ O_3) = T(T(O_1, O_2), O_3)$.

$$T(O_3, O_1 \circ T(O_2, O_1)) = T(O_3, O_2 \circ T(O_1, O_2)) \tag{2.3}$$

Figure 2.7 illustrates how TP2 requires an additional layer of operation combinations to be supported by transformation function logic, since Alice must transform the received $O_3$ operation to account for her previous sequence of applied operations in such a way as to have her $O'_3$ match the $O'_3$ obtained by Bob when transforming $O_3$ to account for his alternate sequence of operations. Even beyond three users, TP1 and TP2 remain as requirements to ensure convergence,

but finding transformation functions that provably satisfy TP2 was determined to be non-trivial [62][63].

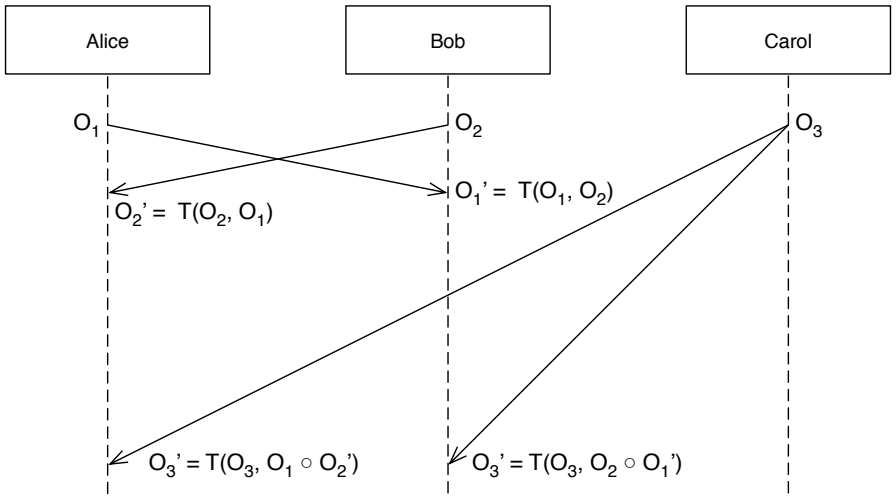



FIGURE 2.7: Transformation Property 2 requirement for three concurrent operations.

In OT system design, it is useful to separate low-level *transformation functions* from high-level *integration algorithms* (sometimes called *transformation control algorithms* [4]). Integration algorithms are responsible for determining how operations move through the OT system before and after the transformation functions. This includes the management of causality requirements of operations (see Section 2.2.1.1), as well as algorithms for communication between sites based on the desired network topology. The correctness of the OT system as a whole therefore depends on the *integration algorithms* working hand-in-hand with the *transformation functions*.

OT systems may be susceptible to *OT puzzles* that cause the system to behave incorrectly in very specific situations. As an example, a scenario now known as the *dOPT puzzle* was discovered by Ellis and Gibbs [3], and solving this puzzle would motivate the creation of many other OT algorithms [64]. A variation is shown in Figure 2.8. In this example, Bob generated operations $O_2$ and $O_3$ concurrently to Alice's operation $O_1$, and Alice's transformation of Bob's $O_3$ against her $O_1$ results with a different final document than Bob's, which Bob obtained by transforming Alice's $O_1$ against his $O_2$ followed by his $O_3$.

The solution to this puzzle is achieved as part of the integration algorithm since the responsibility of an integration algorithm includes determining when a remote operation must be transformed against a local operation. The *Context-based OT*

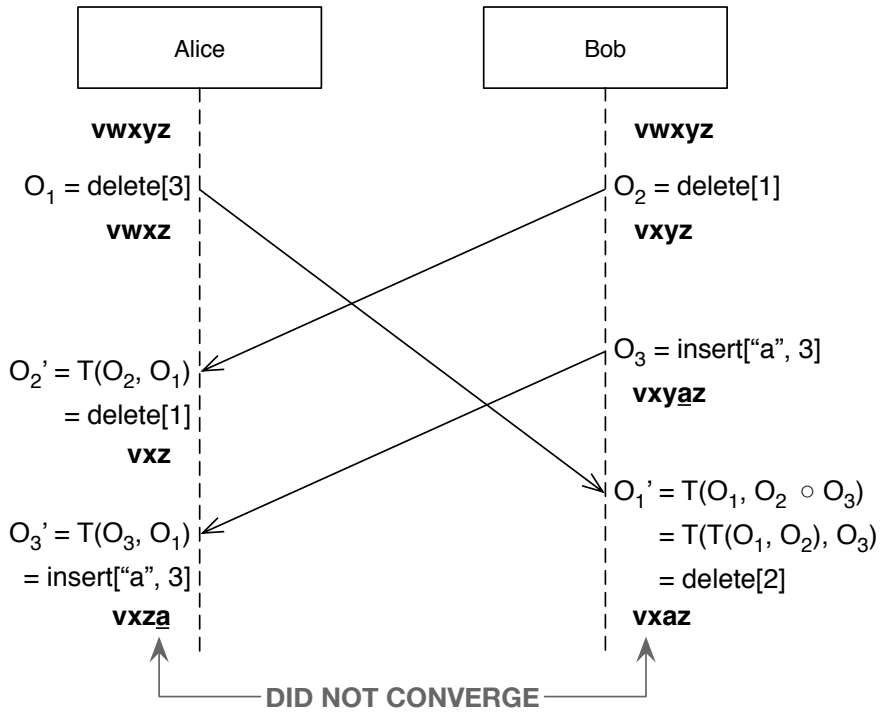



FIGURE 2.8: Example scenario showing the dOPT puzzle.

(*COT*) integration algorithm of Sun et al. [46] proposed the concept of an *operation context* to describe the document state on which an operation is defined. The algorithm ensures that operations can only be applied to a document if the context on which they are *defined* is equal to the context on which they are to be *executed*, and that two operations can only be transformed if they were defined on the same document state [61]. Two operations are then *concurrent* if neither operation appears in the context of the other.

The COT algorithm shows how a variation on state vectors (see Section 2.2.1.1) can be used to track the evolving contexts of operations between multiple sites [46]. By having each site keep a buffer of all previously-applied operations, the contexts of received (remote) operations can be compared to those of the operations in the local history. No transformation is required if the contexts match, but if the context of a received operation is not in the site's local history, the received operation is not yet ready for integration and the site must wait to receive the missing operation on which the received operation is causally dependent. If the local history contains operations that are not included in the context of the received operation, then these local operations are concurrent with the received operation and the received operation must be transformed relative to them.

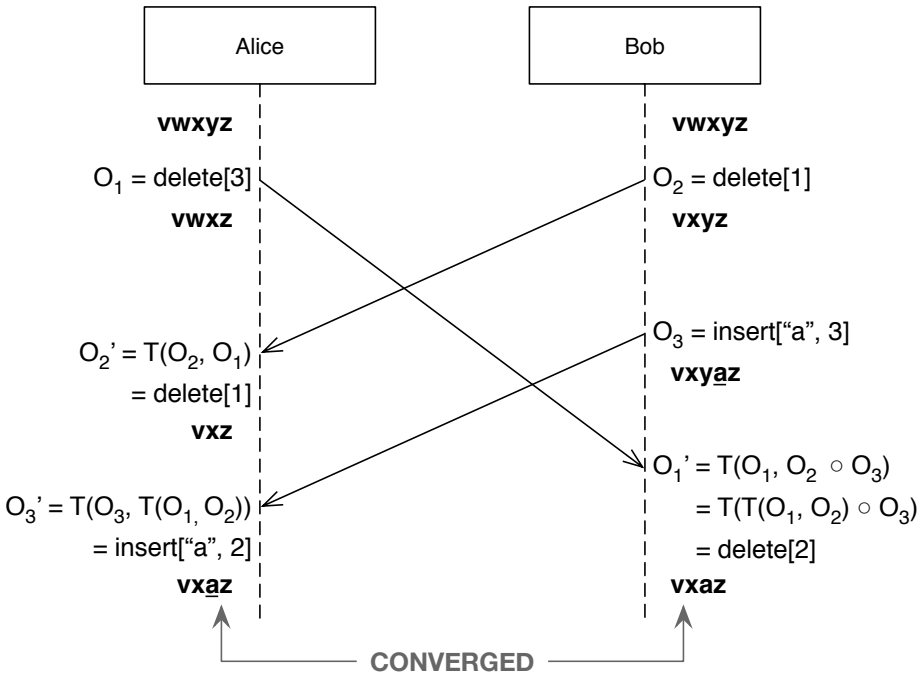



FIGURE 2.9: COT algorithm solution to the dOPT puzzle.

In Figure 2.8, operations $O_1$ and $O_2$ both have an original context of "vwxyz", the initial document state. After Bob applies $O_2$, his document state becomes "vxyz", from which the operation $O_3$ is then obtained, along with the state "vxyaz". When Bob receives $O_1$ from Alice, the context at Bob's site does not match the context of Alice's $O_1$. However, by using the COT algorithm, Bob can upgrade the context of $O_1$ with the transformation $T(O_1, O_2)$. In this way, the obtained intermediate transformation result has a context of $O_2$, which matches the context of $O_3$, and can therefore be transformed against $O_3$ to obtain the final operation $O_1'$ to apply to his document.

The COT algorithm can be used to correct Alice's calculation of $O_3'$, as shown in Figure 2.9 (adapted from [65]). Alice must upgrade the context of Bob's $O_3$ operation by transforming $O_3$ relative to the buffered operations available in the context of Alice's $O_2'$ (namely $O_1$ and $O_2$). Like for Bob, the context of $O_3$ at Alice's site is $O_2$, so Alice first performs $T(O_1, O_2)$ to match this context before transforming $O_3$ against the resulting operation. In this way, Alice obtains an operation that ensures document convergence with Bob and thereby solves the *dOPT puzzle*.

With careful design of the integration algorithm, it is possible to guarantee convergence between more than two users without requiring TP2 to be ensured by

transformation functions and while also avoiding the *dOPT puzzle*, as will be shown with the Jupiter algorithm of Section 2.2.1.5, the SOCT4 algorithm of Section 2.2.1.6, and the CLOT algorithm proposed in Chapter 4.

#### 2.2.1.3 "Operation Intention Preservation" Requirement

The use of serialization techniques to achieve the convergence of data is typically sufficient for concurrency algorithms used in applications such as databases, where policies such as *last-writer-wins* can be applied to retain only the value with the largest timestamp [66]. However, OT systems cannot simply discard or overwrite updates, but rather must combine the effects of concurrent updates such that replicas converge on a final shared state that preserves the meaning of each original operation. That is, the *intention of an operation O* can be defined as the execution effect obtained by applying $O$ on the document state from which $O$ was generated [4].

It is therefore essential that operations, their attributes, and the transformation functions that act upon them, are defined in such a way as to best capture the true intended purpose of a specific operation within a specific target application such as a plaintext editor. If the intention of an *insert* operation is defined as "the character *char* must end up at position *pos*", the scenario of Figure 2.10 would result from the use of a corresponding transformation function that meets this definition. Besides the fact that the content fails to converge, the intention of each

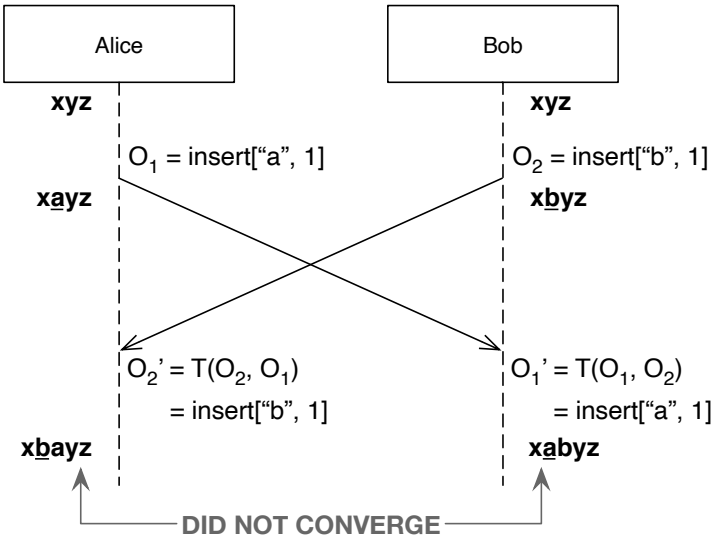



FIGURE 2.10: An "insert vs. insert" transformation without insert-tie handling due to incorrect definition of operation intention.

user's local operation is not preserved after the transformed remote operation is applied (each wanted their respective letter at position 1 in the final result but neither ended up with this). The definition of the intention of *insert* operations must therefore be relaxed to allow for such insert-tie scenarios. Sun proposed the following improved definition which can be used to verify the correctness of a corresponding transformation function with respect to intention preservation of *insert* operations [61]:

> A transformed *insert* operation preserves the intention of the original *insert* operation if:
>
> 1. the transformed *insert* operation inserts and only inserts the character inserted by the original operation; and
> 2. the insert position relative to other existing characters in the document is preserved.

Maintaining the relationship between a character relative to its neighbouring characters (i.e. an inserted character must appear to the left or right of other characters from the original document) is especially relevant to the functionality of CmRDTs, as will be explored in Section 2.2.2. The transformation functions of Section 2.2.1.2 (Algorithm 2.1 to Algorithm 2.4) showed how, as long as both involved sites implement the same logic, unique user identifiers can be used to preserve the original effects of the operations involved while still allowing documents to converge. In the case of two operations with a letter inserted at the same position, the target (local) operation of a user with a lower *uid* takes priority over the reference (remote) operation since the target operation's position parameter remains unchanged when using the given algorithms.

#### 2.2.1.4 "User Intention Preservation" Requirement

Even if all *intentions of operations* have been preserved correctly and documents converged correctly, users may not be satisfied with the resulting document state. This is known as a violation of the *intentions of the users*. As will be shown with examples in this section, the *intentions of the users* may be conflicting in such a way that they cannot be resolved automatically without human observation and correction. A "best effort" approach must therefore be applied when resolving

conflicts between operations so as to best preserve the *intentions of the operations*, as well as the *intentions of the users*, together referred to as *intention preservation*.

For example, concurrent changes may result with semantic inconsistency problems that violate the *intentions of the users* [4]. If an analysis document contains the misspelled word "analye", Alice and Bob may both attempt to correct it by inserting the letter "s", resulting with the word "analysse". This result was clearly not the intended outcome of either user but is relatively easy to spot and correct manually. As another example, Alice may correct the word by inserting the letter "s", while Bob may insert the letter "z", resulting with the word "analysze". In this case, the users may need to use external group communication methods and social protocols to decide if the document is to adhere to English or American spellings before one of the two users applies the correction.

Different users will make different assumptions about how a text editing tool should behave in terms of its user experience. For example, if a document in a single-user rich-text editor contains the text "a**b**c" and a user clicks to position their mouse cursor after the central bold letter "b", some users may expect a letter typed after the "b" to be bold, while others may expect the new letter to match the regular style of the letter "c". Similarly, in collaborative editing, it is possible for documents to reach a state that users perceive as undesirable or unexpected based on the implementation decisions of the underlying conflict resolution algorithms (rather than errors in those algorithms). For example, if an operation from Alice deletes a paragraph while a concurrent operation from Bob adds a sentence to it, the transformation function must be implemented to select one of the following three alternatives to automatically resolve the conflict:

1. *Update-Wins - Leave the new paragraph that includes Bob's sentence:* In this case, Alice may be unsatisfied that her deletion was ignored. Perhaps she felt the entire paragraph wasn't of value, irrespective of any new change from Bob.

2. *Delete-Wins - Delete both the paragraph and the new sentence:* In this case, Bob may be unsatisfied for losing the work he did on the new sentence. Perhaps he felt the new sentence changed the paragraph in such a way that its deletion is no longer appropriate.

3. *Neither-Wins - Delete the paragraph, but leave just Bob's new sentence:* In this case, both users are likely unsatisfied, as the sentence may make little sense on its own.

The *Delete-Wins* alternative was selected for the implementation of algorithms in this thesis. This selection was made since the deletion operation is considered as acting upon the higher-priority parent structure, thereby rendering child-level changes moot in such conflict scenarios. However, with a clear model of the actions that are possible within a given collaborative co-editing application, operations in an OT system can be designed in such a way as to minimize the scenarios where one of the above three policies is even required. Operations and transformation functions can be created to capture the semantic meaning of a user action with the necessary level of detail so as to avoid data-losing conflict resolution outcomes.

While the *dOPT* algorithm of Ellis and Gibbs [3] used a single linear address space for the document data model along with character-wise *insert* and *delete* operations, rich-text editing functionality requires defining a tree-structured data model and the corresponding operations with support for hierarchical addressing. Early work attempting to define OT operations for tree-based content relied on just three primitive operations (*insert*, *delete* and *update*) [67]. While rich text co-editing behavior can indeed be modeled using these operations, they are insufficient for preserving the intentions of all users during many co-editing scenarios. For example, consider a scenario where Alice drags and drops a paragraph to a different location while Bob adds a sentence to that same paragraph. With the set of operations limited to *insert*, *delete* and *update*, the drag-and-drop action will be mapped to a *delete* operation followed by an *insert* operation containing the original text. Depending on the design of the operations and implementations of the transformation functions involved, Bob's insertion may be lost or appear outside of the moved paragraph instead of reaching the new location of the paragraph as the user intended. In order to correctly preserve the intentions of both users in this scenario, Alice's *delete* and *insert* operations can be combined into a *moveParagraph* operation to be used in such situations. Transformation functions of Alice's *moveParagraph* against Bob's *insert* operation would then contain the conflict resolution logic that can properly associate the destination of Bob's *insert* operation with the new location of the paragraph within the *moveParagraph* operation.

This thesis will focus on reaching a hierarchical DOM-based data model [33]. A set of operations must therefore be defined to operate on such a data model while remaining compatible with transformation functions and their requirements for convergence and intention preservation. As Section 5.1.1 will discuss, many opportunities for new operations have been discovered in order to preserve the intentions of users as best as possible throughout all DOM-based co-editing tasks.

#### 2.2.1.5 Jupiter Algorithm

Section 2.2.1.2 identified how the properties of TP1 and TP2 are important conditions that enable transformation functions in an OT system to achieve convergence. However, they are not *sufficient* conditions for a correct OT system, since there are also additional requirements such as how applying $O'$ on the local replica must have the same effect as applying $O$ did on its original remote replica (intention preservation). They are also not *necessary* conditions, as the integration algorithm can be used to ensure that a pair of operations is never transformed in different orders or in different contexts, thereby making it possible to omit the TP2 requirement entirely [61]. This will now be shown with the Jupiter algorithm of Nichols et al. [6].

The Jupiter algorithm avoids the need for TP2 by using a centralized server to establish a global sequence of operations. In this way, the algorithm ensures that one operation is only ever concurrent with one other operation, even in the presence of more than two users. After immediately applying operations locally, clients transmit their operations to a centralized server, which serializes all operations and propagates them to the remaining clients using a broadcasting mechanism. The global order imposed on operations ensures that the causality and convergence (TP1) requirements are met, while transformation functions such as Algorithm 2.1 to Algorithm 2.4 of Section 2.2.1.2 remain valid for achieving intention preservation. In fact, the transformation functions used by clients are the same as those used by the server. Each client synchronizes its local replica only with the server, and each client-server link therefore participates in a separate instance of the Jupiter two-party synchronization protocol.

The Jupiter integration algorithm uses a two-dimensional *state space graph* at each site in order to track all possible operational transformation paths and guide the selection of operations for transformation [5]. The two-way synchronization

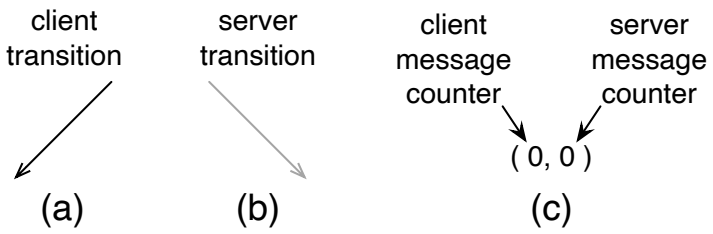



FIGURE 2.11: Visual representations of (a) client transition arrow and direction, (b) server transition arrow and direction, and (c) message counter elements in state vectors.

method can be extended to $n$-way synchronization by having the server keep a separate state space graph for each client and passing messages between internal buffers associated with each state space graph [68]. The serialization order of received operations can be established by using a policy such as arrival time at the server. Based on the state vector value at each node of the graph, each site determines the necessary transformations as new operations arrive, and the server broadcasts possibly-transformed operations to ensure all clients remain synchronized. The state space graph enforces that any two operations to be transformed must have originated from the same state of the document. Document models have converged when client and server transitions reach the same state of the state space graph. The approach builds on other algorithms that have used lattice structures to represent time in distributed systems [57][58].

To better explain the Jupiter algorithm and its use of the two-dimensional state space graph, an example is now given using the visual elements introduced in Figure 2.11. Figure 2.11(a) shows how client transition arrows will be denoted as black and pointing towards the left, while Figure 2.11(b) shows the server transition arrow as grey and pointing to the right. Figure 2.11(c) illustrates how the first element in any state vector represents the client's message counter, while the second element always represents the server's message counter. The message counters are scalar-based timestamps representing the number of client or server messages processed to get to the given state.

Figure 2.12(a) shows the initial (empty) state space graph of user Alice, where both Alice and the server (not shown) start with the same document state of $(0, 0)$ and move downward in time through their state space graphs as new operations are introduced. Figure 2.12(b) uses the client transition arrow of Figure 2.11(a) to show Alice's state space graph after she performed a local operation $a$ and transmitted it to the server. This change brings Alice's document from state $(0, 0)$

into state $(1, 0)$. Once the server has processed Alice's message, it will have the same graph for Alice, but will also maintain a separate graph for a second client, Bob [68]. After immediately applying a change to their local document, clients may experience high levels of latency in their connection to the server and cannot assume that they will receive immediate replies from the server, therefore opening the possibility for conflicts to occur. In order to observe a conflict scenario, assume that Bob submitted a concurrent operation, which the server processed *before* Alice's operation.

Before receiving an acknowledgement for her operation, Alice will receive a new operation from the server, namely the server's broadcast of Bob's operation. Figure 2.12(c) shows Alice's state space graph after accepting this operation, denoted as $b$ on the server transition arrow. The server had reached a different state due to Bob's operation, namely $(0, 1)$, and Alice has now become aware of this divergence.

Although both Alice and the server (and Bob) had the same initial document state, the document has now transitioned into two different states. In order for both Alice and the server to once again reach a consistent state, the Jupiter algorithm dictates that they must both (individually) execute the *xform* function shown in Equation 2.4. The function is similar to the $T$ function of the *dOPT* algorithm (Equation 2.1 of Section 2.2.1.2), but while the $T$ function obtains one transformed value for the operation passed as the first input parameter (the *target*), one call to the *xform* function returns the transformation results of both $T(a, b)$ and $T(b, a)$ as the output.

$$xform(a, b) = \{a', b'\} \tag{2.4}$$

Figure 2.12(d) shows the resulting diamond structure that visually represents the transformed operation results $a'$ and $b'$ (shown using dashed transition lines) obtained by Alice. Alice can apply $b'$ directly to her document to reach state $(1, 1)$. Alice calculates $a'$ in case documents diverge by more than one operation, as will be explained below. Since the server will do the same transformations upon receiving Alice's conflicting operation $a$, the server will compute and apply $a'$ to its state of $(0, 1)$ to reach the state of $(1, 1)$. The documents of Alice and the server will therefore have converged, and the server will broadcast the transformed $a'$ operation to Bob, which Bob can apply directly to reach the convergent state of $(1, 1)$ as well.

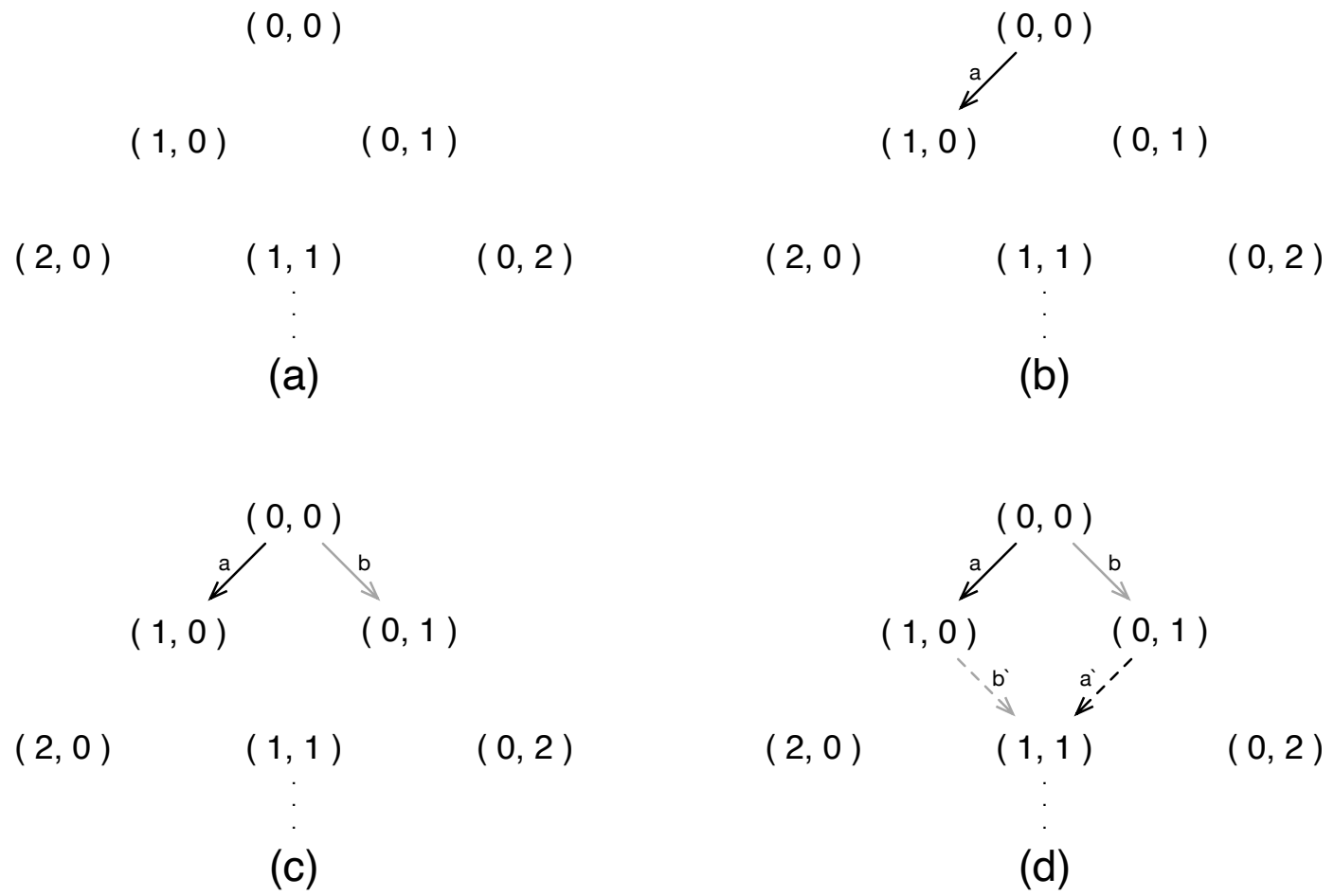



FIGURE 2.12: Evolving stages of Alice's state space graph in a sample Jupiter algorithm execution, where (a) shows the converged clean graph, (b) shows the diverged graph after Alice's local operation $a$, (c) shows the graph with a conflict after receiving operation $b$ from the server, and (d) shows the graph after applying the *xform* function.

Bob's path through the state space is shown in Figure 2.13, where $b$ represents Bob's original operation, and $s$ represents Alice's transformed operation (previously denoted as $a'$, but denoted here as server operation $s$ since Bob did not encounter any conflicts and therefore was not required to execute *xform*). Although different paths were taken through the state space, all sites have now converged by reaching the same state of $(1, 1)$.

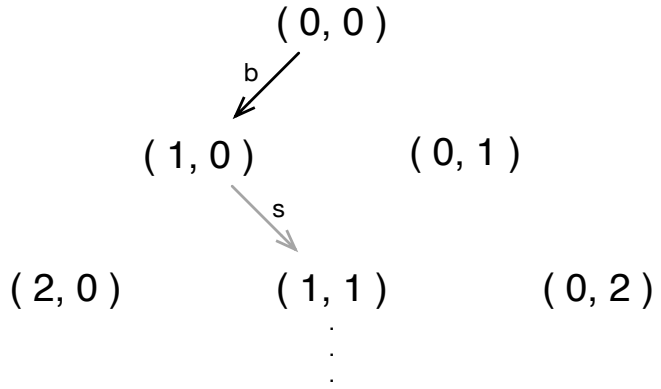



FIGURE 2.13: Bob's state space graph in a sample Jupiter algorithm execution.

The use of the state space graph by the Jupiter algorithm allows clients and the server to diverge by more than just one operation, while still guaranteeing the same document content once the same state in the state space graph is reached. Divergence by more than one operation, however, requires computing and storing intermediate transformation results. For example, in a variation of the previous scenario occurring after Figure 2.12(b), the server may have accepted a different concurrent operation $c$ from a third user Carol before processing Alice's operation $a$. Figure 2.14(a) shows the extended state space graph that includes the new operation $c$ that Alice received from the server in this scenario, thereby indicating that the server diverged by two operations. In this case, Alice can use *xform* as before to compute operations $a'$ and $b'$, but cannot simply transform operation $c$ against her local operation $a$. As per the concept of *operation context* defined in Section 2.2.1.2, this is because $a$ and $c$ do not have the same generation state, since $a$ was generated from $(0, 0)$ while $c$ was generated from $(0, 1)$. However, the $a'$ value Alice obtained after executing the *xform* function *does* have the same generation state as operation $c$. Alice can therefore compute $xform(a', c) = \{a'', c'\}$, which includes the transformation of operation $c$ against $a'$ to get a proper operation $c'$ for reaching state $(1, 2)$, as shown in Figure 2.14(b). Alice also obtains $a''$ from this execution of *xform*, which is saved for future transformations in case the server once again sends a new message instead of having integrated her operation $a$. By

using this technique to ensure that operations are always transformed in the correct context, the Jupiter algorithm avoids the *dOPT puzzle* defined in Section 2.2.1.2. While the use of the Transmission Control Protocol (TCP) was not assumed for the original *dOPT* algorithm [3], Jupiter's use of TCP for connections between clients and the server also helps to simplify the algorithm since TCP not only guarantees the delivery of data, but also guarantees that packets will be delivered in the same order in which they were sent [69].

An evolved version of the Jupiter approach is at the heart of the Wave collaboration tool (see Section 2.3.2) and has been powering Google Docs since 2010 [28], thereby showing the algorithm's strong real-world applicability. OT algorithms NICE [8] and TIBOT [9] also use server-side transformations combined with scalar-based timestamping techniques, but the approach used in the CLOT algorithm proposed in this thesis assumes a "serverless" architecture *without* server-side transformations.

#### 2.2.1.6 SOCT4 Algorithm

The SOCT4 OT integration algorithm [7] avoids the server-side transformation requirement of the Jupiter algorithm while also avoiding the TP2 condition defined in Section 2.2.1.2. This is accomplished by having clients request a global timestamp from a central server before submitting each of their operations. As is expected of OT algorithms, operations are immediately applied locally first to ensure that the client user interface continues to feel responsive to user actions, but in the SOCT4 algorithm, operations are not immediately sent to the server after this application (this is known as a *deferred broadcast*). Instead, each client calls a server-side *Ticket* function and the server provides continuously growing positive integer values in response. Based on the value of the received timestamp value, the client will know if it is missing any operations, which it will then wait for and apply as they are received. Any local outgoing operation is then updated (*forward transposed*) to account for the newly received operations. The timestamp received via the *Ticket* function is used in the message to the server, which the server will always accept and rebroadcast.

The SOCT4 algorithm therefore builds on previous distributed systems research in the area of *sequencers* [70] for globally serializing operations. In this way, the OT

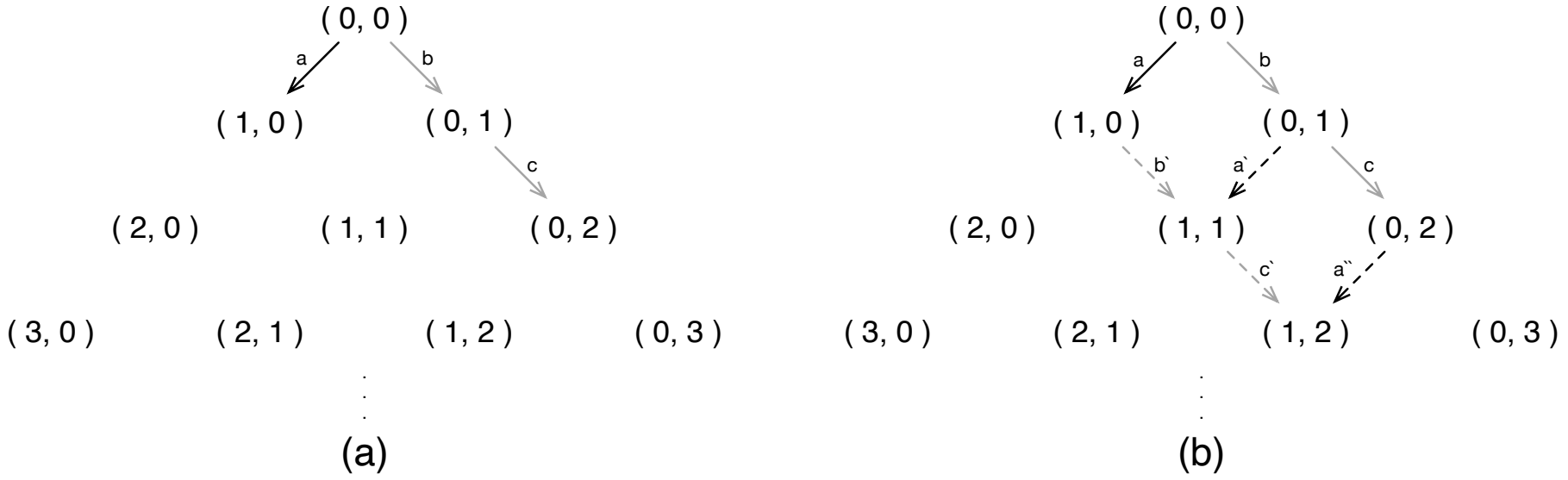



FIGURE 2.14: State space graph of Jupiter algorithm execution for (a) divergence by more than one operation, and (b) use of the *xform* function to reach a consistent state.

algorithm ensures that the causality preservation requirement is met since a continuous global order of operations is maintained in a similar fashion to the causal precedence techniques presented in Section 2.2.1.1. Convergence is also achieved by using the timestamps associated with each operation since the operations can be applied based on the order of the timestamps. Finally, intention preservation is achieved with the transformation of concurrent operations using transformation functions such as Algorithm 2.1 to Algorithm 2.4 of Section 2.2.1.2.

Rather than being assigned a fixed logical timestamp from the central server, the approach proposed in this thesis submits operations with a client-desired timestamp based on a Finite State Machine at each client site. After applying an operation locally, each client sends and stores the operation until it has been acknowledged by the server. Upon receiving a new operation from the server that is not an acknowledgement, each client transforms the incoming operation so that it can be applied locally, and then *re-sends* an updated version of the local operation. This process is repeated until the client's operation has been acknowledged by the server. Having the timestamp specified by the client enables the server to consist of only very basic logic for accepting new timestamps. Total ordering is therefore achieved in a way that allows "serverless" databases to be used for scalable cloud deployments of OT-based web applications.

### 2.2.2 Commutative Replicated Data Types

In 2006, Oster et al. [71] proposed an alternative optimistic consistency control methodology that has evolved into a class of algorithms now known as *Commutative Replicated Data Types* (CmRDTs), an operation-based subset of what are now called *Conflict-free Replicated Data Types* (CRDTs) [72].

CRDTs use a document data structure defined in such a way as to allow operations to be commutative. In this way, the order in which the operations are applied does not matter and the need for any transformation of the operations is avoided. Instead of the OT approach of using absolute positions such as the value 1 in operations such as $insert[\text{“}a\text{”}, 1]$, CmRDTs have operations based on positions relative to other characters. The ideas were formalized by Shapiro et al. in the paper "Designing a Commutative Replicated Data Type" [73].

CRDTs have applications beyond text editing, with products such as the open source key-value distributed NoSQL database known as "Riak" providing five "CRDT-inspired data types": Flags, Registers, Counters, Sets, and Maps [74]. However, the use of CRDTs in real world co-editing applications remains rare with several significant problems remaining to be solved which do not exist for OT [75], including "the correctness of key CRDT data structures and functional components, tombstone overhead, variable and lengthy identifiers, inconsistent position-integer-ordering and infinite loop flaws, position-order-violation puzzles, and concurrent-insert-interleaving puzzles" [76]. Support for a DOM-based data model and the development of more complex operations such as splitting, merging and moving of nodes also remains a research challenge, as not all operations can be made commutative [77]. The remainder of this thesis will therefore focus on the OT approach to optimistic consistency control.

## 2.3 Collaboration Platforms

This section explores existing platforms, libraries, and frameworks for web-based collaboration. Since usable real-world systems based on CmRDTs are still rare [76], all approaches included in this section make use of OT techniques, except the *NCCT Lab Projects*, which are included to provide further background on the early history and motivation that have lead to the work of this thesis. Section 8.4 will revisit the functionality offered by approaches most relevant to the objectives of this thesis in a summary table (Table 8.1).

### 2.3.1 NCCT Lab Projects

In 2007, a real-time platform that supports collaboration across multiple domains was developed at the University of Ottawa's NCCT Lab and named "UC-IC" [78][79]. UC-IC was a cloud-based real-time collaboration platform which allowed users to send, share and distribute control over web applications in real-time. The Session Initiation Protocol (SIP) standard was used and extended to bridge users on different domains such that they could discover other users and share applications with each other. Users could concurrently collaborate on multiple applications from several domains, where each application could be part of multiple collaborative sessions within a browser-based desktop-like environment. This

technique was similar to the windowing toolkit and "code shipping" approach identified by Nichols et al. [6] but was implemented entirely using web-based technologies. While UC-IC contained a TinyMCE-based word processing application, conflict resolution techniques such as OT were not used at the time. UC-IC is described further in previous works of NCCT team members [80][81].

In 2009, a web-based platform with a focus on collaborative multimedia content sharing called "Watch Together" was developed that also aimed to achieve real-time collaboration between users who, as a group, share a collaborative session [82][81]. This implied that all users in the session saw the same state of the system. For example, all users within the same session saw the same video (at generally the same playback time), the same image, or the same page in a document or slideshow. Actions performed by a user (such as fast forwarding in the video, changing the image, or changing the page) were replicated across all users in order to ensure that the same state was maintained. Instead of providing a collaborative online desktop like UC-IC, Watch Together was created as a lighter version that allowed for quicker development of synchronized applications. The client side was developed using Adobe Flash, and provided an API which applications used to synchronize data among the users of an established session. With the rise of HTML5, Flash has now been phased out by most modern web browsers, but at the time, Flash allowed Watch Together to provide video and audio chat via live webcam streams so that users could see and hear each other as they collaborated over the multimedia content. The server acted as a simple relay for synchronization messages and would not process or examine the message contents. A user would request that a message be broadcast to other users in their session. New users that joined the session would request the state of the session from existing users. The platform was evolved to scale within a cluster, as well as geographically across multiple clouds [83][84].

### 2.3.2 Wave & Google Docs

One of the first web-based implementations of OT was included as part of *Wave*, which was first introduced in a developer preview at the Google I/O 2009 conference as *Google Wave* [85]. After the product was discontinued by Google in 2010, the source code was transferred to the Apache Software Foundation [86] to form *Apache Wave*. The *Apache Wave* project was then retired in 2018, although a

few members of the open source community continue to maintain the project on GitHub [87].

Wave is a real-time collaboration framework that allows multiple *participants* to co-edit *waves* consisting of one or more threaded conversations known as *wavelets* [88]. XML documents within *wavelets* are used to represent messages and are manipulated by generating operations via the Wave Java API. An example document containing the text "abc" is shown in Listing 2.1.

```
1 <body>
2   <line/>abc
3 </body>
```

Listing 2.1: Example XML document within a Google Wave wavelet.

A set of *annotations* is stored alongside each XML document as meta-data and is interpreted by a client when rendering a document in the browser. Each annotation contains a key-value pair that describes a type of effect to be applied to an arbitrary range of characters within the document. Annotation ranges may overlap and, among other functionality, serve to define text styles within the document (bold, italic etc.) [89]. By modeling the document as a linear sequence of characters and keeping such annotations alongside the well-formed XML, Wave is able to achieve rich-text editing with primitive operations such as *insertCharacters* and *deleteCharacters* [90]. Adapted from [32], Figure 2.15 summarizes this architecture.

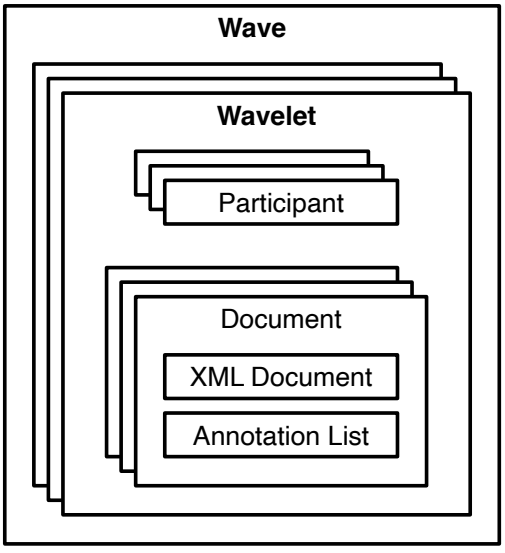



Figure 2.15: Wave block structure.

The approach of *Jupiter* [6] presented in Section 2.2.1.5 was found to be an appropriate foundation for the implementation of Wave, where browser-based Wave clients communicate with other Wave clients through a centralized server [91]. However, in contrast to Jupiter, Wave requires clients to maintain a buffer of local operations and wait for a sent operation to be acknowledged by the server before the client sends additional operations. An operation composition technique was added to reduce the number of operations that are transformed or transmitted. While the Jupiter approach required the server to maintain a separate state space for each client and convert client operations between them, the Wave extension to Jupiter requires only a single copy of the document (and its associated history) to be maintained on the server, thereby reducing the server-side memory consumption.

Wave remains one of the most sophisticated browser-based collaboration systems to date, once having over 1.1 million lines of Google Web Toolkit (GWT) code that was compiled into client-side JavaScript code (compatible with browsers going as far back as Internet Explorer 7) and server-side Java objects [92]. However, the annotation technique used by Wave only works with a predefined list of editing functionality (where annotations are defined for each type of style, for example) and cannot support general HTML tags and features of the DOM such as multiple text nodes per element node. A different approach is therefore required to synchronize any arbitrary DOM between multiple users. In fact, the official Wave FAQ from Google [93] stated the following:

> "While HTML attempts to separate semantic content from presentation, waves also need to be concerned with the operational transformation layer underneath, and HTML makes OT (Operational Transformation) difficult if not impossible."

Based on their experience with Wave, Google updated Google Docs in 2010 to use a similar Jupiter-based approach [28]. The annotation-based data model requires Google Docs to manage a proprietary DOM that is very verbose and difficult to work with outside of Google Docs, often requiring a laborious cleanup process when copied and pasted into other locations [94]. For example, the rich-text content shown in Figure 2.16 applies a few basic styles to three words, where the first two words are separated by a line break and the last is separated by a new paragraph. Adapted from [95], Table 2.2 shows the HTML DOM produced by the approach

of Google Docs when compared with Firepad (Section 2.3.6 below) and the DOM of a typical (non-collaborative) WYSIWYG editor from which Figure 2.16 was obtained.

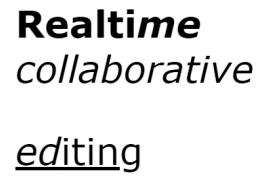



FIGURE 2.16: Sample rich-text document content.

While all three editors generally render an output resembling Figure 2.16, Google Docs and Firepad cannot produce a difference between the line break and the paragraph break, instead displaying the same amount of space between the three words. This is because both Google Docs and Firepad operate on a predefined intermediate data format and generate an HTML DOM output from this internal format. While the line break limitations and messy DOM output are just some examples of the drawbacks introduced by this mapping phase, modifying the underlying DOM in this fashion allows both editors to enable their respective OT-based real-time collaboration features. The non-invasive *transparent adaptation* [29] approach taken in this thesis aims to allow existing WYSIWYG editors and other web applications to become collaborative while they continue to maintain their clean DOM. A generalized DOM synchronization solution therefore implies locating and modifying nodes without the aid of a simplified intermediate data format or custom markup, as the operations and algorithms of Chapter 5 will explore.

### 2.3.3 ShareDB

*ShareDB* (formerly *ShareJS*) is an open source library that supports OT-based real-time synchronization of data [96]. Its modular nature allows it to be extended with a variety of "OT Types". Text-based and JavaScript Object Notation (JSON) OT Types have already been developed for ShareDB. In addition, it can persist data in a variety of databases such as the NoSQL document database MongoDB. ShareDB was created by one of the developers on the Google Wave project, Joseph Gentle, who, upon learning of the demise of Google Wave, wanted to implement a simpler open source version of some of the concurrency control techniques behind Google Wave [92]. ShareDB is therefore also based on ideas from Jupiter [6]. The

TABLE 2.2: Comparing HTML DOM of three editors displaying the same rich-text content.

| Editor | HTML DOM |
|---|---|
| **Google Docs** | Realtime collaborative < div class="kix-lineview" style="height: 20px; direction: ltr; text-align: left;">  editing. |
| **Firepad** | <pre><br>Realti<br>me<br></pre><br><pre><br>collaborative<br></pre><br><pre><br>ed<br>iting<br></pre> |
| **Typical WYSIWYG Editor (Thesis Focus)** | <br>Realtime<br><br>collaborative<br> <br> <br><u>editing</u><br> |

Jupiter integration algorithm requires that the client and the server run the same OT algorithms. ShareDB OT Types are therefore written in JavaScript in such a way that allows them to be included via the ShareDB web browser library, but also to be used by the ShareDB server, which is built on the Node.js JavaScript run-time environment.

Klokmose et al. made use of the JSON OT capabilities of ShareDB for their

project known as Webstrates [97]. Their paper explores the importance of DOM synchronization across multiple devices. Webstrates focuses on a concept referred to as "shareable dynamic media", where DOM sections know as "substrates" are shared between different users. For example, the authors show collaborative document authoring where two different editor interfaces are used to edit the same document. By having both editors subscribe to the same "substrate", the DOM contents within the editor body remain synchronized, and the surrounding buttons and editor's visual appearance can be customized based on each author's preference. While such use cases show the unique power of DOM synchronization, the underlying "json0" OT Type of ShareDB is not ideal for the XML-based nature of the DOM.

In Table 2.3, the list of operations of json0 is adapted from the project's GitHub page [98], where `[path]` is a list of keys to reach a target JSON element. As shown in an example from the project's documentation, given the JSON document `{"a":[100,200,300],"b":"hi"}`, the operation `[{p:["a",0],ld:100}]` can be used to delete the 0th element (the "100") of key "a". While the operations of Table 2.3 have been defined (along with the corresponding transformations) to reliably manipulate JSON data, mapping an intermediate JSON format to DOM-based documents introduces problems for intention preservation. Klokmose et al. recognize such limitations and state that, for example, "the Operational Transformation algorithm does not recognize move operations, which are interpreted as delete and insert" [97].

With the source code of ShareDB [96] and Webstrates [99] available in a deployable fashion on GitHub, a closer inspection of the approach was performed for this thesis. It was observed that even before a conversion of JSON to XML or DOM is attempted, concepts such as splitting or merging nodes are also not supported by the json0 OT Type. This was also confirmed in an Issue of the ShareJS GitHub project [100]. To demonstrate why this is the case, an adapted version of the scenario discovered in the Issue is repeated here. Assume a rich-text editor that operates on an object containing a key named "text" that represents the document content. The value of "text" can be a simple string or an array if multiple sections of text are required. As an item in this array, suppose a JSON object with a key of "bold" identifies the text that is to be displayed as bold within a rich-text editor.

The scenario is summarized in Figure 2.17. Alice and Bob both start with a document containing "abcd", which is represented in JSON as `{text:"abcd"}`.

TABLE 2.3: Operations supported by the json0 OT Type of ShareDB.

| Operation | Description |
|---|---|
| `p:[path,idx], li:obj` | Inserts the object `obj` before the item at `idx` in the list at `[path]`. |
| `p:[path,idx], ld:obj` | Deletes the object `obj` from the index `idx` in the list at `[path]`. |
| `p:[path,idx], ld:before, li:after` | Replaces the object `before` at the index `idx` in the list at `[path]` with the object `after`. |
| `p:[path,idx1], lm:idx2` | Moves the object at `idx1` such that the object will be at index `idx2` in the list at [path]. |
| `p:[path,key], oi:obj` | Inserts the object `obj` into the object at `[path]` with key `key`. |
| `p:[path,key], od:obj` | Deletes the object `obj` with key `key` from the object at `[path]`. |
| `p:[path,key], od:before, oi:after` | Replaces the object `before` with the object `after` at key `key` in the object at `[path]`. |
| `p:[path,offset], si:s` | Inserts the string `s` at offset `offset` into the string at `[path]`. |
| `p:[path,offset], sd:s` | Deletes the string `s` at offset `offset` from the string at `[path]`. |

Alice wishes to bold the letter "b" while Bob wishes to delete the letter "d". Based on the operations of Table 2.3, Alice can obtain a document where the letter "b" is bold by using an `od` (object delete) operation combined with an `oi` (object insert) operation to create an array that begins with `"a"`, followed by {`bold:"b"`}, followed by the remaining text `"cd"`. Alice applies this to her local document and ends up with {`text:["a",{bold:"b"},"cd"]`}. Bob can delete the letter "d" from his local version of {`text:"abcd"`} by using the `sd` operation and specifying the `offset` value as 3 to reach the position of the desired letter. Bob is then left with the local document of {`text:"abc"`}.

The desired final state of the document that meets the intentions of both users is {`text:["a",bold:"b","c"]`}, where the "b" is bold as Alice wanted and the

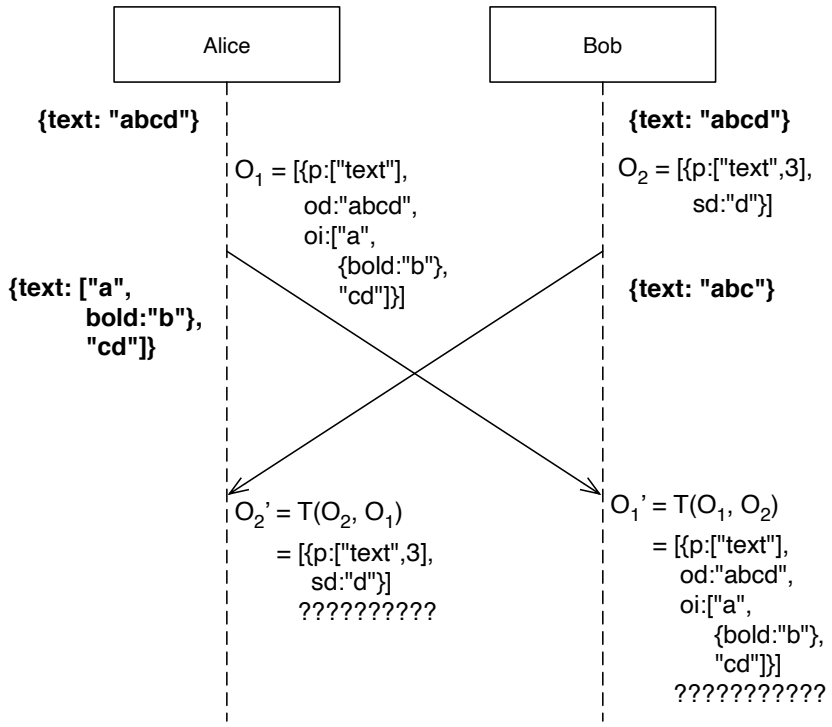



FIGURE 2.17: Attempt at using the json0 OT Type of ShareDB to split a string by introducing a "bold" section of text.

letter "d" is no longer present as Bob wanted. However, with the json0 implementation, Alice receives Bob's operation of `[p:["text",3], sd:"d"]`, and must transform it against her local operation of `[{p:["text"], od:"abcd", oi:["a",{bold:"b"},"cd"]}]`. Since the operations and transformations were not designed for editing text, the transformation leaves Bob's operation unmodified. Given Alice's document of `{text:["a",{bold:"b"},"cd"]}`, she cannot find the offset `3` within the new array and therefore does not delete anything. As a result, Bob's operation is lost. Similarly, Bob encounters a problem with Alice's (`od`) operation, which expects a deletion of an `"abcd"` string that is no longer present in his document, while applying Alice's `oi` operation would bring the letter "d" back by introducing the tailing `"cd"` section.

While many potential solutions have been discussed in the GitHub Issue since 2011 [100], the example reinforces that a generalized DOM synchronization solution requires designing the necessary operations and transformations for the direct manipulation of DOM nodes rather than intermediate JSON values. Starting with the definition of a new set of operations that includes the *giveData* operation for splitting text nodes (Section 5.2.6), this thesis will revisit the above example and demonstrate a functional solution to it in the context of real-world DOM nodes.

### 2.3.4 Generic Collaboration Infrastructure

In 2012, Heinrich et al. [30] presented the "Generic Collaboration Infrastructure" for the implementation of collaborative web applications and applied it to existing web-based rich-text editors. By converting changes to the browser's DOM into OT operations, web-based editor content was observed to remain synchronized among multiple users in real-time. In [31], Heinrich et al. present an attempt at synchronizing TinyMCE HTML DOM content. The Ph.D. thesis of Heinrich [32] elaborates on the methodology used. Heinrich did not focus on developing new OT algorithms, but rather made use of an existing groupware development library from the software company SAP. The library, known as SAP Gravity, is proprietary and little public information about it remains online. However, it can be seen from Heinrich's publications that SAP Gravity does not provide operations for moving, merging or splitting nodes. Operations for updating nodes and attributes also appear to be absent, which means that user intention cannot be preserved properly since one such operation will overwrite the other (in fact, Heinrich mentions little about intention preservation in his work). Although Heinrich succeeds at converting single-user applications into multi-user ones, the described technique appears to modify the nodes of the DOM by appending unique identifiers to assist the SAP Gravity component. The solution proposed in this thesis does not modify the original DOM other than to synchronize changes made by other users. While Heinrich relies on the older DOM3 APIs such as DOM Mutation Events [101] to detect changes, these events have been deprecated in DOM4 [33] and replaced with the more powerful *MutationObserver* object in this thesis. As there are few implementation details (especially server-side), it is difficult to evaluate many aspects of the system. However, Heinrich did explore synchronizing JavaScript variables (in addition to DOM synchronization) for a general-purpose web framework. This thesis will focus on DOM synchronization while also permitting additional key-value pairs to be synchronized, but will not address specific web frameworks.

### 2.3.5 The Pulsar System

In his 2013 Ph.D. thesis entitled "Instant Synchronization of States in Web Hypertext Applications" [102], David Linner presented a software architecture for synchronizing HTML using OT on a hierarchical data model. A set of operations

and the corresponding transformation functions were defined for resolving conflicts between concurrent operations while attempting to respect user intentions. A synchronized rich-text editor and "Bomberman" game were built on top of the developed platform, which was named "The Pulsar System". While the objectives appear similar to the work undertaken in this thesis, the set of operations that was ultimately chosen for manipulating the DOM was limited to storing characters as associative arrays. Such a model cannot support multiple text nodes within an element node. When attempting to implement the rich-text editor, Linner observed that the designed operations were insufficient, and a revised set of operations was proposed that unfortunately still omitted operations such as splitting, merging or moving nodes, all of which are required for properly preserving the user's intentions. The Pulsar System also focused on the inclusion of the *fairness* property that allows multiple users to perceive the effect of an action (such as a response from the server) at the same time. While this property is important for certain types of competitive games, it was implemented using pessimistic synchronization techniques, and ultimately the real-time game implementation was found to be "sluggish" due to the communication requirements of this pessimistic approach (a response is needed from the server in order to perform any action). The work also did not discuss persisting the data or operations, and found "shortcomings of implementation with regard to scalability". The database and cloud scalability aspects of the architecture proposed in this thesis are given special attention as they represent important industry trends.

### 2.3.6 Firebase

Backend-as-a-Service (BaaS) and "serverless" concepts have been found to be increasingly important in meeting the scalability and reliability requirements of modern web and mobile applications [10]. Now part of the commercial cloud offerings of Google, Firebase [11] is a backend-as-a-service (BaaS) platform that allows web clients that include the Firebase JavaScript library (among many other supported client libraries) to access a scalable cloud-based database. An overview of how Firebase can scale such "serverless" database functionality to support over a million concurrent users [103] is available in a technical talk given by former Firebase developer Greg Soltis [104]. Data is stored in the JSON format and security features are provided to restrict read or write access to specific values. WebSocket

connections [105] to Firebase are kept active and the client is instantly notified of data model updates through a publisher/subscriber architecture.

First introduced in 1987 by Birman and Joseph [106], the publisher/subscriber architecture has been the foundation of most modern real-time communication research for its ability to instantly *push* data to clients, rather than requiring clients to continuously poll for updates at regular intervals. Data publishers categorize their messages into different *topics*. They do not need to know how many subscribers there will be (if any) or what the subscribers will do with the received messages. Subscribers signal their interest in specific message topics, generally without knowing if there are any publishers for that topic. A message broker can be used to allow publishers to send messages on a specific topic and to allow subscribers to register to a specific topic. This way, publishers and subscribers can operate independently of one-another. This is different from traditional client/server communication, where the client is unable to send messages to the server if the server is not running, and the server is unable to receive messages from the client when the client is not running.

By acting as a message broker, Firebase can therefore synchronize data between multiple clients in real-time as they write and subscribe to different topics identified by paths within the database service. New data of one client will be persisted once it reaches the Firebase server and will then instantly be received by the other subscribed clients. For example, the browser can make calls such as `myRef = new Firebase("https://.../myPath")` to identify the path to the database location `/myPath` where a value should be stored, and then simply call `myRef.set('myValue')` to have the key-value pair `{myPath:"myValue"}` be persisted in the Firebase database. In addition to *.set()*, the *.on()* function from the Firebase API can be used to listen to a location in the database for changes, such as the insertion, addition, removal and modification of children under the given path. The Firebase website provides a visual browser of the database contents to help manage paths, and the Firebase JavaScript API is thoroughly documented.

Of particular interest for the timestamp approach used in this thesis is an API function called *transaction()* [107], which allows atomically setting a value in the database only if a value doesn't already exist at that location (the *.set()* function always overwrites the location). When combined with numeric location names that are incremented by the clients, it becomes possible to manage submitted data as log entries or revisions in a document history. Figure 2.18 shows an example where

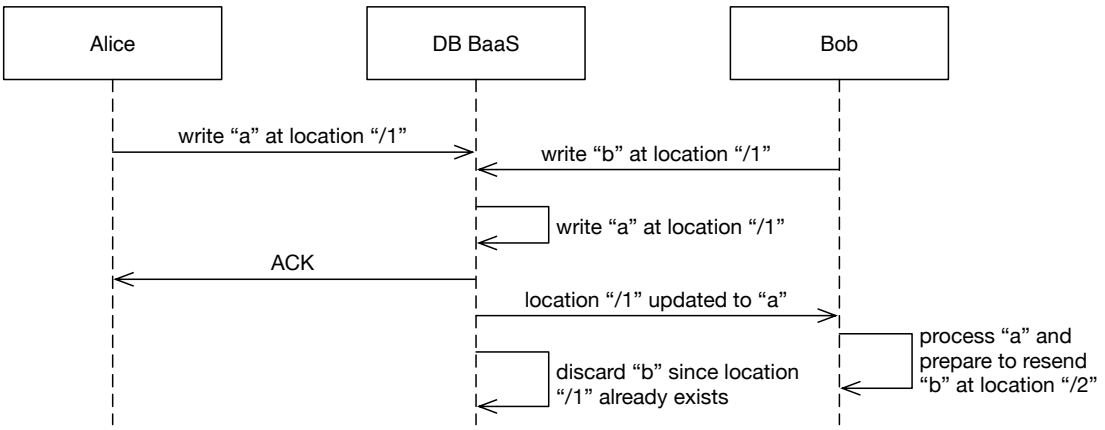



FIGURE 2.18: Using numeric locations in a serverless database to achieve a global history of submitted data.

two browser-based users, Alice and Bob, access the BaaS functions of a serverless database provider such as Firebase (the actor “DB Baas”). Alice wishes to add the letter “a” to a list, while Bob wishes to add the letter “b”. Alice’s transaction request for the location “/1” is received by the DB BaaS first, while Bob’s request is placed in a queue. The DB BaaS persists the “a” at location “/1” and sends an acknowledgement to Alice. Bob, who had subscribed to receive updates to any changes to the data under “/” of the DB BaaS, is notified of the added `{"1":"a"}` value. The “DB BaaS” then processes Bob’s request for location “/1” and rejects it since the location was already occupied. Bob must therefore prepare to re-submit his letter “b” against location “/2” in hopes of receiving an acknowledgement and making it into the “official” record of the DB BaaS. If Bob were to request location “/1” again after the flow of Figure 2.18 (due to logic issues, reconnection issues etc.), it would again be rejected by the server. By using operations instead of simple letters, an authoritative version of a document (“ground truth”) can be established, where the ordered entries can be replayed to obtain the final state.

Firebase has released an example that makes use of the open source *ot.js* library [108] (further discussed in Chapter 3) to create a collaborative real-time editor named Firepad [109]. As mentioned in the comparison of Section 2.3.2, the underlying OT makes use of the string-based annotation method similar to Google Wave, and therefore cannot perform complex operations required for DOM synchronization (in fact, the demo editor on the Firepad webpage does not support tables). Even though the Firebase team noticed that “[real-time text editing] was the primary use-case for many of our developers” [110], the Firepad example was released in 2013 and has not evolved significantly since then. However, several

startup companies have adapted the Firepad example to create useful products. For example, Koding [111] provides a collaborative Integrated Development Environment (IDE) that allows multiple programmers to co-edit source code together (among chat and other features that make use of Firebase). The Firebase platform therefore provides many useful and reliable features for the development of multi-user collaborative applications, including scalable multi-protocol bidirectional access to persisted key-value data in real-time, as other academic projects have discovered [112][113][114]. In particular, the Firepad project explores how an OT algorithm can be made to function without the need for server-side transformation logic. Rather than having the client wait on the server to return a transformed operation, this approach requires the client to re-transform and re-submit any operations previously rejected by the server. Since such approaches have not been investigated in existing OT literature, the transformation-less approach of Firepad will be further analyzed and extended as part of the new Control Loop-based OT integration algorithm (CLOT) proposed in this thesis (Chapter 4). With a focus on DOM-based synchronization, CLOT has been generalized to not depend on the services of any specific cloud provider while retaining the "serverless" properties in high demand for today's web-based deployments.

### 2.3.7 Other Relevant Work

Nicolaescu et al. [115] introduced an open source framework named Yjs [116]. Yjs supports XML synchronization based on CmRDT principles that enable a peer-to-peer approach [117][118], although complex operations such the moving of nodes or changing individual characters in an attribute value are not provided. In addition, Yjs does not persist the document state in a database.

André et al. [119][65] attempted to define a set of operations for keeping the rich-text editor content from the XWiki [19] content environment synchronized. The editor component in XWiki was restricted to a tree-based structure that consisted of multiple paragraph nodes, each containing one or more content nodes, which are either text nodes or style nodes with a limited set of styles. Ignat et al. [120] later refined this work into a set of eight operations for keeping the rich-text editor content from the XWiki [19] wiki system synchronized. For such a document model, they identified the operations shown in Table 2.4. Rather than addressing the full DOM specification, an intermediate tree-based structure was

TABLE 2.4: Operations identified by Ignat et al. for XWiki document synchronization.

| Operation | Description |
|---|---|
| **InsertText** | Insert one character in a text node. |
| **DeleteText** | Delete one character in a text node. |
| **NewElement** | Create a new element with an empty text node. |
| **UpdateElement** | Update a new element with an empty text node. |
| **MoveElement** | Move one element. |
| **MergeElement** | Merge two adjacent elements. |
| **Split** | Split one paragraph in two. |
| **Style** | Add/delete a style to some text. |

developed that can be mapped to the XWiki "wikitext" document model as well as to HTML for the XWiki WYSIWYG editor. Elements in this document model can only be of type *paragraph*, *heading*, *list*, *table* or *template*, and cannot be inserted within the same node as existing text content, among other restrictions. The eight operations identified include intention-preserving versions of *Split*, *MergeElement* and *MoveElement* operations. However, the general HTML DOM is more complex than the document model of XWiki and requires a different set of operations, including additional operations to deal with attributes and nested content. For example, the use of only `<p>` tags to represent paragraphs is unrealistic, as some DOM structures may use `<div>` or `<span>` tags (or combinations thereof). The authors also did not explore the aspect of scaling the conflict resolution component as part of a larger web application architecture.

As updates to the live browser DOM are computationally expensive due to the layout calculations that the browser must perform, the React.js web framework first applies component changes to an in-memory "Virtual DOM tree" (VDOM), and periodically determines the differences between the real DOM and VDOM in order to obtain the minimum set of changes to apply to the real DOM [121]. Voutilainen et al. [122] applied a similar VDOM technique to obtain "patch operations" that can be transmitted to remote users to achieve a "synchronized DOM state between the browsers". However, as existing VDOM approaches are focused on

browser rendering performance rather than user intention preservation, they don't account for the moving or splitting of nodes, and the operations they generate are not ideal for use with real-time collaboration applications.

Based on this examination of existing academic and commercial solutions, Chapter 3 will review the motivations for a new architecture that uses Finite State Machines (FSMs) to enable OT-based DOM synchronization to a degree not explored by previous approaches and includes support for "serverless" deployments. The requirements and FSM-based high-level design of the proposed architecture are then presented.

# Chapter 3

# High-Level FSM Model Design for Collaborative Platforms

This chapter presents a high-level architecture for analyzing and implementing distributed conflict resolution algorithms as dynamic real-time systems. The architecture includes a real-time feedback control mechanism where operations are transmitted to other users as a feedback signal via a central server. This will enable the architecture to be modeled and simulated using formal groups of hierarchical Finite State Machines (FSMs). The Web-Based Collaborative Platform is built on the architecture to provide scalable "serverless" application development based on DOM synchronization within a web browser.

In order to "increase the likelihood of a successful realization", Selic et al. [123] recommend the creation of design models for managing the complexity of distributed real-time systems, since models can be systematically modified, analyzed and verified. A model-based approach will therefore be used for the development of the Web-Based Collaborative Platform proposed in this thesis. In the book of Selic et al. [123], real-time systems are defined as systems which exhibit a dynamic internal structure and have to perform their functions "on time". The term *real-time* refers to the measurement in time between an event's occurrence and when a user is able to perceive the resulting effect of that event [124]. Depending on the nature of the application, events need to be delivered in a short enough amount of time for them to still be relevant and useful for the user. An interactivity time

constraint of 100ms is generally recommended for multi-user co-editing applications with good usability [125], and this timeliness requirement remains important for modern web-based interfaces [126].

It is important to note that "real-time" in the context of the web is often called "near real-time" since a "best-effort" approach is generally taken to minimize delays throughout the architecture. All JavaScript events (such as timer events or mouse actions on DOM nodes) are collected in the same queue where the event callbacks are processed sequentially, meaning that any new event on the client side must wait for the current function or method to complete execution before the event can be attended to. In addition, specialized techniques for dealing with the delay inherent of network-based communication systems (such as Quality of Service) are not assumed to be used since the architecture should emphasize ease of accessibility (for system integrators as well as end users) in addition to supporting real-time data.

The architecture proposed in this thesis uses optimistic consistency control methodologies that allow immediate rendering of a local modification (such as typing) rather than waiting for confirmation from the network. As the dynamic nature of network links (latency, reliability etc.) cannot be avoided, the architecture must also handle editing conflicts efficiently. The Operational Transformation approach must therefore use small operations that precisely pinpoint changes to content; a property which also enables improved intention preservation, as discussed in Section 2.2.1.4. As an example of such enhanced intention preservation, a deletion of a paragraph should not require an operation for the deletion of every single letter inside the paragraph.

## 3.1 Motivations for a New Architecture

Recent advancements in web technologies have enabled real-time collaborative applications to run inside web browsers, and commercial products such as Google Docs [14] have shown how this technology helps teams to be more connected and productive. Increasingly, real-time collaborative applications have become essential tools for allowing users and enterprises to reduce the time required to complete tasks as a group. Of particular interest are scalable and reliable web-based applications which are accessible by remote workers and mobile devices.

The rising popularity of such groupware products has created a demand for an architecture that enables other types of web applications to provide real-time collaboration functionality. In order to enable such flexibility, however, the architecture must support the synchronization of the HTML Document Object Model and any changes that web applications may make to it. Since users expect the ability to edit the same document concurrently, the architecture needs to provide a conflict resolution mechanism that best preserves each user's intentions while ensuring the documents of all users converge to a consistent state. The architecture must also provide a method for cloud-based scaling such that the application remains available with increasing user demand, as is now expected of modern web deployments.

Although a good amount of time has elapsed since the first Operational Transformation (OT) algorithms and viable implementations were devised, OT remains a rapidly evolving technology which continues to generate research challenges motivated by useful collaborative applications. As was shown in Section 2.2.2, newer variations on OT such as Commutative Replicated Data Types (CmRDTs) have also appeared in recent years. Research regarding real-time behavior of co-editing applications is in high demand for various types of content, including tree-based DOM structures. However, no solution currently exists for keeping an arbitrary web application's DOM structure properly synchronized among multiple users, as a large number of operations is required to describe all possible DOM changes. Each operation must be precise and specific so that it can be combined with another (potentially conflicting) operation without losing the intended effects of either user's change. Through the use of new algorithms based on the established optimistic consistency control principles presented in Chapter 2, editing conflicts need to be resolved in such a way that the user's intentions for the DOM modifications are preserved.

An architecture is therefore needed that can capture the dynamic nature of collaborative distributed systems in a way that facilitates the verification of the FSM and the implementation and investigation of the distributed system. During all these complex and fine grained states and transitions, the system complexity must remain manageable as client-server interactions are modeled and simulated to ensure that the architecture enables new advanced data structures like DOM to be synchronized. Once the models have been verified to a high degree of confidence, the implementation of a Web-Based Collaborative Platform must follow to

demonstrate the real-world usefulness of the intention-preserving operations and the capability of synchronizing DOM changes among multiple users.

## 3.2 FSMs & DOM

In this thesis, a new approach regarding OT, along with the corresponding architecture and new functionalities of OT, are developed. The approach resides in the interpretation of the functionality of the Web-Based Collaborative Platform as FSMs. In this respect, a series of FSMs cooperating to enable the synchronous collaboration of a certain document (consisting of any content) are devised, thereby rendering the Web-Based Collaborative Platform controllable.

In what follows, FSMs are considered formal methods used to describe the functionality of many Discrete Systems represented by a fixed number of unique states and certain transitions between them. OT falls into this category as well. The following advantages are obtained by applying FSM theory to OT:

- The causality-related properties of the approach are made obvious by the input-output relationships of FSMs.
- FSMs allow establishing a clear separation of inputs and outputs of the OT building blocks, which allows them to be connected in series, parallel and feedback configurations.
- The description of an FSM requires specifying and implementing the transition functions which describe the OT transitions from state to state; this will allow modeling the transformation-related rules required of OT systems.
- A special FSM block of the architecture is designated as the *Controller* which allows the introduction of causality-related logic to determine which operations are accepted based on the transitions triggered.
- Assuming that all possible transitions have been specified correctly, the FSM models will imply that the causality, convergence and intention preservation properties (*consistency model*) of OT systems have been demonstrated.

Two aspects are crucial for collaborative editing applications: the ability to *generate* operations, and the ability to *apply* operations. The generation of operations

can be achieved by observing changes to the state of the local editor HTML Document Object Model (DOM). Such changes are typically triggered by keyboard or mouse input from the user. The *application* of operations requires the use of the DOM APIs of the web browser to invoke a type of change in a specific part of the DOM hierarchy as supported by the W3C specification [33] and described by the operation. A generalized solution would imply capturing all possible DOM changes independent of any underlying editor or DOM-manipulating application. These DOM modifications then need to be converted into operations that allow multiple users to reach the same document state at all times. The operations must be designed to meet the requirements of Operational Transformation algorithms, including intention preservation [4]. The operations therefore need to be as specific as possible so that localized updates to a DOM can be made, thereby reducing the chance of conflicts between operations. While new concurrency algorithms continue to appear in this very active area of research, so far no approach exists for accomplishing this task.

Most existing OT approaches are generally designed to operate on plaintext strings with only two or three distinct operation types. The solution proposed in this thesis goes well beyond existing work and aims to resolve conflicts automatically by introducing new change detection and application algorithms based on Virtual DOM principles. Such principles support the management of the complex set of required operations such as moving, splitting and merging of tree-based DOM nodes. While several academic publications, commercial projects, and open source concurrency control libraries claim to synchronize rich-text content and even hierarchical or graph-based data types, no solution currently exists that contains the concurrency control functionality necessary for resolving conflicts on a W3C-compliant DOM to reach a consistent state that is not only correct, but also matches the user's expectations in more cases than previously possible. Once modeled and simulated, this real-time feedback control system is used as the foundation for a "serverless" Web-Based Collaborative Platform that simplifies the development of DOM-based distributed applications by handling many of their complexities, as will be shown with a collaborative rich-text editor and a multi-user 3D virtual environment.

In order to provide the state-of-the-art interactivity that is characteristic of the modern web, HTML5 web pages can now exhibit a large variety of complex DOM behavior. For example, by monitoring the changes that occur within existing web

applications such as popular WYSIWYG editors, many unusual adjustments can be observed, such as hidden element nodes that appear then instantly disappear, text nodes that are split into multiple adjacent text nodes, and empty text nodes that are not normalized. In order to ensure that all collaborating users remain synchronized with an identical DOM, as many of such possible changes as possible (as guided by the DOM4 specification [33]) need to be accounted for. Managing the complex set of operations required for encoding DOM changes while still preserving user intentions therefore remains an unsolved problem.

## 3.3 Architecture Requirements

In order to enable the development of a Web-Based Collaborative Platform, the following requirements were determined for the architecture presented in this thesis:

1. FSM-Based Design

    1.1. An FSM-based design is to be used throughout the architecture to help organize and simplify OT algorithms, thereby assisting developers with their understanding of OT-related concepts and challenges. This way, the system can be extended with the new operations and transformation functions in a clean and manageable fashion, which is vital to meeting the "DOM Support" requirements below.

    1.2. A centralized method of maintaining a global order of messages in a distributed system is required to ensure that all users of the system remain synchronized. For example, Figure 3.1 shows how a *Controller FSM* can interact with multiple *Collaborating Client FSM* instances in a centralized fashion. Several centralized OT techniques have been established in previous academic research [55][6][7] but will need to be adapted for use with the other components of this architecture and will omit server-side transformations in order to meet the "serverless" requirements below. In this thesis, this approach will be known as the Control Loop-based OT integration algorithm (CLOT).

2. DOM Support

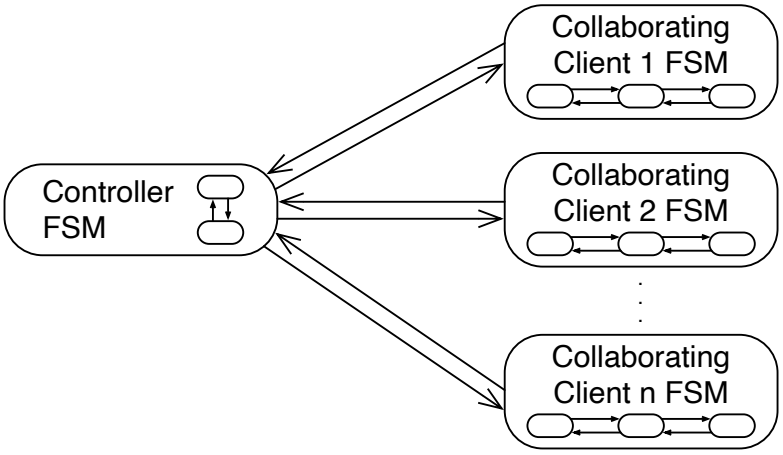



FIGURE 3.1: Centralized FSM-based collaboration architecture.

2.1. A DOM synchronization technique is required to keep the markup of the web browser synchronized. The technique must be capable of XML-based synchronization that supports concurrent changes to the hierarchical node structure, as well as to individual characters at the text node level (strings). The concurrency aspect can be accomplished using mechanisms such as Operational Transformations [67] or Commutative Replicated Data Types (CmRDTs) [72].

2.2. A "diff" component is required that is capable of detecting any possible DOM change in the browser and creating the necessary operations to summarize that change. This can be achieved by comparing the state of the DOM before and after a change, while using Virtual DOM concepts to ensure that changes can be summarized in such a way that they can be applied on the different DOM of another user. Transformation logic is also required based on the new set of operations that are able to better maintain the property of intention preservation [4] when multiple users are concurrently editing a document. New algorithms are required in order to enable operations capable of splitting text nodes and to support as much of the W3C DOM4 Specification [33] as possible.

3. Serverless

3.1. An object synchronization library is required that is capable of storing data locally in the browser and synchronizing with a key-value database whenever possible (thereby supporting intermittent connections of mobile devices; this approach is known as "offline first"). Products in this category also make use of a publisher/subscriber mechanism such that

multiple clients can be subscribed to the data and receive real-time updates as the data changes in the database. PouchDB [127] is an example of such a system based on the CouchDB NoSQL database. The Meteor web framework [128] makes use of a "livequery" mechanism on top of MongoDB to notify subscribers of model updates. Backend-as-a-Service providers such as Firebase [11] (sometimes called *serverless database* providers) and open source databases such as RethinkDB [129] are also designed to provide such bi-directional synchronization capabilities to JavaScript clients with support for atomically creating new values in the database to assist with the global message order requirement identified above.

3.2. Increasingly, Backend-as-a-Service (BaaS) [11] and "serverless" [10] concepts have been shown to be instrumental in meeting the scalability and reliability requirements of modern web and mobile applications. The proposed architecture should therefore be built on "serverless" principles so that it can be scaled to handle an increasing number of collaborative users and sessions as demand for the services grows.

Section 8.4 will revisit the above requirements in order to determine how the resulting Web-Based Collaborative Platform fares in comparison to the existing approaches presented in Chapter 2.

## 3.4 Architecture and Component Design

In this thesis, the OT algorithm structure is to be modeled as a real-time system using Finite State Machine (FSM) theory. To start this process, Figure 3.2 represents a co-editing platform modeled using a basic real-time system architecture in which a server-side *Central Controller* receives the operations executed by a number of *Clients* who are modifying the same content via an editing application. By typing inside its *Editor Instance*, a client introduces a disturbance into the control loop as a new operation. The *Central Controller* then behaves as a publisher/subscriber (pubsub) message broker component (previously defined in Section 2.3.6), redistributing the operation to all clients of a collaborative session. The originating client will interpret the return of its own operation as an acknowledgement, while all other clients will apply the received operation to their

local state, possibly transforming the operation first in the case of a local conflict. An *Events Generator*, itself represented as a FSM, can be used to exercise these components and evaluate their behavior.

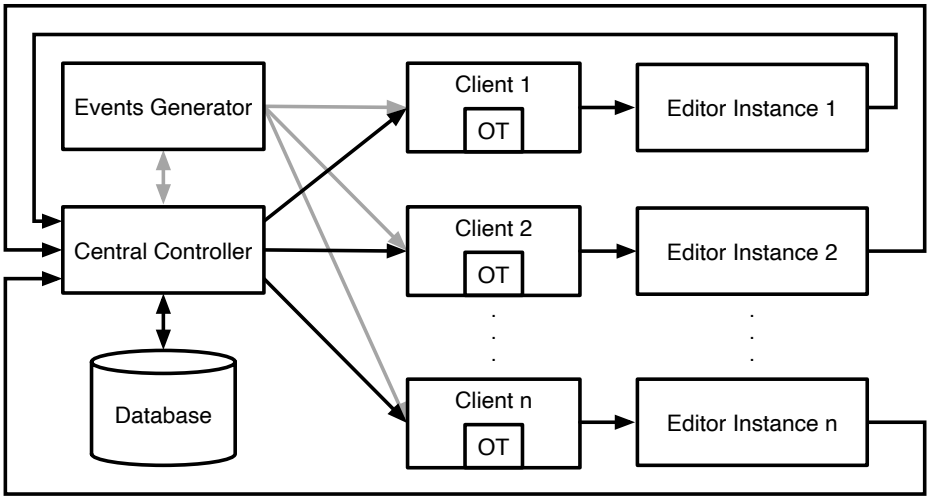



FIGURE 3.2: Centralized OT-based collaboration architecture.

With additional refinement, the Web-Based Collaborative Platform was structured as a real-time feedback control system as shown in Figure 3.3. The feedback data is generated from the local web-based editor (shown as *Browser Editor*) in the form of DOM objects via the *GET DOM* block. The *Feedback DOM Observer* (FDO) ensures that the DOM produced at time $(k)$ is collected and stored as a VDOM such that its latest state $(k+1)$ and previous state $(k)$ are always available. A *VDOM Map* is also created to ensure DOM node references remain useful, as will be explained in Chapter 5. A *Comparison Logic* block uses Rule-Based logic to determine the change in the DOM and generate the corresponding set of operations based on the operation format described in Section 5.1 for sending remotely with a new timestamp. A *Central Controller* (collaboration server) enforces a global order via a *Timestamp Logic* component that only accepts operations bearing new timestamps $(k+1)$. For new operations, the *Central Controller* then acts as a publisher/subscriber-based message broker (*PubSub Broker*) to broadcast the new operation to all users subscribed to the same session identifier. Operations are received by a *Decision Maker* block that uses additional Rule-Based components to decide how to transform any incoming operation for local application, if necessary. Finally, a *SET DOM* block then applies the change locally on the client to update the DOM of the browser.

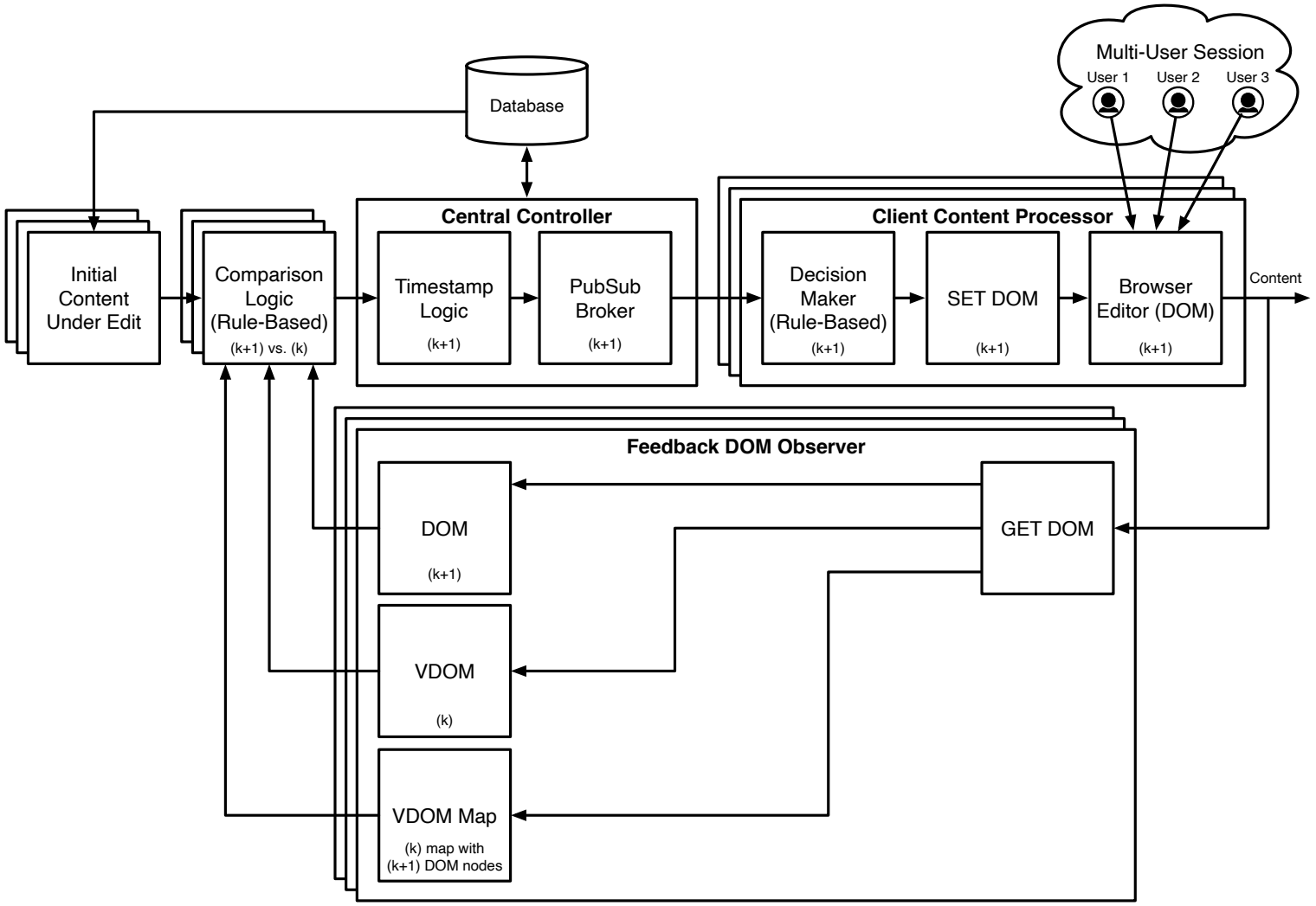



FIGURE 3.3: Web-Based Collaborative Platform modeled as a Real-Time Feedback Control System.

When new changes are created by a user (such as typing inside an editor), the *Browser Editor* is instantly updated to display the change, and the resulting DOM content ($k+1$) is sent to the *Comparison Logic* block, where the difference is determined and operations are generated for sending to other users. Upon initialization of the client, a starting VDOM value must therefore be assigned (as time $k$) in order for the *Comparison Logic* block to operate correctly. This *Initial Content Under Edit* may originate from a predefined template or database entry.

The key components of the architecture can therefore be summarized as follows:

- The *Feedback DOM Observer* (FDO) consists of the *GET DOM*, *DOM*, *VDOM* and *VDOM Map* blocks. The FDO stores the latest DOM, previous VDOM, as well as a VDOM Map required by the *Comparison Logic* block to obtain a set of transformable operations describing the changes detected in the DOM.

- The *Comparison Logic* block makes use of the stored DOM, VDOM, and VDOM Map and a Rule-Based "diff" algorithm to produce operations that summarize and encode the changes in the local DOM for sending to the *Central Controller* with an incremented timestamp value. In WYSIWYG editors such as TinyMCE, user actions such as the simple addition of one letter may result in a set of several DOM-level operations (delete a temporary element, then create a new text node, then add a letter to the text node etc.).

- The *Central Controller* (henceforth simply *Controller*) is a collaboration server that checks the timestamp value of the received set of operations, and keeps a global order [55] on the execution of the above operations. Whenever a change takes place within the *Editor Instance* (such as a typed letter modifying the DOM of a web-based WYSIWYG editor), the change is published as an operation to all users. In addition, the *Controller* can optionally be connected to a database for storing operations, user access control properties, and to address other key-value data persistence needs of the web application. By keeping their design simple in this fashion, the *Controller* and the database can together be realized with a "serverless" cloud platform (e.g. Firebase).

- The *Client Content Processor* block consists of the *Decision Maker*, the *SET DOM* block and the *Browser Editor* block. The *Decision Maker* contains

the Operational Transformation logic and uses a Rule-Based approach to transform the remote operations against the local ones to produce operations that can be applied locally. The *SET DOM* component applies the transformation locally on the client to update the DOM of the *Browser Editor* consisting of a WYSIWYG editor such as TinyMCE.

This thesis will show how a Web-Based Collaborative Platform can be implemented by recognizing the components and guiding principles outlined in this architecture. Finite State Machine (FSM) theory is used to model the key components and algorithms required for the implementation of the DOM-based synchronization platform.

## 3.5 High-Level FSMs for Operational Transformations

As the interaction between users and the server is dynamic, elements of the OT architecture will now be modeled as real-time processes using FSMs. A high-level FSM must first be established to define the computational flow of an Operational Transformation algorithm with single-character plaintext data. The clean FSM-based organization will then be evolved in a step-by-step manner towards supporting advanced new DOM-based documents and operations.

The initial approach makes use of the *Jupiter* algorithm of Nichols et al. [6] and assumes the presence of a centralized server that is able to perform the same transformations as the clients. A closer look at the *Jupiter* state space graphs of Figure 2.12 of Section 2.2.1.5, when combined with the improved individual operation acknowledgement approach of *Wave* described in Section 2.3.2, reveals the following client-side pattern for user Alice:

1. Alice is initially idle at state $(0, 0)$, although she may receive new operations from the server while in this state.

2. Alice types a key, sees the key displayed in her local document, and sends the key to the server to reach state $(1, 0)$.

3. While in state $(1, 0)$, Alice may receive an operation from the server, which she must transform before applying to her local document, **or** she may receive an acknowledgement (ACK) for her previously-sent operation.

From this description, an initial FSM-based model consisting of two states, *Synced* and *AwaitingACK*, is deduced as shown in Figure 3.4. As is typical for such diagrams, a slash (/) separates an event from an action in the label on a state transition arrow. It is worth noting that the *xform* action in the figure implies both possible transformations are performed as per Equation 2.4 of Section 2.2.1.5. Alice needs to transform the received operation to account for her local operation, and must also transform her local operation in anticipation of an acknowledgement from the server containing the updated version of her local operation. This design is therefore based on the *Wave* approach and assumes the server is capable of performing transformations. Table 3.1 shows a state table that represents the figure as a matrix.

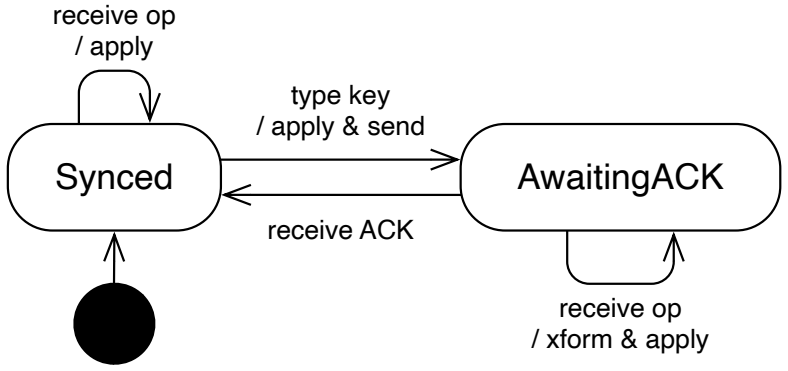



FIGURE 3.4: Two-state client FSM-based OT implementation.

TABLE 3.1: State table of a two-state FSM-based OT implementation.

| **Input** / **Present State** | **type key** | **receive op** | **receive ACK** |
|---|---|---|---|
| **Synced** | apply & send, AwaitingACK | apply, Synced | – |
| **AwaitingACK** | – | xform & apply, AwaitingACK | Synced |

However, this simplified two-state design does not account for the fact that Alice may type additional letters while awaiting a server reply in the *AwaitingACK* state. Since the responsiveness requirement of OT systems states that new local

input must immediately be applied to the local document for display to the user [3], the *AwaitingACK* state must be modified to accept a *type key* event that leads to an *AwaitingWithBuffer* state responsible for queuing up multiple additional local typing operations, as shown in Figure 3.5.

It is therefore possible to represent the functionality of an OT client-server system as a simple three-state FSM consisting of *Synced*, *AwaitingACK* and *AwaitingWithBuffer* states. To start, the client enters the *Synced* state. By typing a key, the key is applied locally and the operation representing the key is sent on the network (*apply & send*). The client transitions to the *AwaitingACK* state. At this point, the client may receive an acknowledgement from the server indicating that the operation has been accepted into the server's official history of the document. The client then returns to the *Synced* state. However, due to network delays, the client may receive an operation from another client (via the server) while in the *AwaitingACK* state. When this happens, the client needs to transform the remote operation against its local operation before it can apply the incoming change to the editor (*xform & apply*). In addition, the client also needs to transform its local copy of the previously-sent operation so that it is awaiting a transformed version of the operation from the server. While in the *AwaitingACK* state, the client might also receive additional local user input. When this happens, the client enters an *AwaitingWithBuffer* state, within which it continues to transform and apply any incoming operations (that are not an *ACK*) and buffers any new local operations (*apply & buffer*). When an *ACK* is finally received, it sends the entire buffer (*send buffer*) and returns to the *AwaitingACK* state to await the *ACK* for these buffered operations. A state table of this design is given in Table 3.2.

TABLE 3.2: State table of a three-state FSM-based OT implementation.

| **Input** / **Present State** | **type key** | **receive op** | **receive ACK** |
|---|---|---|---|
| **Synced** | apply & send, AwaitingACK | apply, Synced | – |
| **AwaitingACK** | apply & buffer, AwaitingWithBuffer | xform & apply, AwaitingACK | Synced |
| **AwaitingWithBuffer** | apply & buffer, AwaitingWithBuffer | xform & apply, AwaitingWithBuffer | send buffer, AwaitingACK |

Inspired by the web-based Operational Transformation foundation of *Wave* introduced in Section 2.3.2, several open source projects have attempted to simplify

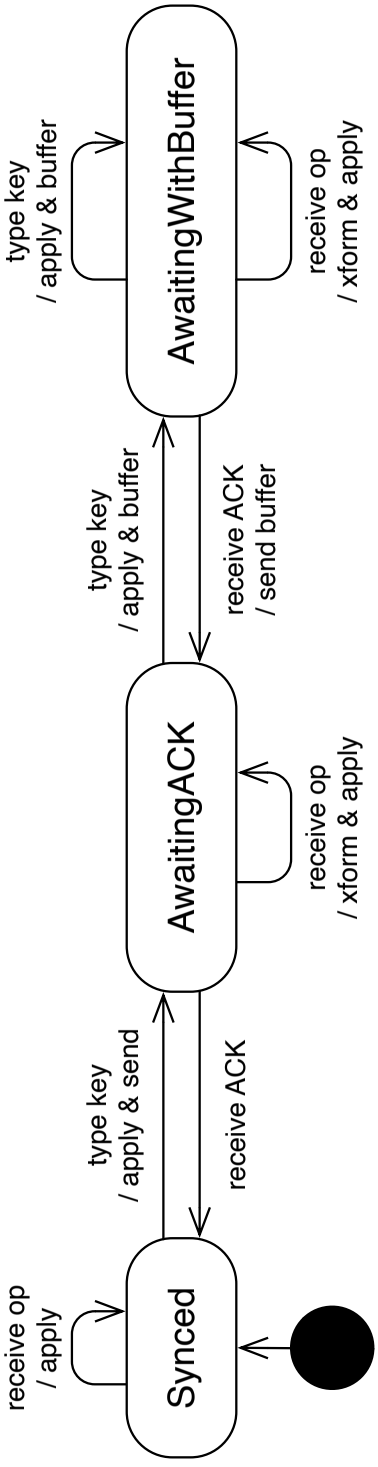



FIGURE 3.5: Three-state client FSM-based OT implementation.

or re-implement the concepts of *Wave* in different programming languages and styles. The Scala-based *Common Collaborative Coding Protocol* (CCCP) project of Daniel Spiewak [89] is an example of a project that is able to reuse *Wave* components, and the *ot.js* project of Tim Baumann [108] features a JavaScript-based implementation of CCCP using a Node.js-based server. While no existing academic publications exist about either of these projects, an analysis of their source code reveals a flow that is similar to the FSM structure proposed above.

However, as will be shown in Chapter 4, the CLOT algorithm proposed in this thesis simplifies the server-side by removing its transformation requirement and instead relies on a centralized timestamp mechanism to ensure that operations are accepted by the central server in an atomic fashion. This approach enables "serverless" web application deployments and removes a large amount of server-side processing, but performing operational transformations on hierarchical DOM-based structures still requires significant complexity on the client-side. DOM-based synchronization was therefore not explored by other "serverless" implementations such as Firepad [109]. In this thesis, the principles of control theory and Finite State Machines (FSMs) are used to model and organize the real-time distributed system behavior in a well-structured fashion, thereby also helping to mitigate the client-side challenges. By using the architecture of Section 3.4 as a guide, the system can be modeled as a set of *Client* instances that are coordinated in a feedback control loop by a central *Controller*. This allows the creation, redistribution and application of operations to remain manageable as the number of operations and transformation functions is increased to support as much of the DOM4 specification [33] as possible.

# Chapter 4

# Control Loop-based OT Algorithm

As defined in Section 2.2.1.2, integration algorithms in OT systems are responsible for preparing operations to be used as inputs into transformation functions, as well as for managing communication-related functionality such as the sending and receiving of operations between clients. By working in unison with transformation functions, integration algorithms ensure that the causality preservation (Section 2.2.1.1), convergence (Section 2.2.1.2) and intention preservation (Section 2.2.1.3 and Section 2.2.1.4) requirements of OT systems are met.

Based on the system architecture described in Chapter 3, the desired integration algorithm has a new scalability requirement not explored in previous OT research. That is, it must support "serverless" databases which provide basic read and write API functions, where the write operations can be *transactional* or *atomically modified* for a numeric key. Firebase [11] is an example of a cloud provider offering such a service supported by reliability and scalability guarantees (see Section 2.3.6 discussion of *transaction()* function), while the open-source RethinkDB [129] project offers similar functionality for self-hosted deployments. The *Timestamp Logic* component of the *Controller* must therefore be realized with this extremely limited server-side functionality. For the *PubSub Broker* component, both products mentioned also include a real-time broadcasting mechanism that notifies browser-based clients of database changes based on the publisher/subscriber messaging model.

In order to support such "serverless" architectures, the proposed Control Loop-based OT integration algorithm (CLOT) builds on concepts from Jupiter (see Section 2.2.1.5) for client-side management of two operations at a time (thereby avoiding the TP2 condition of Section 2.2.1.2), but simplifies the server-side to not require transformation logic. While the SOCT4 integration algorithm also avoided server-side transformations (see Section 2.2.1.6), the approach used by CLOT will omit the centralized *Ticket* function required by SOCT4. Such sequencing functions imply custom server-side logic beyond what is typically offered by "serverless" databases. Instead, the proposed CLOT algorithm has clients generate incrementing timestamps and continuously retry to send their operations until their operations have been acknowledged as accepted by the server-side. The previously-mentioned transactional write model of "serverless" databases will prevent two of the same timestamp values from being written, and clients can determine that their operations have been acknowledged by receiving their own operations back in broadcasts of database updates. As in Chapter 3, the server-side is referred to as a *Controller* to highlight its control loop nature for managing such client-based retries.

In order to meet the causality preservation requirement of OT algorithms, CLOT maintains a continuous global order of operations with the use of incrementing timestamps (see Section 2.2.1.1). By ensuring that operations are only applied based on the ordered timestamp attached to each operation, CLOT also meets the convergence requirement (see Section 2.2.1.2). When transforming concurrent operations, transformation functions can be designed to achieve intention preservation by using well-established patterns, such as Algorithm 2.1 to Algorithm 2.4, and others from the Jupiter-based platforms of Section 2.3. Section 4.4 will revisit these ideas after the FSMs and corresponding algorithms have been presented.

The low-level algorithm details for the *Client* and the *Controller* will be described in the remainder of this chapter with the use of Finite State Machines. This chapter will present a way of modeling the CLOT algorithm without nested states and assumes a plain text data model where just one operation is contained within each client-server exchange. Chapter 5 will extend the work to use hierarchical FSMs with tree-based DOM structures to enable more advanced organizations of operations and new transformation functions.

## 4.1 Client FSM

The high-level FSM models introduced in Section 3.5 of Chapter 3 will now be extended to model the CLOT algorithm. Whereas the two-state FSM of Figure 3.4 contains just the *Synced* and *AwaitingACK* states, the updated design must build around them by separating reusable components (applying, transforming, sending etc.) into separate states where different stages of the CLOT algorithm may execute. For example, since the CLOT algorithm is developed based on the principle of reattempting to submit client operations with client-generated timestamps, the updated model must account for having to resend transformed operations (with an incremented timestamp) for a *receive op* event from the *AwaitingACK* state. A reusable *SendingOpToController* state is therefore modelled for such messaging. The updated model is represented in Figure 4.1, where the necessary events and their parameters are denoted on transition lines. The corresponding state table is shown in Table 4.1. The last column of the table labelled "No Input" implies the transition is taken when the present state terminates rather than with the arrival of a particular event. Rather than the generalized states of Section 3.5, the extra transitions allow for cleaner FSM logic with reusable states (for example, the *SendingOpToController* state is reached from the *ApplyingLocalOp* state, as well as the *ApplyingRemoteOpWithoutACK* state).

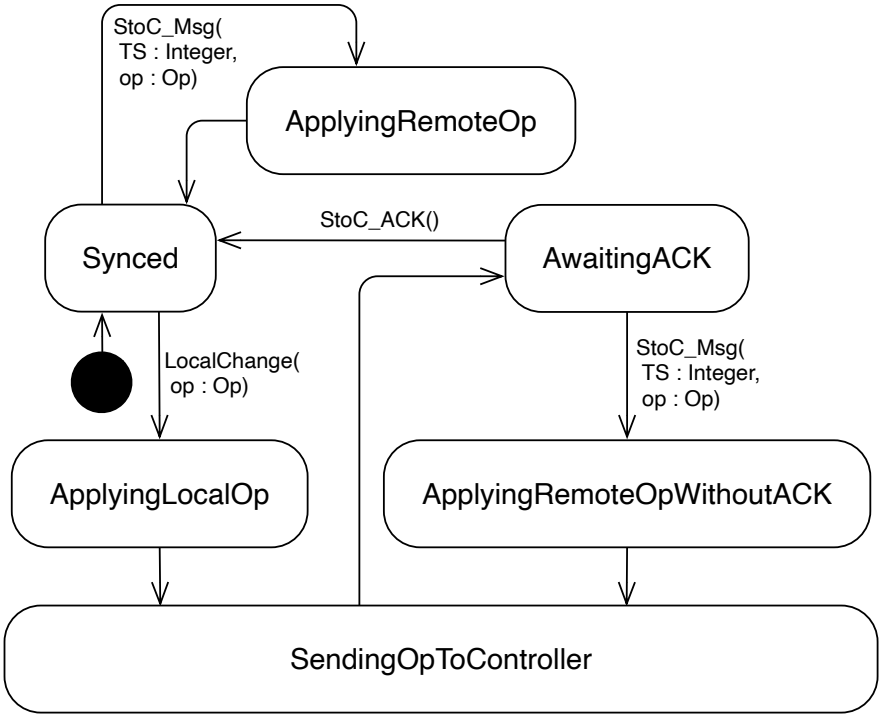



FIGURE 4.1: Basic Client Finite State Machine.

TABLE 4.1: State table for Basic Client Finite State Machine.

| Present State \ Input | **LocalChange** | **StoC_Msg** | **StoC_ACK** | **No Input** |
|---|---|---|---|---|
| **Synced** | ApplyingLocalOp | ApplyingRemoteOp | – | – |
| **AwaitingACK** | – | ApplyingRemoteOpWithoutACK | Synced | – |
| **ApplyingLocalOp** | – | – | – | SendingOpToController |
| **ApplyingRemoteOp** | – | – | – | Synced |
| **ApplyingRemoteOpWithoutACK** | – | – | – | SendingOpToController |
| **SendingOpToController** | – | – | – | AwaitingACK |

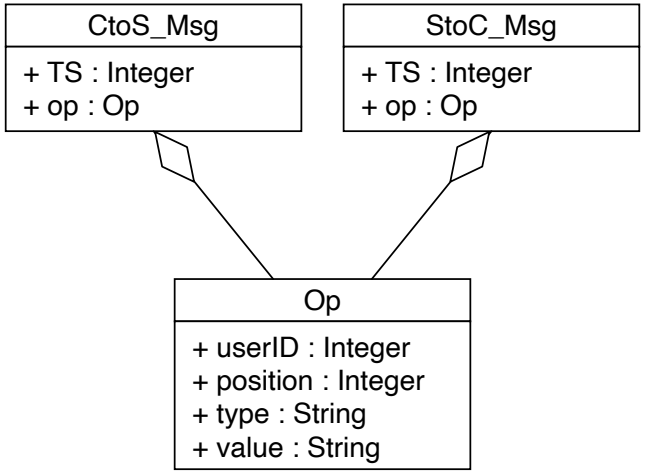



FIGURE 4.2: Class diagram of Op, CtoS_Msg and StoC_Msg data classes.

The *type key* event of Figure 3.4 is now shown as a *LocalChange* event that can be consumed by the *Synced* state and could originate from the appearance of a local change in an editing application (such as typing identified by a separate *Comparison Logic* component), or from a *Events Generator* (as previously presented in the basic feedback control loop of Figure 3.2). The *LocalChange* event contains an instance of an *Op* class. Building on the definition of an operation from Section 2.2.1.2, the *Op* class consists of four attributes to describe the change: a *userID* representing the author of the operation, an operation *position*, an operation *type*, and an operation *value*. Figure 4.2 summarizes the relationship between the *Op* class and the classes *CtoS_Msg* (where "CtoS" denotes "client-to-server") and *StoC_Msg* (where "StoC" denotes "server-to-client") explained below.

The *LocalChange* event causes a transition from the *Synced* state to the *ApplyingLocalOp* state. As shown in Algorithm 4.1, an *applyOp(op)* method within the *Client* accepts an *Op* instance and modifies the local document state to include the change described by the operation. This change is accomplished using string manipulation functions on a string named *localDoc* that represents the entire current document of the *Client*. The local timestamp value, represented by an integer named *localTS*, is incremented and the operation is transmitted to the *Controller* via the *SendingOpToController* state. As required by the CLOT algorithm, an operation's timestamp is always incremented before it is sent to the *Controller* since the *Controller* will only accept operations with new timestamps. The *Client* instance then enters the *AwaitingACK* state. The use of *applyOp(op)* corresponds to the *SET DOM* block within the *Client Content Processor* component of the architecture shown in Figure 3.3 of Chapter 3.

**Algorithm 4.1** Algorithm for a basic ApplyingLocalOp state.

1: set *localOp* to the *Op* in the received *LocalChange* event
2: increment *localTS*
3: call *applyOp(localOp)*

Both the *Synced* and *AwaitingACK* states can consume a *StoC_Msg* event when a new operation is received from the *Controller*. *StoC_ACK* events may be received while in the *AwaitingACK* state. Although clients can be implemented to interpret the return of their own previously-sent operations as an acknowledgement, a separate *StoC_ACK* event is used here for clarity.

In the *Synced* state, the reception of a *StoC_Msg* event indicates that a change was made by a remote client while the local client was synchronized and had not made

any new local changes. Similar to applying a local operation, this case therefore means that the *applyOp(op)* method is invoked to update the *localDoc* and *localTS* values based on the incoming *Op* and *TS* values contained within the *StoC_Msg*. This is performed in the *ApplyingRemoteOp* state, as shown in Algorithm 4.2.

**Algorithm 4.2** Algorithm for a basic ApplyingRemoteOp state.

1: set *remoteTS* and *remoteOp* to the values within the received *StoC_Msg* event
2: set *localTS* to the value of *remoteTS*
3: call *applyOp(remoteOp)*

The *ApplyingRemoteOpWithoutACK* state is entered if a *StoC_Msg* event is received while in the *AwaitingACK* state. Reaching the *ApplyingRemoteOpWithoutACK* state therefore means that a new operation has been received from another user (via the *Controller*) while the *StoC_ACK* event for a previous local operation has not yet been received (its older *TS* value caused it to be rejected by the *Controller*). The incoming operation must therefore be transformed before being applied to *localDoc* since *localDoc* contains a change that the incoming operation did not account for. The transformation logic within the *xform(op1, op2)* method (recall Equation 2.4 of Section 2.2.1.5) is therefore invoked by *ApplyingRemoteOpWithoutACK* using the incoming operation from the remote user as *op1* and the previous local operation (the one that is still awaiting a *StoC_ACK*) as *op2*. The timestamp value is also incremented. This corresponds to the *Decision Maker* component identified in the architecture of Chapter 3. Algorithm 4.3 shows the corresponding logic in this state before the updated *localOp* is retransmitted with a new *localTS* timestamp to the *Controller* via a *CtoS_Msg* assembled by the *SendingOpToController* state.

**Algorithm 4.3** Algorithm for a basic ApplyingRemoteOpWithoutACK state.

1: set *localTS* to *remoteTS*
2: increment *localTS*
3: set *remoteTS* and *remoteOp* to the values within the received *StoC_Msg* event
4: obtain *remoteOpPrime* and *localOpPrime* by evaluating *xform(remoteOp, localOp)*
5: call *applyOp(remoteOpPrime)*
6: set *localOp* to the value of *localOpPrime*

## 4.2 Client FSM with Buffer

In Chapter 3, the three-state FSM of Figure 3.5 introduced the requirement of an *AwaitingWithBuffer* state to ensure that user changes can always reach the local document immediately rather than appear delayed due to network round-trip times with the client blocked in the *AwaitingACK* state.

Figure 4.3 shows the complete FSM model of the *Client* that includes the logic for dealing with buffered operations, where intermediate states have again been introduced to enable a more modular organization. The state table representation is shown in Table 4.2. The three main states established in Chapter 3, namely *Synced*, *AwaitingACK*, and *AwaitingWithBuffer*, remain and all allow a *LocalChange* event to be consumed, thereby addressing the responsiveness requirement of OT systems. In addition, all three states can handle new messages from the *Controller* in the form of *StoC_Msg* events.

The state *ApplyingBufferedLocalOp* is reached with the consumption of a *LocalChange* event while in *AwaitingACK*. As per Algorithm 4.4, the state applies the new local change to the *localDoc* string while also storing it in a *opBuffer* array for future processing. The variable *opBuffer* is treated as a FIFO buffer (a queue) and is therefore assumed to have methods such as `opBuffer.first` and `opBuffer.last` for accessing the corresponding element.

**Algorithm 4.4** Algorithm for a basic ApplyingBufferedLocalOp state.

1: add *Op* from the received *LocalChange* event to *opBuffer*
2: call *applyOp(opBuffer.last)*

The state *AwaitingWithBuffer* is then reached, where the client awaits one of the three events (*LocalChange*, *StoC_Msg* or *StoC_ACK*) as it did in *AwaitingACK*. A *LocalChange* event will result with the operation being added to the *opBuffer* in the same way described above. Similar to reaching *ApplyingRemoteOpWithoutACK* from *AwaitingACK*, the state *ApplyingRemoteOpWithBuffer* is reached when a *StoC_Msg* is received while in the *AwaitingWithBuffer* state. For example, a client typed a letter, and while awaiting an acknowledgement for it, typed a few more letters (which end up in the buffer). The client then received a new remote operation, which must now be transformed against the original local operation as well as the buffer contents before it can be applied. Algorithm 4.5 must therefore first obtain a version of the newly received *remoteOp* that can be

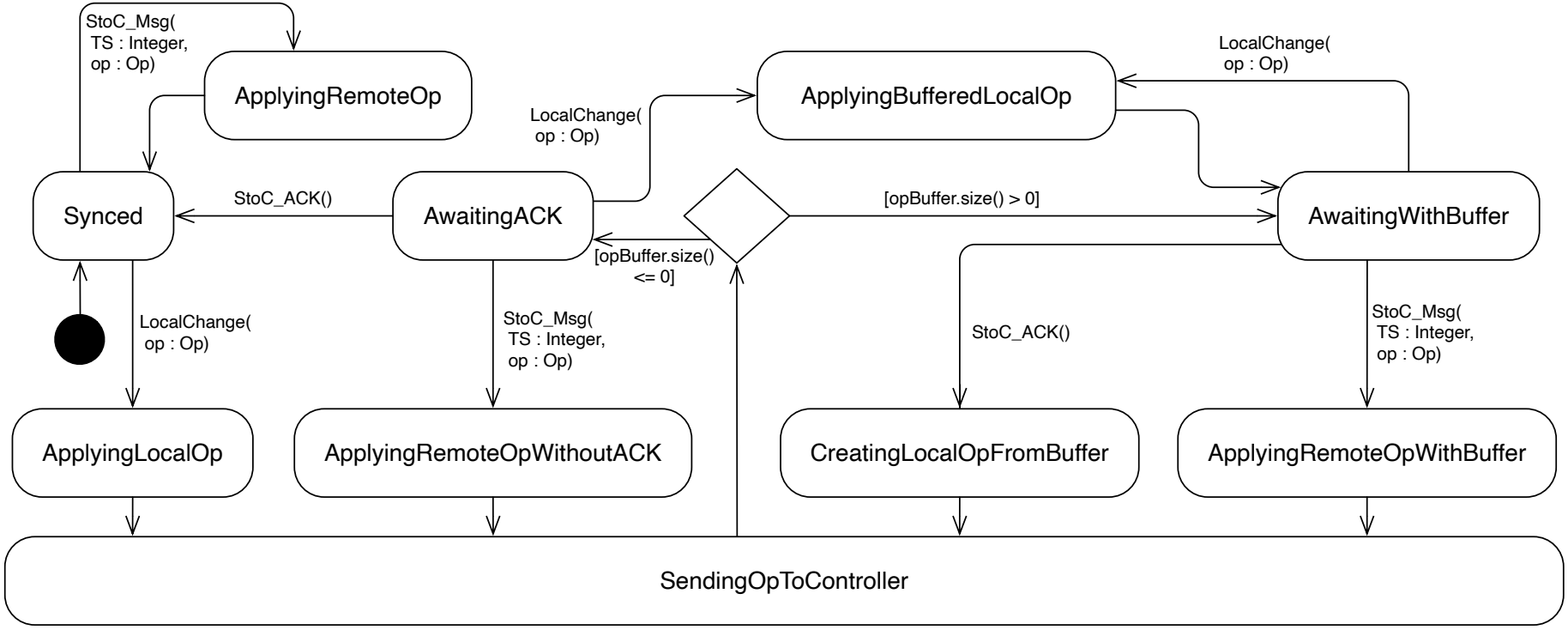



FIGURE 4.3: Complete Client Finite State Machine.

TABLE 4.2: State table for Complete Client Finite State Machine.

| Present State \ Input | **LocalChange** | **StoC_Msg** | **StoC_ACK** | **No Input** |
|---|---|---|---|---|
| **Synced** | ApplyingLocalOp | ApplyingRemoteOp | – | – |
| **AwaitingACK** | ApplyingBufferedLocalOp | ApplyingRemoteOpWithoutACK | Synced | – |
| **ApplyingLocalOp** | – | – | – | SendingOpToController |
| **ApplyingRemoteOp** | – | – | – | Synced |
| **ApplyingRemoteOpWithoutACK** | – | – | – | SendingOpToController |
| **SendingOpToController** | – | – | – | if opBuffer.size() $>$ 0 AwaitingWithBuffer |
| | | | | if opBuffer.size() $\leq$ 0 AwaitingACK |
| **ApplyingBufferedLocalOp** | – | – | – | AwaitingWithBuffer |
| **AwaitingWithBuffer** | ApplyingBufferedLocalOp | ApplyingRemoteOpWithBuffer | CreatingLocal-OpFromBuffer | – |
| **CreatingLocalOpFromBuffer** | – | – | – | SendingOpToController |
| **ApplyingRemoteOpWithBuffer** | – | – | – | SendingOpToController |

applied locally by transforming the *remoteOp* against the *localOp* (the one still awaiting a *StoC_ACK* event) and all operations in the *opBuffer* to obtain a *remoteOpPrime* array. Secondly, it must obtain a new version of *localOp* for re-sending via *SendingOpToController*, which it does by transforming *localOp* against the received *remoteOp*. Finally, the buffered operations must also be transformed against the previously-transformed *remoteOp* stages of the *remoteOpPrime* array for sending out via *CreatingLocalOpFromBuffer* (Algorithm 4.6) once the *StoC_ACK* event for the previous operation is received. A decision node is used to check if additional *opBuffer* elements are available for sending, with guard conditions denoted between solid brackets (`[]`). The client returns to the *AwaitingACK* state once the last buffered item has been sent. Although the *xform(op1, op2)* method is used in Algorithm 4.5, only the transformation of the operation passed as the first input parameter is required in all four cases and the result of the reverse transformation is not used.

**Algorithm 4.5** Algorithm for a basic ApplyingRemoteOpWithBuffer state.

1: set *localTS* to *remoteTS*
2: increment *localTS*
3: obtain *remoteOpPrime[0]* by evaluating *xform(remoteOp, localOp)*
4: **for** all items i in *opBuffer* **do**
5: obtain *remoteOpPrime[i+1]* by evaluating *xform(remoteOpPrime[i], opBuffer[i])*
6: **end for**
7: call *applyOp(remoteOpPrime.last)*
8: obtain *localOpPrime* by evaluating *xform(localOp, remoteOp)*
9: set *localOp* to the value of *localOpPrime*
10: **for** all items i in *opBuffer* **do**
11: obtain *opBuffer[i]* by evaluating *xform(opBuffer[i], remoteOpPrime[i])*
12: **end for**

**Algorithm 4.6** Algorithm for a basic CreatingLocalOpFromBuffer state.

1: increment *localTS*
2: set *localOp* to *opBuffer.first*
3: remove *opBuffer.first* from *opBuffer*

## 4.3 Controller FSM

As described in Chapter 3, the proposed CLOT algorithm is based on principles of real-time systems. That is, a real-time feedback control mechanism is used

TABLE 4.3: State table for Controller Finite State Machine.

| **Present State** \ **Input** | **CtoS_Msg** | **No Input** |
|---|---|---|
| **Listening** | if remoteTS $\geq$ localTS + 1<br>PersistingNew<br>if remoteTS $\leq$ localTS<br>Listening | – |
| **PersistingNew** | – | SendingACKToClient |
| **SendingACKToClient** | – | SendingToRemainingClients |
| **SendingToRemainingClients** | – | Listening |

where operations are transmitted to other users as a feedback signal via a central *Controller* which makes the proper decision on the action to be taken. This architecture considers every modification made to the document content as a new disturbance introduced into the control system, while the document content edited up to the current point is the input of the system.

The *Controller* maintains the ground truth of the collaborative content based on a timestamp value generated by the Finite State Machine at each *Client* site. The *Controller* only accepts operations submitted by *Client* instances against a new content timestamp, thereby establishing a continuous global order of operations. For example, two different operations from two different *Client* instances could be transmitted to the *Controller* at the same time, both containing the same new timestamp. Based on the order of arrival within the message queue of the *Controller*, only the first operation to arrive bearing the new timestamp is accepted by the serial logic of the *Controller*. In this way, the serialization of operations is ensured and an official history of operations is formed. The FSM for the *Controller* was therefore modeled as illustrated in Figure 4.4, with Table 4.3 showing the state table.

The *Controller* remains in the *Listening* state until the arrival of a *CtoS_Msg* containing a timestamp value *TS* and an *Op* object. The decision node after the *Listening* state has guard conditions that compare the received *remoteTS* value from the *CtoS_Msg* and the *localTS* value of the *Controller*. If the received timestamp value is greater than the *Controller*'s known recorded timestamp value, the *PersistingNew* state is reached where the operation and corresponding timestamp are

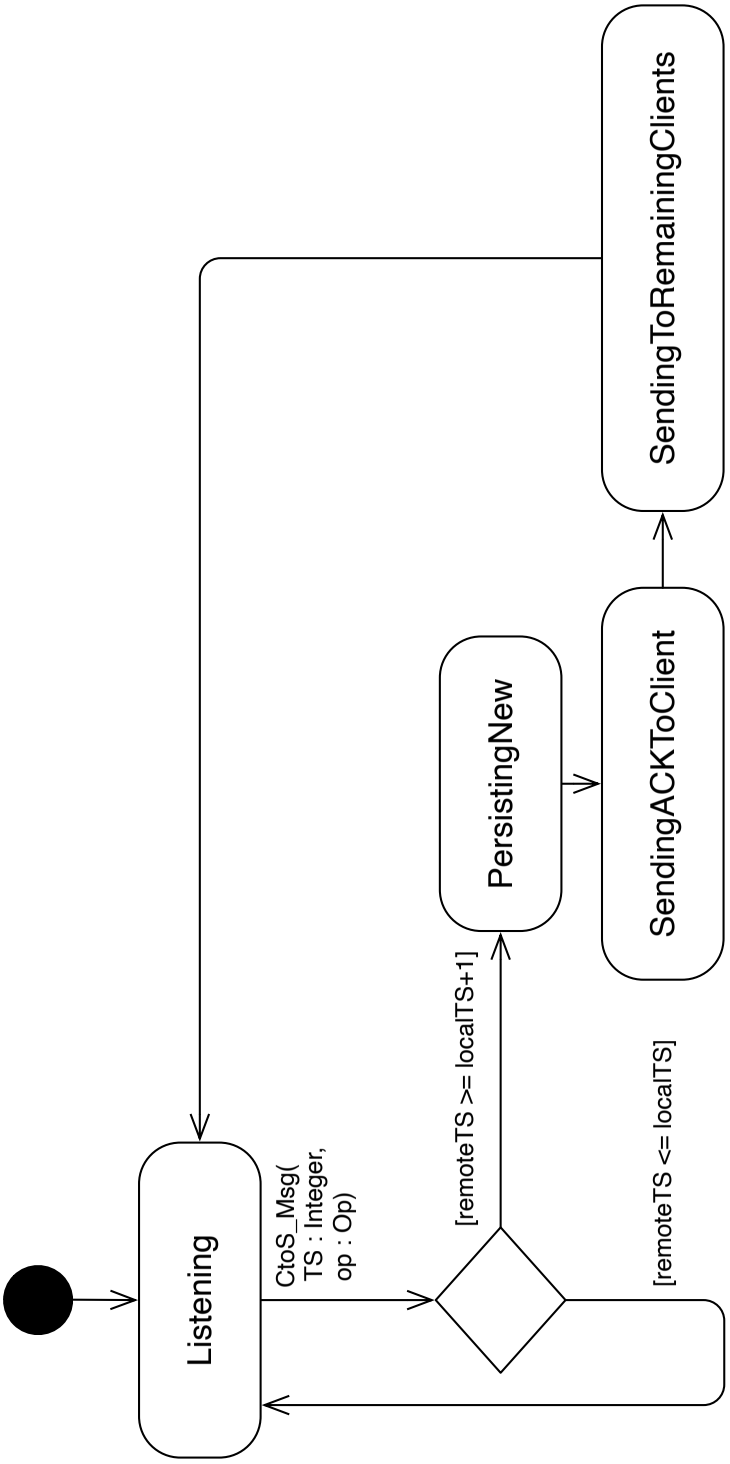



FIGURE 4.4: Controller Finite State Machine.

added to a local array or persisted in a database, as shown in Algorithm 4.7. The *SendingACKToClient* state then emits a *StoC_ACK* event to the *Client* that originated the operation, and the *SendingToRemainingClients* state emits a *StoC_Msg* (containing the newly-accepted *Op* object) to all other *Client* instances, thereby continuing progression through the FSMs of all clients (Algorithm 4.8 and Algorithm 4.9, respectively). Operations arriving with out-of-date timestamps will not be acknowledged, and the corresponding *Client* will re-transmit the transformed version of their unacknowledged operation upon having received the *StoC_Msg* event (instead of the *StoC_ACK* event) from the *Controller*. Note that the *SendingACKToClient* state may be omitted if the *Client* determines the arrival of an acknowledgement by checking if the received operations are their own, in which case one state can be used after *PersistingNew* that simply broadcasts to all *Client* instances.

**Algorithm 4.7** Algorithm for a basic PersistingNew state.

```
1: set localTS to the value from the received CtoS_Msg event
2: persist the Op as part of the official document history
```

**Algorithm 4.8** Algorithm for a basic SendingACKToClient state.

```
1: send StoC_ACK event to the clientID of the accepted Op
```

**Algorithm 4.9** Algorithm for a basic SendingToRemainingClients state.

```
1: for all non-author Client instances do
2:     send StoC_Msg containing the accepted Op and new localTS value
3: end for
```

## 4.4 CLOT Algorithm Analysis

Chapter 2 identified the *consistency model* for OT-based collaborative editing systems as requiring *causality preservation*, *convergence*, and *intention preservation*. As this section will discuss, the FSM-based design of the CLOT algorithm allows all three requirements to be met.

The Lamport timestamp technique defined in Section 2.2.1.1 showed how incrementing a scalar timestamp value before sending a message (which includes the timestamp) and continuing to increment from received timestamps allows achieving a total ordering of events in a distributed system. As a variation on this

logical clock approach adapted to a centralized "serverless" paradigm, the causality preservation requirement is satisfied by the *Controller* logic for maintaining a continuous global order of operations by only rebroadcasting messages with incremented timestamp values that have been accepted into the official record. This is similar to the sequencer approach of the SOCT4 algorithm [7], but instead of using a *Ticket* function, *Client* instances retry to send operations until their timestamp is acknowledged as accepted by the *Controller* (see Section 2.2.1.6). On the *Client*, the timestamp incrementing is performed by Algorithm 4.1 of the *ApplyingLocalOp* state, Algorithm 4.3 of the *ApplyingRemoteOpWithoutACK* state, Algorithm 4.6 of the *CreatingLocalOpFromBuffer* state, and Algorithm 4.5 of the *ApplyingRemoteOpWithBuffer* state, all of which then make use of the *SendingOpToController* state to send the message containing the new timestamp. Similarly, Algorithm 4.2 of the *ApplyingRemoteOp* state, Algorithm 4.3 of the *ApplyingRemoteOpWithoutACK* state, and Algorithm 4.5 of the *ApplyingRemoteOpWithBuffer* state all handle received *StoC_Msg* events by first setting their local timestamp value to that of the received message.

The convergence requirement of Section 2.2.1.2 is also realized through the incrementing timestamps and *Controller* broadcasts of accepted operations, thereby ensuring that all operations are applied to all *Client* instances in order. In this way, the *Controller* coordinates the flow of messages to *Client* instances such that they can achieve convergence at quiescence. By buffering local operations until they are accepted by the *Controller*, the FSM of the *Client* ensures that the two operations taking part in any transformation have originated from the same document state. Chapter 7 will show how a large suite of integration tests, as well as a random events generation component, are executed in a modern simulation tool to verify that all FSMs terminate without deadlocks or infinite loops.

Finally, intention preservation (defined in Section 2.2.1.3 and Section 2.2.1.4) is enabled by having the *Client* FSM enforce a sequence of two operations to be transformed at a time, as determined to be necessary by each *Client* instance in order to resolve local conflicts. Since the presented CLOT integration algorithm has been generalized for messages containing a single plaintext operation, the transformation functions previously shown in Algorithm 2.1 to Algorithm 2.4 of Section 2.2.1.2 remain valid for basic character-based intention preservation. For $n$ concurrent operations, the *xform()* function used by Algorithm 4.3 and Algorithm 4.5 has a complexity of $O(n)$, which is typical of Jupiter-inspired algorithms

(Chapter 8 contains additional performance discussions). The next chapter, Chapter 5, will present more advanced FSM-based transformation algorithms required for tree-based synchronization of DOM nodes, as well as show how the CLOT algorithm can handle multiple operations per message transmitted between the *Controller* and *Client* instances.

# Chapter 5

# DOM-Based FSM Model Design

This chapter proposes a low-level design that enables the synchronization of HTML Document Object Model (DOM) content between multiple users by introducing a set of operations designed around the requirements of OT systems, along with new algorithms based on a novel "map" approach for working with them. The FSM design established in Chapter 4 is enhanced with hierarchical relationships between states in order to support the DOM synchronization functionality required by the Web-Based Collaborative Platform.

## 5.1 Modeling the Required DOM Operations and Algorithms

The architecture in Figure 3.3 of Chapter 3 defined the use of a *Feedback DOM Observer* and a *Comparison Logic* block for identifying operations that represent changes appearing in the HTML DOM as a result of a user's actions within a web-based environment such as a rich-text WYSIWYG editor. The resulting set of operations must be transmitted to the *Controller* block before it can reach the other users of a given collaborative session. Once operations have been generated and transmitted in this fashion, the conflict-handling transformation logic of the *Decision Maker* must prepare the received operations such that they can be applied to each user's DOM by the *SET DOM* block.

At the heart of the implementation of the proposed DOM synchronization functionality lie the operations, which must account for the tree data structure used

to represent the hierarchy of nodes in the HTML DOM. The JavaScript Object Notation (JSON) was determined to be appropriate for encoding operations given the web-based focus of the target real-time collaboration systems. This section outlines the DOM-based algorithms behind each of the aforementioned blocks, other than the *Controller*, which was already described in Section 4.3. The serverless design of the *Controller* ensures that it does not require modification based on the operation content (text vs. DOM) but rather depends only on the timestamps appended to the operations.

### 5.1.1 Defining New DOM Operations

When designing operations for an OT system, it is important to remember the principles established in Chapter 2, such as the need for precision in order to best preserve a user's intentions (Section 2.2.1.4). Guided by these principles and a review of the DOM4 specification [33], as well as observation of the changes enacted upon the DOM by numerous WYSIWYG editors and web applications throughout the course of this research, Table 5.1 shows the proposed list of operations for DOM-based OT systems. The focus on operations that utilize true DOM manipulations, rather than manipulations to intermediate layers that get rendered into a DOM, has resulted with the most comprehensive list of operations explored so far in this domain (see Section 2.3).

Evolving on the definition of an *Op* object from Section 4.1, a DOM-based operation is now defined to have at least a "userid" to identify the creator of the operation, an "opcode" that identifies the type of the operation, an "opcon" with operation-related content (such as positions or values), and a "loc" with DOM-based location information. While not listed in Table 5.1, the proposed OT system also makes use of an *identity* operation, which does not require the "opcon" or "loc" attributes. The *identity* operation was previously introduced in Section 2.2.1.2 and is required for transformation functions such as *deleteData* vs. *deleteData* (Algorithm 5.8 described in Section 5.1.2.4 below).

Section 5.2 provides examples of the JSON structure for each type of operation listed in Table 5.1. The operations have been designed for a JavaScript-based implementation and therefore take advantage of the dynamic nature of JSON attributes to keep messages small. However, this means that operations can vary greatly in their use of attribute data types, where, for example, an *insertNode*

TABLE 5.1: List of operations required for intention-preserving DOM synchronization.

| Operation | Description |
|---|---|
| **insertNode** | Insert a node at a path in the DOM tree. |
| **deleteNode** | Delete an existing node at a path in the DOM tree. |
| **moveNode** | Move a node from a source path to a destination path. |
| **insertData** | Insert a string into a text node. |
| **deleteData** | Delete a string from a text node. |
| **giveData** | Separate an existing text node such that text is "given" to another text node. |
| **receiveData** | Combine multiple text nodes into one "receiving" text node. |
| **insertAttribute** | Insert an attribute and attribute value into an element. |
| **deleteAttribute** | Delete an attribute from a given path in the DOM. |
| **updateAttribute** | Update an attribute value with string-based operations. |

operation has the "opcon" attribute as an integer (the integer represents an index, as will be explained below), while the *insertData* operation has the "opcon" attribute as an array containing a string (the letter to insert) and an integer (the position to insert the letter at). The mapping to an object-oriented structure is therefore not exact, but it can roughly be summarized by Figure 5.1, with data types omitted for attributes requiring such flexibility.

A *FullOp* object is defined to contain multiple *Op* objects, as well as zero or more *Payload* objects (where Payload objects may themselves contain *PayloadAux* objects). When a *FullOp* JSON object is represented as a string and omits the timestamp (TS), the resulting character array can be thought of as having the structure [ +{} , [] ], where +{} denotes "one or more *Op* objects" and the inner [] denotes an array of zero or more payload arrays. Not all operations require the additional payload section, and the payload section is represented as an array in

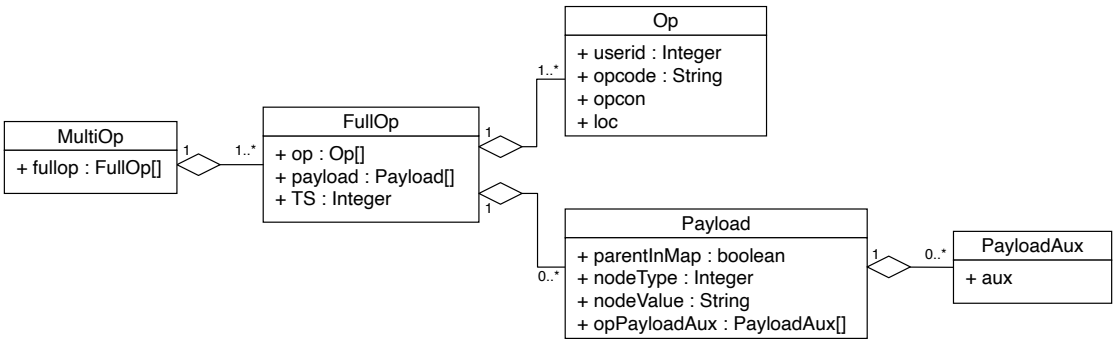



FIGURE 5.1: Relationships between data types required for modeling the proposed set of operations.

JSON for extra flexibility (rather than an object with well-defined attribute names like *Op*). An example *FullOp* containing two operation objects and one payload array can be expanded as shown in Listing 5.1. By composing multiple operations together into a *FullOp* in this fashion, more complex operations can be devised. For example, a "node splitting" operation can be obtained by combining a *giveData* operation and one or more *insertNode* operations, as will be shown in Section 5.2.6 below. Chapter 6 will explore how *FullOp* objects can also be appended together to form a *MultiOp*, which can then be transmitted as messages between components such as *Client* instances, the *Controller*, and an *Events Generator*.

```
1  [
2    {
3      ...      } —— Operation 1
4    },
5    {
6      ...      } —— Operation 2
7    },
8    [
9      [
10       ...    } —— Payload 1
11     ]
12   ]
13 ]
```

LISTING 5.1: Sample FullOp format containing two operations and one payload array.

### 5.1.2 Operation Maps & Algorithms

The DOM4 specification [33] defines *node objects*, including *element nodes* ("node type 1") with properties such as *tagName*, and *text nodes* ("node type 3") with properties such as *data*. The proposed operation-handling algorithms require the DOM to be represented as an array that stores DOM node objects as array indexes. This array will henceforth be called a "map" for its ability to map remote changes to local nodes. As a simple example, a DOM containing just `<p></p>` would produce the map [*p*]. Index 0 of this array is stored as a node object (in this case, an *element node* whose *tagName* attribute will return "p"). However, note that the DOM node objects within the map are not necessarily objects from the final user-visible DOM that is rendered by the browser, but may be in-memory clones of DOM objects (VDOMs).

Three such "maps" are required for the DOM-related algorithms in this thesis:

1. *realMap*, a "real map" (generated from the latest live DOM);
2. *prevMap*, a "previous map"; and
3. *transMap*, a "transitional map".

The nodes within the arrays of the *realMap* and the *prevMap* are ordered such that all parent nodes at the same level always appear before any child nodes. Listing 5.2 shows a DOM that has a *body* node at the first level, a *p* node and a *div* node at the second level, and two text nodes at the third level. The quotations around the text are used to indicate the presence of a single text node within the *p* and *div* elements. The map for this DOM is represented as [*body, p, div, "The quick brown fox", "jumps over the lazy dog"*].

```
1 <body>
2   <p>"The quick brown fox"</p>
3   <div>"jumps over the lazy dog"</div>
4 </body>
```

Listing 5.2: Example DOM with three element nodes and two text nodes.

The remainder of this subsection will describe how the maps are used in the main algorithms corresponding to the components of the architecture in Figure 3.3 of Chapter 3, after which examples with real-world values are provided.

#### 5.1.2.1 Initialization & SET DOM

Algorithm 5.1 is to be executed upon initialization of the system and runs only once. An input DOM node is required which will be used as the root node on which changes are monitored by a "change observer". The *realDOM* variable will therefore always correspond to real nodes from the live DOM that is being rendered by the browser, and great care must be taken when changing these nodes. A *vDOM* variable is initialized on line 2 as an in-memory clone of *realDOM* so that the same nodes are available without connections to the rendered output. This variable will define the "previous" state of the DOM once a change has been detected. The *realMap* and *prevMap* arrays are also initialized before the "change observer" is started.

**Algorithm 5.1** Initialization with node binding.

**Input:** DOM object of root node to bind
1: set *realDOM* to the DOM object of the root node to bind
2: set *vDOM* to be a clone of *realDOM*
3: set *realMap* by generating a map from *realDOM*
4: set *prevMap* to be a copy of *realMap*
5: start change observer

In this thesis, variables within the algorithms that contain the word "DOM" are DOM objects (nodes) as defined in the DOM4 [33] specification. Variables that contain the word "Map" are arrays of DOM objects as described above. It is also important to note the difference between the use of the word "clone" and the word "copy" in the given algorithms. The word "clone" is used to identify a deep clone on a DOM object whereby all child nodes are copied as new unique objects, while the word "copy" is used to denote a copy of the map array consisting of DOM nodes where the nodes themselves are not new objects, but rather are *references*. For example, after the execution of Algorithm 5.1, modifying a node by using the objects of the *realDOM*, *realMap* or *prevMap* variables will affect the rendered user-visible DOM nodes, but modifying the nodes of the *vDOM* variable will not.

The *transMap* is used as an intermediate step and is assembled based on the information contained as part of some *FullOp* objects. When a client receives a *FullOp*, the *transMap* is formed by extracting the payload contained within the *FullOp* and appending it to the end of the latest *realMap*. A simplified version of this *genTransMap()* algorithm is given as Algorithm 5.2. A full implementation of the algorithm would handle additional node types (for example, "type 8" comment

nodes that appear in the DOM as `<!-- ... -->`) and include logic for the *parentInMap* boolean within payload arrays. Previously introduced in the *Payload* class of Figure 5.1, the *parentInMap* boolean is used to indicate if the required parent element node is already present in the DOM, as will be shown in the examples of Section 5.2.

**Algorithm 5.2** Algorithm genTransMap() for obtaining a transitional map.

**Input:** *realMap*, *inputFullOp*
**Output:** *transMap*

```
1: set transMap to realMap
2: if payload section of inputFullOp is empty then
3:     return transMap
4: end if
5: for all payload items p in inputFullOp do
6:     if p.nodeType is an element node then
7:         use p.nodeValue as the tagName to create a new element node and
   append it to transMap
8:     else if p.nodeType is a text node then
9:         use p.nodeValue as the data to create a new text node and append it
   to transMap
10:    end if
11: end for
12: return transMap
```

Algorithm 5.3 shows the pseudocode corresponding to a *SET DOM* block in the architecture of Figure 3.3. That is, the *FullOp* received by the *SET DOM* block is ready to be processed to modify the local DOM of the user, with all necessary transformations already being accounted for by the operations within. In the algorithm, the input *FullOp* is identified as *remoteFullOp* since it typically contains the result of the *Decision Maker* block (described in Section 5.1.2.4 below), but the *SET DOM* block can also be used to apply test operations produced by an *Events Generator* (discussed further in Section 5.3 below).

Algorithm 5.3 makes use of the *genTransMap()* function to assemble the *transMap* on line 4. In addition, an *applyOp()* function is called from line 5 and uses string manipulation functions together with the functions of the DOM4 API [33] to modify the user-visible DOM so that the received changes described in *remoteFullOp* appear in the web browser (Listing 6.3 of Chapter 6 provides an example). The change observer, described further in Section 5.1.2.2 below, is disabled so that the modification made by *applyOp()* is not detected as a new local operation, as this

would result in an infinite loop of the same operation being applied, detected, and re-transmitted around the system.

**Algorithm 5.3** Algorithm to apply a received FullOp.

**Input:** remoteFullOp

1: stop change observer
2: update *prevDOM* to be a clone of *realDOM*
3: update *prevMap* to be a copy of *realMap*
4: obtain *transMap* by evaluating *genTransMap(realMap, remoteFullOp)*
5: **for** all *op* instances in *remoteFullOp* **do**
6: call *applyOp(op, transMap)*
7: **end for**
8: restart change observer

#### 5.1.2.2 Feedback DOM Observer

Algorithm 5.4 describes the sequence of instructions to be performed whenever a local change has been detected in the DOM, thereby implementing the *Feedback DOM Observer* functionality of the architecture of Figure 3.3. Previous research [30] has used older DOM3 APIs such as DOM Mutation Events [101] for this purpose, but the Mutation Events APIs have been deprecated and replaced with the more capable MutationObserver object in DOM4 [33]. However, while an event indicating the presence of a change is useful, the descriptions of the changes themselves returned by these APIs were found to be overly verbose and yet missing key relationships between nodes which are required for generating the ten operations proposed in Section 5.1.1. Instead, as will be described in the next section, a custom *diff()* function was implemented to encode changes, where each change is described using the proposed operations. The *diff()* function is called on line 4 of Algorithm 5.4.

**Algorithm 5.4** Algorithm to execute when a DOM change has been detected.

1: stop change observer
2: update *prevMap* to be a copy of *realMap*
3: update *realMap* by generating the latest map from *realDOM*
4: obtain *localFullOp* by evaluating *diff(vDOM, realMap, prevMap)*
5: update *vDOM* to be a clone of *realDOM*
6: send *localFullOp* to *Controller*
7: restart change observer

#### 5.1.2.3 Comparison Logic

Algorithm 5.5 shows the *diff()* logic at the heart of the *Comparison Logic* block of the architecture, which must produce a *FullOp* for sending to the *Controller* via the remainder of Algorithm 5.4. The *prevMap* contains the older list of nodes, but the nodes themselves are from the real DOM. By comparing *prevMap* with a map created from the latest DOM, differences such as the presence of new nodes can be determined, but a map of the *vDOM* is also required in cases such as matching previous parent nodes (line 4). This is because *vDOM* contains independent deep copies of the nodes, while the nodes in *prevMap* will be references to real nodes that may or may not still exist in the latest state of the DOM. For example, nodes that have just been deleted will no longer be present in the *realMap* but will remain in the *prevMap*, so the *vDOM* must be used to determine which nodes of the *prevMap* no longer have a parent (and are therefore no longer attached to any part of the DOM) in order to assemble an array of *deletedNodes*.

**Algorithm 5.5** Algorithm diff() to assemble a set of operations.

**Input:** *vDOM, realMap, prevMap*

**Output:** *fullOp*

1: set *vMap* by generating a map from *vDOM*
2: obtain *insertedNodes* array by looping over all nodes of *realMap* to see which are not in *prevMap*
3: obtain *transMap* by appending *insertedNodes* to *realMap*
4: obtain *deletedNodes*, *movedNodes*, *changedData* and *changedAttributes* arrays by looping over all nodes of *prevMap* to see which changed in *realMap* and using *vMap* where necessary
5: obtain initial *fullOp* by evaluating *genOp(transMap, deletedNodes, "deleteNode")*
6: repeat updating of *fullOp* by appending result of evaluating *genOp(transMap, movedNodes, "moveNode")*, etc. for all types obtained above
7: obtain *payload* by evaluating *createPayload(prevMap, transMap, newNodes)*
8: append payload to end of *fullOp* array
9: **return** *fullOp*

Algorithm 5.5 operates by first forming new arrays of the determined node differences, and these arrays are sent to a *genOp()* function by line 5. The *genOp()*

function assembles the JSON object to properly describe an operation with all necessary attributes detailed in Section 5.2. A *createPayload()* function assembles the *payload* array containing properties of new nodes in a manner similar to *genTransMap()* of Algorithm 5.2. This array is then appended to the end of the final returned *fullOp* for sending to the *Controller*.

#### 5.1.2.4 Decision Maker

Upon receiving a *FullOp* from the *Controller*, the *Decision Maker* block of a *Client* performs a version of the transformation-related Algorithm 4.3 of Chapter 4 with an important change due to the nature of rich-text editors like TinyMCE and how they operate on the DOM. The problem appears with line 5 of Algorithm 4.3, where the transformed version of the received operation must be applied to the local DOM. WYSIWYG editors such as TinyMCE typically exert a degree of automated control over the DOM content for which they are responsible. For example, multiple text nodes within the same parent node may automatically become combined into one text node, simply because that is how the editor prefers to manage them. While normal use of the editor will not be affected much by this "sanitization" process, content pasted from a different document or forced into the DOM (using the proposed *applyOp()* function) is susceptible to be modified in often-unpredictable ways. Any external changes made to the DOM can therefore be considered as suggestions, since, ultimately, the editor will have the final say in how the DOM is structured once it has "settled". The sanitization approach can vary between WYSIWYG editors, editor versions, and even web browsers.

To work around this sanitization behavior of real-world web applications while retaining a generalized DOM synchronization solution, Algorithm 4.3 is updated to use the *diff()* function presented in Algorithm 5.5 in order to determine the operations to be sent to the *Controller* after the local DOM has settled. That is, *xform()* is again only required to modify the first input parameter (as per Algorithm 4.5) in order to obtain the operation that needs to be applied locally, and once it's been applied and the DOM has been allowed to settle, the *diff()* function is used to determine the updated operation that represents the local change. Algorithm 5.6 shows the updated version of Algorithm 4.3 that accounts for these changes. Variables like *remoteOp* have now been evolved to use the plural *remoteOps* since, as established in Figure 5.1, a *FullOp* object can contain one or

more *Op* objects. To account for such *sets* of operations, two loops are required to ensure all remote operations are transformed against all local operations, as shown on lines 6 to 12. The use of *xformedOps[remoteLoopInt]* as the target of the *xform()* call but also as the destination of the transformation result ensures that a cumulative result is obtained by transforming against all *localOps*. Based on the received *remoteOps*, line 4 computes the *transMap* using *genTransMap()* (Algorithm 5.2). On line 13, the *applyOps()* function then uses the resulting *xformedOps* along with the nodes of *transMap* to modify the *realDOM*. Once the *realMap* is updated based on the latest DOM, line 15 calls the *diff()* function (Algorithm 5.5) to obtain the latest *localOps* for sending to the *Controller*.

**Algorithm 5.6** Algorithm for a DOM-based Decision Maker block.

**Input:** *vDOM, realDOM, prevMap, remoteTS, remoteOps, localOps*
**Output:** *localOps*

1: set *localTS* to *remoteTS*
2: increment *localTS*
3: set *remoteTS* and *remoteOps* to the values within the received *StoC_Msg*
4: obtain *transMap* by evaluating *genTransMap(prevMap, remoteOps)*
5: initialize *xformedOps* as an empty array
6: **for** every index *remoteLoopInt* in *remoteOps* **do**
7: set *remOp* to *remoteOps[remoteLoopInt]*
8: add *remOp* to *xformedOps*
9: **for** every index *localLoopInt* in *localOps* **do**
10: evaluate xform(xformedOps[remoteLoopInt], localOps[localLoopInt]) and set xformedOps[remoteLoopInt] to the result
11: **end for**
12: **end for**
13: call *applyOps(xformedOps, transMap)*
14: update *realMap* by generating the latest map from *realDOM*
15: update *localOps* by evaluating *diff(vDOM, realMap, prevMap)*

The *xform()* function now uses a series of DOM-based Operational Transformation rules, and to begin defining such rules, the fundamental transformation functions previously explored in Section 2.2.1.2 of Chapter 2 are extended to support the proposed DOM-based operations listed in Table 5.1.

Algorithm 2.1 of Section 2.2.1.2 introduced the transformation function Tii for two text-based *insert* operations. Algorithm 5.7 shows how Algorithm 2.1 has been upgraded to handle two DOM-based *insertData* operations and renamed to TII. While the use of conditional and *return* statements in the algorithm can be rearranged to decrease the number of lines, the given version of the algorithm is

more explicit in order to enable a clearer explanation of the underlying OT logic and facilitate the creation of a corresponding FSM-based model.

**Algorithm 5.7** Transformation function for insertData vs. insertData.

```
1:  function TII(insertData[opcon_tar, userid_tar, loc_tar],
                 insertData[opcon_ref, userid_ref, loc_ref])
2:      if loc[0]_tar ≠ loc[0]_ref then
3:          return insertData[opcon_tar, userid_tar, loc_tar];
4:      else
5:          if opcon[1]_tar < opcon[1]_ref or
              (opcon[1]_tar = opcon[1]_ref and userid_tar < userid_ref) then
6:              return insertData[opcon_tar, userid_tar, loc_tar];
7:          else
8:              opcon[1]_tar = opcon[1]_tar + 1;
9:              return insertData[opcon_tar, userid_tar, loc_tar];
10:         end if
11:     end if
12: end function
```

In Algorithm 5.7, unique user identifiers are denoted using $userid_{tar}$ and $userid_{ref}$, where target and reference versions of variables are identified using *tar* and *ref* subscripts, respectively. As will be shown in Section 5.2.4, the location in the DOM graph is retrieved from the "loc" array of *insertData* operations by using *loc[0]*, which line 2 of Algorithm 5.7 uses to verify if the operations take place within the same parent text node. The presence of different parent nodes immediately indicates that the modified text nodes are not conflicting and the unchanged target operation is returned by line 3. Similarly, Section 5.2.4 identifies that *opcon[1]* is used to store the character position for an *insertData* operation. The position is incremented on line 8 for the case that the position of the target operation is after that of the reference operation (or, in the case of an insert-tie, the target operation contains a larger *userid*).

As detailed in Section 5.2.5, the *deleteData* operation also uses *loc[0]* for the parent text node and *opcon[1]* for the character position. Algorithm 2.2, which handles conflicts between two *delete* operations, was therefore rewritten as Algorithm 5.8. Line 6 shows how the *opcon[1]* attribute is decreased for the case where the deletion in the target operation occurred after the reference operation. An identity operation is returned by line 11 if both the target and reference operations performed a deletion at the same position.

**Algorithm 5.8** Transformation function for deleteData vs. deleteData.

```
1:  function TDD(deleteData[opcon_tar, userid_tar, loc_tar],
                 deleteData[opcon_ref, userid_ref, loc_ref])
2:      if loc[0]_tar ≠ loc[0]_ref then
3:          return deleteData[opcon_tar, userid_tar, loc_tar];
4:      else
5:          if opcon[1]_tar < opcon[1]_ref then
6:              return deleteData[opcon_tar, userid_tar, loc_tar];
7:          else if opcon[1]_tar > opcon[1]_ref then
8:              opcon[1]_tar = opcon[1]_tar − 1;
9:              return deleteData[opcon_tar, userid_tar, loc_tar];
10:         else
11:             return identity[userid_tar];
12:         end if
13:     end if
14: end function
```

The transformation of an *insert* operation against a *delete* operation was previously shown in Algorithm 2.3 and has been upgraded to use *insertData* and *deleteData* operations in Algorithm 5.9. In the case that a target insertion occurs after a reference deletion, line 8 shows how the target's position (*opcon[1]*) is decremented before the transformed *insertData* operation containing the updated *opcon* array is returned by line 9.

**Algorithm 5.9** Transformation function for insertData vs. deleteData.

```
1:  function TID(insertData[opcon_tar, userid_tar, loc_tar],
                 deleteData[opcon_ref, userid_ref, loc_ref])
2:      if loc[0]_tar ≠ loc[0]_ref then
3:          return insertData[opcon_tar, userid_tar, loc_tar];
4:      else
5:          if opcon[1]_tar ≤ opcon[1]_ref then
6:              return insertData[opcon_tar, userid_tar, loc_tar];
7:          else
8:              opcon[1]_tar = opcon[1]_tar − 1;
9:              return insertData[opcon_tar, userid_tar, loc_tar];
10:         end if
11:     end if
12: end function
```

In a similar fashion, Algorithm 2.4 was rewritten as Algorithm 5.10 to handle transformations for the *deleteData* vs. *insertData* scenario. Line 8 transforms the target *deleteData* operation by incrementing its *opcon[1]* attribute in the case that the reference *insertData* operation contains a position greater than or equal to that

of the target. This adjustment results with a transformed *deleteData* operation, which is returned by line 9 of this transformation function.

**Algorithm 5.10** Transformation function for deleteData vs. insertData.

```
1:  function TDI(deleteData[opcon_tar, userid_tar, loc_tar],
                 insertData[opcon_ref, userid_ref, loc_ref])
2:      if loc[0]_tar ≠ loc[0]_ref then
3:          return deleteData[opcon_tar, userid_tar, loc_tar];
4:      else
5:          if opcon[1]_tar < opcon[1]_ref then
6:              return deleteData[opcon_tar, userid_tar, loc_tar];
7:          else
8:              opcon[1]_tar = opcon[1]_tar + 1;
9:              return deleteData[opcon_tar, userid_tar, loc_tar];
10:         end if
11:     end if
12: end function
```

In general, the foundational OT patterns established by Algorithm 5.7 to Algorithm 5.10 are extended within the *Decision Maker* to enable DOM-based transformation functions by observing the operation attributes of Section 5.2 and adapting the established logic to the attributes that require adjusting. For example, in the case of a conflicting *insertNode* vs. *insertNode* transformation, the logic follows the patterns of Algorithm 5.7. That is, the *loc[1]* attribute is used to determine if the reference operation's *insertNode* position in the DOM is before or equal to the target node's position, in which case the position of the target operation is increased, resulting with the target node going after the reference node.

## 5.2 DOM-Based Operations and Examples

Each operation of the comprehensive list previously introduced in Table 5.1 will now be presented in greater detail. By designing each operation to have a very specific purpose, the editing intention of each user is preserved as best as possible for every potential change that can be made within the HTML DOM, as will be shown with examples. The examples have been selected to demonstrate the use of each operation in one or two representative scenarios, but all operations have been designed to be flexible in order to cover the wide variety of edits possible within the DOM.

### 5.2.1 insertNode

The *insertNode* operation enables the introduction of a new DOM node at a specific location in the DOM tree. It will now be assumed that all users of a collaborative session begin with the initial DOM typical of a clean WYSIWYG rich-text editor (such as TinyMCE [25]) shown in Listing 5.3. Based on the description of a "map" from Section 5.1.2, the *prevMap* array for Listing 5.3 is determined to be [*body, p, br*], where index 0 of the array is a body element node, index 1 is a paragraph element node, and index 2 is a line break element node.

```
1 <body>
2   <p><br></p>
3 </body>
```

Listing 5.3: Initial DOM with basic paragraph.

Suppose that Alice, a participant of the multi-user collaborative session with a *userid* of 0, inserts a *div* node after the *p* node shown in the initial DOM of Listing 5.3 to obtain the DOM shown in Listing 5.4. This can happen inside of an editor such as TinyMCE when Alice copies content from a different webpage and pastes it after the existing paragraph. Alice's clipboard can contain any arbitrary HTML content and rich-text editors such as TinyMCE will allow almost all such content to be included in the DOM. Synchronization of this content must therefore be supported.

```
1 <body>
2   <p><br></p>
3   <div></div>
4 </body>
```

Listing 5.4: Updated DOM with newly-inserted  node.

When Alice's *Comparison Logic* block is presented with the previous state of the DOM shown in Listing 5.3 and the new state of the DOM shown in Listing 5.4, it needs to produce a *FullOp* for sending to the *Controller* and reaching the other session participants. The *FullOp* must therefore describe this change (the newly-created node) by including all necessary attributes while remaining short for efficient network transmission. Listing 5.5 presents the proposed JSON-based data structure capable of meeting these requirements. The listing shows a *FullOp* containing an *insertNode* operation that inserts a *div* node after the *p* node within

the *body* node of the initial DOM example of Listing 5.3. The *FullOp* contains one operation object (line 2 to line 10) and one payload array (line 12 to line 17), with each attribute name previously introduced in the *Payload* class of Figure 5.1 denoted using italics in Listing 5.3.

```
1  [
2    {
3      "userid": 0,
4      "opcode": "insertNode",
5      "opcon": 3,  } —— Index of the definition for new node in transMap.
6      "loc": [
7        0,         }
8        1          } —— Destination path for new node ([0, 0] is path to <p>).
9      ]
10   },
11   [
12     [
13       true,      } —— Boolean denoting if parent element exists (parentInMap).
14       1,         } —— HTML DOM node type (nodeType).
15       "div",     } —— Supporting data for new node creation (nodeValue).
16       []         } —— Optional path used by nested elements (opPayloadAux).
17     ]
18   ]
19 ]
```

Listing 5.5: Basic insertNode operation example.

The *insertNode* operation of Listing 5.5 has been defined to take an "opcon" attribute to identify the node that is to be used for the insertion. The "opcon" value corresponds to the array index within the assembled *transMap* array that contains the details of the new node. That is, "opcon" identifies the index where the new node information will end up after the payload is appended to the *realMap* in order to form the *transMap* as per Algorithm 5.2 above.

At this point, the other session participants are assumed to still have the *prevMap* of [*body, p, br*] described above (no new or conflicting operations). When the JSON structure of Listing 5.7 is received by these other session participants, the payload item of [*true, 1, "div", []*] is processed to obtain a *transMap* of [*body, p, br, div*]. The "opcon" value of 3 refers to this new last index with the *div* in the *transMap*, and the contents of this index will need to be inserted into the DOM

at the location specified by "loc". The "loc" attribute of Listing 5.7 describes the path to the destination position within the DOM tree where the new node is to reside and in this example contains $[0, 1]$. The first value of this array is a reference to a position in the *transMap* as the starting point, so the new node should be inserted at the position found by first navigating to index 0 in the *transMap* (the *body* element taken as a DOM node) and the new node will be the child at index 1 for this *body* DOM node (the *p* element is the child at index 0 so the new element will end up after it as index 1). Note that while "opcon" simply refers to a linear array index, the "loc" attribute is based on a hierarchical addressing system for reaching children of nodes. This reference-based system allows for smaller and more efficient operations than previous tree-based approaches, as previous non-invasive DOM synchronization techniques have always identified nodes by their full path [67][120][97].

Once the operation has been applied to the real DOM of the other session participants as per the remainder of Algorithm 5.3, their *realMap* variable will contain [*body, p, div, br*]. Section 5.1.2 described how the the nodes in the *realMap* are always ordered based on levels in the DOM hierarchy such that parent nodes appear before their children, so the *br* node is a child of the *p* node, which is at the same level as the *div* node, and therefore the *br* node appears after them in the *realMap*.

Figure 5.1 previously presented the *Payload* object as containing attributes *parentInMap*, *nodeType*, *nodeValue* and *opPayloadAux*. Index 0 of the payload array of Listing 5.5 therefore contains a boolean value indicating whether the parent of this new node was already part of the *prevMap*. In the example of Listing 5.5, this was "true" since the parent *body* node already existed. Index 1 indicates the type of node, such as an element node of type 1 or a text node of type 3 as per the DOM4 specification [33]. Index 2 consists of supporting string data (in this case the tag name; text nodes use this attribute for the text data), and index 3 is used to indicate a path (using the same structure as per "loc" described above) for the case that nested elements such as text nodes are inserted together with the parent element nodes.

As an example of the latter scenario, it will now be assumed that all session participants begin with the DOM contents shown in Listing 5.6. This listing shows the DOM contents of a WYSIWYG rich-text editor with the text "The quick brown

fox" already present. The *prevMap* array for this example is represented as [*body, p, "The quick brown fox"*].

```
1 <body>
2   <p>"The quick brown fox"</p>
3 </body>
```

Listing 5.6: Initial DOM with basic text node.

Previously used as an example in Section 5.1.2, Listing 5.2 shows the DOM contents of Listing 5.6 that have been modified by Alice to introduce a *div* element containing a text node with the text "jumps over the lazy dog". In this case, the *Comparison Logic* block of Alice's *Client* must determine a *FullOp* capable of describing the addition of the *div* element together with its inner text node.

Listing 5.5 is therefore extended to include the new text node, as shown in Listing 5.7. The *FullOp* now contains two payload arrays. When the *FullOp* of Listing 5.7 is received by the other session participants, which started with a *prevMap* of [*body, p, "The quick brown fox"*], processing the first payload item of [*true, 1, "div", []*] with Algorithm 5.2 results with a *transMap* of [*body, p, "The quick brown fox", div*] similar to the previous example.

Processing the second payload item of [*false, 3, "jumps over the lazy dog", [3, 0]*], however, results with the *transMap* of [*body, p, "The quick brown fox", div, "jumps over the lazy dog"*], where the path of $[3, 0]$ indicates that the new text node is associated as the 0th child of the node with index 3 of the *transMap*, namely the *div*. The "false" value for the *parentInMap* index of this second payload array therefore allows multiple connected nodes to be inserted with a single *insertNode* operation.

When applied to the real DOM of the other session participants, their resulting *realMap* will contain [*body, p, div, "The quick brown fox", "jumps over the lazy dog"*] to match the DOM of Alice previously shown in Listing 5.2.

```
1  [
2    {
3      "userid": 0,
4      "opcode": "insertNode",
5      "opcon": 3,          } ── Index of the definition for new node in transMap.
6      "loc": [
7        0,                 }
8        1                  } ── Destination path for new node ([0, 0] is path to <p>).
9      ]
10   },
11   [
12     [
13       true,              }
14       1,                 }
15       "div",             } ── Element node to append and form transMap.
16       []                 }
17     ],
18     [
19       false,                                  }
20       3,                                      }
21       "jumps over the lazy dog",              }
22       [                                       } ── Text node to append to path [3, 0].
23         3,                                    }
24         0                                     }
25       ]                                       }
26     ]
27   ]
28 ]
```

Listing 5.7: Advanced insertNode operation example.

### 5.2.2 deleteNode

The *deleteNode* operation must contain the information required to locate and remove a specific node from another client's DOM. The *insertNode* example of Section 5.2.1 has already established key concepts such as "opcon" and "loc" which will remain valid for *deleteNode*. The *insertNode* section completed with a group of collaborative session participants having the same DOM of Listing 5.2, which corresponds to a *prevMap* of [*body, p, div, "The quick brown fox", "jumps over the lazy dog"*]. Listing 5.8 shows an example of a *deleteNode* operation that

deletes the previously-inserted *div* node. In this case, the target node to delete is identified by "opcon" as index 2 of the *transMap*. In addition, to allow for OT-based position manipulation, "loc" also identifies the node as the child of index 1 in the *transMap* parent of index 0. By removing the *div* node, the child text node is also removed from the DOM. The DOM contents of Listing 5.6 are therefore restored and the *realMap* returns to [*body, p, "The quick brown fox"*].

```
1  [
2    {
3      "userid": 0,
4      "opcode": "deleteNode",
5      "opcon": 2,  } —— Index via transMap of node to delete.
6      "loc": [
7        0,          } —— Path to node to delete via transMap.
8        1           }
9      ]
10   },
11   []
12 ]
```

LISTING 5.8: Sample deleteNode operation.

### 5.2.3 moveNode

The DOM4 specification [33] allows a node to "move", whereby the node remains the same object but is relocated to a different part of the DOM tree, as identified with a new parent location. The *moveParagraph* example of Section 2.2.1.4 previously highlighted the importance of a *move*-style operation for improved preservation of user intentions when compared with solutions that only provide *insert*, *delete* and *update* operations in a DOM-based OT system [67].

As an example of the proposed *moveNode* operation structure, assume that a *div* node containing a text node has been added to the default example of Listing 5.3 above to obtain the DOM of Listing 5.9. The *prevMap* for the DOM shown in Listing 5.9 is [*body, p, div, br, "jumps over the lazy dog"*].

The operation shown in Listing 5.10 moves the text node from the *div* node into the *p* node (after the existing *br*). Since no payload is required for this operation, the

```
<body>
  <p><br></p>
  <div>"jumps over the lazy dog"</div>
</body>
```

Listing 5.9: DOM with text node to be moved.

*transMap* remains [*body, p, div, br, "jumps over the lazy dog"*] like the *prevMap*. The "opcon" attribute identifies position 4 in the *transMap*, namely the text node. The "prev" attributes of $[2, 0]$ then indicate the path to the source node (the second index in the *transMap*, which is the *div* node, and then the child at index 0 within it), and the "loc" attributes then indicate the path to the destination (index 1 of the *transMap* is the first *p* node, and the destination is index 1 within this node since index 0 is already occupied by the *br* node). The *realMap* for the final DOM is still represented as [*body, p, div, br, "jumps over the lazy dog"*], but with the *p* node and the *div* node now having updated *childNodes*, and the text node now having a different *parentNode*.

```
[
  {
    "userid": 0,
    "opcode": "moveNode",
    "opcon": 4,
    "loc": [
      1,
      1
    ],
    "prev": [
      2,
      0
    ]
  },
  []
]
```

"opcon": 4, — Index via *transMap* of node to move.

"loc": [ 1, 1 ], — Destination path of node to move via *transMap*.

"prev": [ 2, 0 ], — Source path of node to move via *transMap*.

Listing 5.10: Sample moveNode operation.

### 5.2.4 insertData

The *insertData* operation describes a text insertion event inside of a text node. The attributes required for text-based operations have been well established in previous OT research and discussed in a DOM context (together with the *deleteData* operation) as part of the transformation functions of Section 5.1.2.4. For example, a document may start with a paragraph that contains a text node with the string "a", as shown in Listing 5.11. The *prevMap* of this DOM is [*body, p, "a"*].

```
<body>
  <p>"a"</p>
</body>
```

Listing 5.11: DOM with single-character text node.

If the letter "b" is added after the letter "a" of Listing 5.9, the operation from Listing 5.12 is generated. For this operation, the "loc" attributes still indicates the index of the target text node within the *transMap*, while "opcon" indicates the characters ("b") and position (1) within the text node at which the characters are to be inserted. As per the position numbering previously presented in Figure 2.4, a destination position of 1 means that the new letter is to be inserted after the existing letter "a".

```
[
  {
    "userid": 0,
    "opcode": "insertData",
    "opcon": [
      "b",        } —— Characters to insert into text node.
      1           } —— Position within text node at which to perform insertion.
    ],
    "loc": 2      } —— Text node via transMap where insertion is to take place.
  },
  []
]
```

Listing 5.12: Sample insertData operation.

### 5.2.5 deleteData

The *deleteData* operation is used to remove characters from a text node and has been considered previously in the development of the transformation functions presented in Section 5.1.2.4. Continuing where the *insertData* example of Section 5.2.4 left off, a text node of "ab" resides inside the *p* node within a *body* node as shown in Listing 5.13. The *prevMap* for this DOM is therefore [*body, p, "ab"*]).

```
<body>
  <p>"ab"</p>
</body>
```

Listing 5.13: DOM with two-character text node.

The operation of Listing 5.14 is generated when the "b" is deleted. The "loc" of the *deleteData* operation indicates the node at index 2 of the *transMap* will be the destination of the deletion, while "opcon" contains an array, where the first value of this array contains the characters to delete, and the second value is the position within the destination text node at which to perform the deletion.

```
[
  {
    "userid": 0,
    "opcode": "deleteData",
    "opcon": [
      "b",       } —— Characters to delete from text node.
      1          } —— Position within text node at which to perform deletion.
    ],
    "loc": 2     } —— Index via transMap to text node to delete.
  },
  []
]
```

Listing 5.14: Sample deleteData operation.

### 5.2.6 giveData

The *giveData* operation allows splitting a text node into two, or even three, separate text nodes while tracking the relationships between the nodes. Other researchers have explored such "node splitting" operations for use with OT-based

conflict resolution in the past [120], but the solution presented here goes beyond existing research by being designed to handle the complex behavior of actual real-world DOM nodes rather than simplified intermediate data types that are later converted into a DOM based on a specific WYSIWYG editor's requirements. To help achieve this, the DOM text node being split is considered to be "giving" text, while the new nodes are "receiving" text.

To demonstrate a proposed *giveData* operation, assume that the DOM begins with the previously-shown contents of Listing 5.13 containing the "ab" text node inside the *p* node, therefore starting with a *prevMap* of [*body, p, "ab"*]. Selecting the "b" character and pressing the "bold" button in a text editor such as TinyMCE causes the "b" character to end up as a new text node inside of a new *strong* node, as per Listing 5.15.

```
1 <body>
2   <p>"a"<strong>"b"</strong></p>
3 </body>
```

Listing 5.15: Separation of "ab" text node after introduction of  node around "b".

The JSON-encoded *FullOp* that describes this change is shown in Listing 5.16 (top half with Op objects) and Listing 5.17 (bottom half with Payload objects). The *transMap* of [*body, p, "ab", strong, "b"*] is obtained by applying Algorithm 5.2 to append the two payload arrays. The *insertNode* operation (line 2 to line 10) refers to the node at index 3 of this *transMap*, thereby introducing a *strong* element node, which is defined in the first array item of the payload (line 24 to line 28). The second payload array item (line 30 to line 38) defines the new text node that resulted as part of the string separation, namely that the "b" ended up inside of a new text node. In this second payload array, "false" appears on line 31 since the parent *strong* was not previously present in the *prevMap*. The node is identified as being of type 3 (a text node), which, based on the provided value of $[3, 0]$, is to be created as a 0th child of the node of index 3 in the *transMap* (the *strong* node).

The *giveData* operation (line 11 to line 22) identifies the index of the giving (source) text node via the *transMap* as the "loc". The first value of "opcon" contains a boolean to indicate if the node was placed somewhere rather than lost, the second value indicates the character position within the source where the split

occurred, and the third value indicates the index in the *transMap* of the new receiving (destination) node that is to contain the text from the source.

Once this *FullOp* is applied to the DOM of Listing 5.13, the final DOM of Listing 5.15 is obtained, which corresponds to a *realMap* of [*body, p, "a", strong, "b"*]. The array is ordered in this way because the "a" text node and the *strong* element node are at the same level within the *p* node, while the "b" text node is at the deepest level and therefore appears last in the *realMap* array.

```
1  [
2    {
3      "userid": 0,
4      "opcode": "insertNode",
5      "opcon": 3,  } —— New node is defined by index 3 of the transMap.
6      "loc": [
7        1,         } —— Destination path for new node (after text node at [1, 0]).
8        1          }
9      ]
10   },
11   {
12     "userid": 0,
13     "opcode": "giveData",
14     "opcon": [
15       [
16         true,    } —— Node was placed somewhere.
17         1,       } —— Character position in node doing the giving.
18         4        } —— Index via transMap of text node doing the receiving.
19       ]
20     ],
21     "loc": 2     } —— Index via transMap of text node doing the giving.
22   },
23   ...
24 ]
```

Listing 5.16: Basic use of giveData operation (Op objects).

```
[
  ...
  [
    [
      true,
      1,
      "strong",
      []
    ],
    [
      false,
      3,
      "b",
      [
        3,
        0
      ]
    ]
  ]
]
```

Element node that will end up at index 3 of *transMap*.

Text node that will end up at index 4 of *transMap*.

Listing 5.17: Basic use of giveData operation (Payload objects).

The complexity of the *giveData* operation increases if two new text nodes are required instead of just one. Consider the initial DOM of Listing 5.18, which is similar to the JSON-based example previously described in Section 2.3.3. In this DOM, "abcd" is now present within the *p* node instead of "ab" of the previous example. Selecting and bolding the "b" character still causes the "b" character to end up as a new text node inside the *strong* node, but also requires a new text node for the "cd" characters which have also been separated (the "cd" text node becomes the last child of the *p* node). Listing 5.19 illustrates the resulting DOM.

```
<body>
  <p>"abcd"</p>
</body>
```

Listing 5.18: DOM with four-character text node.

```
1 <body>
2   <p>"a"<strong>"b"</strong>"cd"</p>
3 </body>
```

Listing 5.19: Separation of "abcd" text node after introduction of  node around "b".

The *FullOp* that encodes this behavior now requires two *insertNode* operations and one *giveData* operation. The beginning section containing the operation objects is shown in Listing 5.20, while the remainder containing the payload is shown in Listing 5.21.

By appending the payload of Listing 5.21, the *prevMap* of [*body, p, "abcd"*] obtained from Listing 5.18 is extended to a *transMap* of [*body, p, "abcd", strong, "cd", "b"*]. In Listing 5.20, the second *insertNode* operation (line 11 to line 19) and the second array within the "opcon" section of the *giveData* operation (line 29 to line 33) show how this new third text node is defined by adding a splitting position using the same pattern as the previous example. The final DOM results with a *realMap* of [*body, p, "a", strong, "cd", "b"*], with the "b" text node again at the deepest level of the DOM hierarchy and therefore last in the array.

```
[
  {
    "userid": 0,
    "opcode": "insertNode",
    "opcon": 3,
    "loc": [
      1,
      1
    ]
  },
  {
    "userid": 0,
    "opcode": "insertNode",
    "opcon": 4,
    "loc": [
      1,
      2
    ]
  },
  {
    "userid": 0,
    "opcode": "giveData",
    "opcon": [
      [
        true,
        1,
        5
      ],
      [
        true,
        2,
        4
      ]
    ],
    "loc": 2,
  },
  ...
```

New node is defined by index 3 of the *transMap*.

Destination path for new node (after text node at [1, 0]).

New node is defined by index 4 of the *transMap*.

Destination path for new node (after  at [1, 1]).

Node was placed somewhere.

First split position in node doing the giving.

Index 5 in *transMap* will receive (inside the ).

Node was placed somewhere.

Second split position in node doing the giving.

Index 4 in *transMap* will receive (after the ).

Index via *transMap* of text node doing the giving.

Listing 5.20: Advanced use of giveData operation (Op objects).

```
1  ...
2  [
3    [
4      true,
5      1,
6      "strong",
7      []
8    ],
9    [
10     true,
11     3,
12     "cd"
13   ],
14   [
15     false,
16     3,
17     "b",
18     [
19       3,
20       0
21     ]
22   ]
23 ]
24 ]
```

— Element node that will end up at index 3 of *transMap*.

— Text node that will end up at index 4 of *transMap*.

— Text node that will end up at index 5 of *transMap*.

LISTING 5.21: Advanced use of giveData operation (Payload objects).

### 5.2.7 receiveData

As the inverse of the *giveData* operation, the *receiveData* operation is used when multiple text nodes are joined together and has not been explored in previous research in the context of text nodes based on the DOM4 specification [33]. For example, Listing 5.22 shows a paragraph node contain the text nodes "abc" and "xyz". which has a corresponding *prevMap* of [*body, p, "abc", "xyz"*].

The contents of the *p* node may have become split in this fashion for a variety of reasons. For example, WYSIWYG editors vary in how they remove a node such as the *strong* node of Section 5.2.6 and may leave the separated text nodes behind. To the user, the rendered text inside the editor will not look any different

from a single joined text node, but for the proposed OT system of this thesis, it is important to handle such scenarios that may appear in a valid real-world DOM.

```
1 <body>
2   <p>"abc""xyz"</p>
3 </body>
```

Listing 5.22: DOM with two text nodes inside a paragraph node.

The operation of Listing 5.23 shows how the *receiveData* operation allows joining the two text nodes into a single text node within the same paragraph node (this merging process is also known as "normalization"). In this case, the *transMap* is equivalent to the *prevMap* since there is no payload to append. The "loc" of the *receiveData* operation indicates the node at index 2 of the *transMap* will be the destination of the merge, while "opcon" contains an array, where the first value of this array is the number of characters to skip before inserting the string in the destination, and the second value is the index within the *transMap* to use as the source text node for the merge. Listing 5.24 shows the resulting normalized text node.

```
[
  {
    "userid": 0,
    "opcode": "deleteNode",
    "opcon": 3,          } —— Index via transMap of node to delete.
    "loc": [
      1,                 } —— Path to node to delete via transMap.
      1                  }
    ]
  },
  {
    "userid": 0,
    "opcode": "receiveData",
    "opcon": [
      [
        3,               } —— Position at which to insert the string to merge.
        3                } —— Index via transMap of text node doing the giving.
      ]
    ],
    "loc": 2             } —— Index via transMap of text node doing the receiving.
  },
  []
]
```

Listing 5.23: Sample receiveData operation.

```
<body>
  <p>"abcxyz"</p>
</body>
```

Listing 5.24: DOM with normalized text node.

As an additional example, consider the *FullOp* required to reverse the bold operation applied in the first example of Section 5.2.6, that is, to return the DOM of Listing 5.15 (containing the *p* node with the "a" text node and a *strong* node with a "b" text node) to that of Listing 5.13 (containing just a paragraph with the merged "ab" text node).

Starting with a *prevMap* of [*body, p, "a", strong, "b"*], which remains unchanged for the *transMap*, Listing 5.25 shows the *FullOp* required to merge the node at index 4 of the *transMap* (the "b", as referenced via the second item in the *receiveData* "opcon" array) into position 1 (as per the first item in the "opcon" array)

of the node at index 2 (the "a", which is addressed via the "loc" attribute). The *deleteNode* operation then removes the *strong* node from the DOM, returning the *realMap* to [*body, p, "ab"*].

```
1  [
2    {
3      "userid": 0,
4      "opcode": "deleteNode",
5      "opcon": 3,  } —— Index via transMap of node to delete.
6      "loc":[
7        1,         } —— Path to node to delete via transMap.
8        1          }
9      ]
10   },
11   {
12     "userid": 0,
13     "opcode": "receiveData",
14     "opcon": [
15       [
16         1,       } —— Position at which to insert the string to merge.
17         4        } —— Index via transMap of text node doing the giving.
18       ]
19     ],
20     "loc": 2     } —— Index via transMap of text node doing the receiving.
21   }
22 ]
```

Listing 5.25: Sample receiveData operation.

### 5.2.8 insertAttribute

The *insertAttribute* operation is used to add attributes to existing nodes. For example, the bullets in rich-text editors can be styled to be an outline of a circle rather than the default black circle. The DOM shown in Listing 5.26 contains a regular bulleted list and has a *prevMap* of [*body, ul, li*].

Changing the bullet style to an outlined circle style by using a WYSIWYG editor's user interface may result with the DOM shown in Listing 5.27. Since no payload is required for this operation, the *transMap* remains [*body, ul, li*]. This introduction

```
<body>
  <ul>
    <li></li>
  </ul>
</body>
```

Listing 5.26: DOM with bulleted list (default style).

of an attribute and its value can be accomplished by using the *insertAttribute* operation shown in Listing 5.28. The "loc" of the *insertAttribute* operation indicates the node at index 1 of the *transMap* will be receiving the new attribute (the *ul* node), while the "opcon" indicates the name of the attribute to insert, the value of the attribute to set, and the position within the list of existing attributes at which to insert the new attribute. Since previous attempts at defining attribute-related operations have not focused on synchronizing arbitrary real-world DOM content, they have not explored the ability to specify the attribute position [118]. The final *realMap* for this example remains [*body, ul, li*], where the new attribute is contained within the *ul* node object.

```
<body>
  <ul style="list-style-type: circle;">
    <li></li>
  </ul>
</body>
```

Listing 5.27: DOM with bulleted list (outlined style).

```
[
  {
    "userid": 0,
    "opcode": "insertAttribute",
    "opcon": [
      "style",                    } —— Name of attribute to insert.
      "list-style-type: circle;", } —— Value of attribute to insert.
      0                           } —— Position within list of attributes (first).
    ],
    "loc": 1                      } —— Node is at index 1 via transMap.
  },
  []
]
```

Listing 5.28: Sample insertAttribute operation.

### 5.2.9 deleteAttribute

The attribute added by using *insertAttribute* in Section 5.2.8 can be removed by using a *deleteAttribute* operation as shown in Listing 5.29. The "loc" of the *deleteAttribute* operation indicates that the node at index 1 of the *transMap* is where the attribute to be deleted can be found. The "opcon" simply indicates the name of the attribute to delete.

```
1 [
2   {
3     "userid": 0,
4     "opcode": "deleteAttribute",
5     "opcon": "style",            } —— Name of attribute to delete.
6     "loc": 1                     } —— Node is at index 1 via transMap.
7   },
8   []
9 ]
```

Listing 5.29: Sample deleteAttribute operation.

### 5.2.10 changeAttribute

The *changeAttribute* operation allows modifying the string in the value of an attribute. For example, Listing 5.30 shows the DOM of a *body* node with no child nodes but with a "class" attribute containing the value "mce-content-body". The *prevMap* is simply [*body*].

```
1 <body class="mce-content-body">
2 </body>
```

Listing 5.30: DOM with attribute on body node.

In TinyMCE, adding the "mce-visualblocks" value to the "class" attribute on the *body* node activates a plugin that causes a dashed outline to appear around block-level elements (like *p* and *div*) within the editor. Listing 5.31 gives the operation to achieve this, that is, to change the value of the "class" attribute from "mce-content-body" to the value of "mce-content-body mce-visualblocks". The "loc" of the *changeAttribute* operation indicates the change will be made to an attribute

of the node at index 0 of the *transMap* (the *body* node since the *transMap* remains the same as the *prevMap*). The "opcon" then identifies the name of the attribute that is to be changed and contains an array with a list of changes to be made. Each change has a boolean value indicating if the change is an insertion (true) or deletion (false), a position where the change is to be made withing the attribute's value, and finally the actual text that is to be inserted in the case of an insertion.

```
1  [
2    {
3      "userid": 0,
4      "opcode": "changedAttribute",
5      "opcon": [
6        "class",                  } —— Name of attribute to update.
7        [
8          [
9            true,               } —— Insertion (set to 'false' for deletion).
10           17,                 } —— Position in attribute value where insert.
11           " mce-visualblocks" } —— Text to insert.
12         ]
13       ]
14     ],
15     "loc": 0                  } —— Node is at index 0 via transMap.
16   },
17   []
18 ]
```

Listing 5.31: Sample changeAttribute operation.

Unlike other approaches which only offer operations for replacing the entire attribute value each time an update is needed (for example, `setAttribute(name, value)` in [118]), the proposed *changeAttribute* operation allows pinpointing character-based insertions and deletions to the attribute value, thereby better preserving each user's editing intentions with well-established OT rules in the case of a conflict (see Section 5.1.2.4).

## 5.3 Hierarchical Client FSMs for DOM-Based Operations

While the architecture in Figure 3.3 assumes an external WYSIWYG editor modifies the DOM in often-unpredictable ways common of real-world editing, the evaluation of the proposed FSM-based OT system will first assume predefined (or randomly generated) operations are introduced by an *Events Generator* as previously shown in Figure 3.2. The *Feedback DOM Observer* and *Comparison Logic* blocks required for a web-based implementation (as will be explored in Chapter 8) can therefore be replaced by a second *SET DOM* block to handle operations introduced in this fashion locally before they are sent to the *Controller*. However, rather than concentrating on generalized high-level blocks, the components within the well-defined FSMs of the CLOT integration algorithm introduced in Chapter 4 will now become the focus of the proposed OT system.

To begin exploring the FSM-based design of an OT system for DOM synchronization based on the CLOT algorithm, a modeling technique based on a hierarchical arrangement of states is employed. Figure 5.2 shows the nested FSM of the *ApplyingLocalOps* state, which was previously introduced as *ApplyingLocalOp* in the higher-level FSM of Figure 4.3 (Section 4.1). The name of the state has now been evolved to use the plural *Ops* to account for the possibility of a *FullOp* object containing more than one *Op* object.

The *ApplyingLocalOps* state is responsible for processing a *FullOp* of DOM changes and applying those changes to the local DOM of the *Client*. The *FullOp* is contained in a *localOps* variable based on the contents of a received event from the *Events Generator*. As per Algorithm 5.3 of Section 5.1 above, the *ApplyingLocalOps* FSM is modeled to first generate new local maps required for DOM manipulations and then loop over every *Op* in the *FullOp*. The upper decision node then verifies the *OpCode* attribute of the *Op* being processed. A state such as *ApplyingInsertData* then uses the functions of the DOM4 API [33] along with string manipulation functions in order to apply the operation to the live (user-visible) DOM data model. Figure 5.2 only shows four of the ten operations identified in Section 5.1, but additional operations can be introduced to the FSM diagram without great difficulty by following the established pattern of states. The loop over the variable $i$ repeats until all operations of the *FullOp* have been applied to the live DOM (the *sizeOp()* function returns the number of operations within

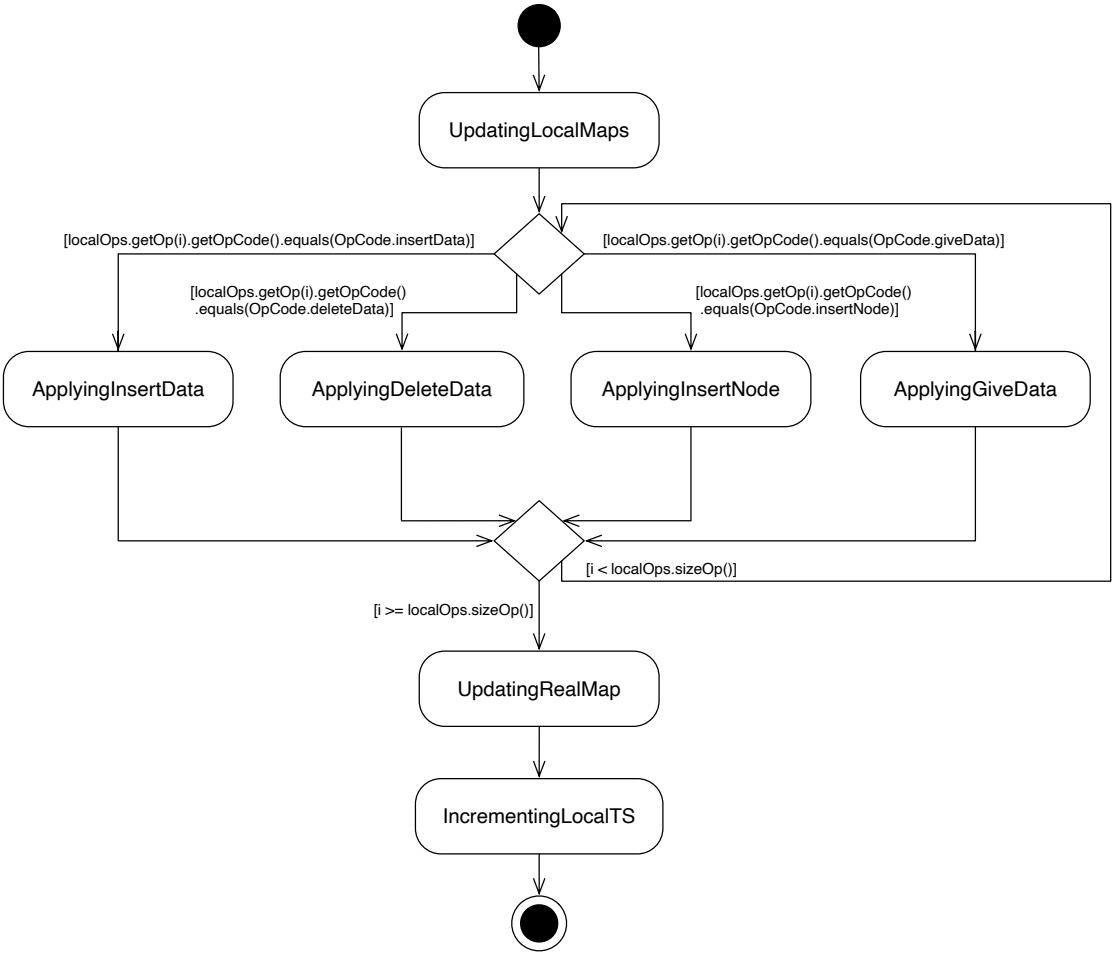



FIGURE 5.2: Example FSM of ApplyingLocalOps state.

the *FullOp*). Based on the updated DOM, the local *realMap* is also updated and the *localTS* value is incremented before the nested state completes execution. The parent *Client* diagram, previously shown in Figure 4.3, then continues by sending an event containing the *localOps* (along with the new timestamp) to the *Controller* via the *SendingOpsToController* state.

Similarly, rather than calling an *xform()* function with the rules of Section 5.1.2.4, the FSM of the conflict-resolving state *ApplyingRemoteOpsWithoutACK* can make use of another layer of FSMs representing each type of possible transformation. The FSM for a target *deleteData* vs. a reference *insertData* transformation is shown in Figure 5.3. As established in Algorithm 5.10, none of the attributes of the target operation need adjusting if both operations do not have the same *loc* value (and are therefore not in the same node), and the FSM structure clearly illustrates the available shortcut and how the entire transformation logic section is omitted in this case. If both operations are in the same node, the position

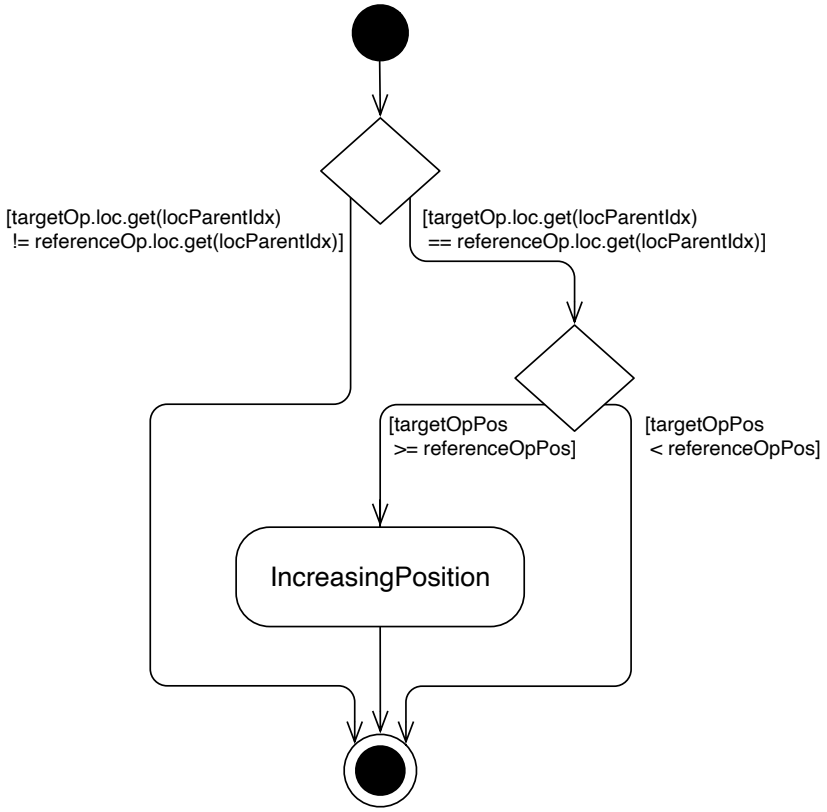



FIGURE 5.3: Example FSM of a substate of ApplyingRemoteOpsWithoutACK for transforming deleteData operations against insertData operations.

attribute of the target operation is incremented only if the target deletion is after the reference insertion (the *opcon* value corresponding to the position to delete is represented as *targetOpPos* in the diagram). Other transformations will follow similar attribute manipulation patterns established above and in previous OT research detailed in Chapter 2.

By building on the key algorithms and FSM models introduced so far in this thesis, Chapter 6 will use an advanced modeling tool to develop the proposed OT system in such a way that it can be observed through simulation.

# Chapter 6

# Advanced FSM Models for OT

This chapter presents how the architecture of Chapter 3 and FSM-based OT algorithms of Chapter 4 and Chapter 5 were evolved and modeled with the use of modern simulation tools and techniques. MATLAB is used to model a distributed system consisting of not only the proposed high-level control loop architecture, but also the low-level FSM-based logic. In this way, the entire real-time system can be simulated and validated for correct OT behavior that allows reaching previously-unexplored DOM-level operations, as will be shown with the test suites and randomized conflict scenario executions of Chapter 7.

## 6.1 Modeling FSMs for Real-Time OT

In order to simulate how the real-time feedback control loop architecture applies OT functionality and ensures that changes among multiple users remain synchronized even for complex hierarchical data types, a robust modeling environment was required that supports nested Finite State Machines and XML-based objects for representing a DOM without the use of a web browser. Extensive testing, debugging and logging functionality are also essential for successful modeling and simulation of the proposed architecture and algorithms, and cross-platform accessibility for development across Windows, MacOS, and Linux was also highly valued.

IBM's Rational Software Architect Designer 9.6 (RSAD) environment was initially selected for this purpose since it allowed modeling State Machines based on the

Unified Modeling Language (UML) standard [42][43][44]. However, the *Model Execution and Simulation* component was found to be unreliable and has since been deprecated in newer versions of the product [130]. RSAD also lacked the ability to incorporate third-party DOM or XML libraries as part of a simulation, and its reliance on the limited Unified Action Language (UAL) standard made the re-implementation of full DOM functionality unfeasible.

Ultimately, the MATLAB environment was selected for the full implementation and simulation of the proposed FSM models. MATLAB is a popular tool for algorithm development in the science and engineering community, and its inclusion of the *Simulink* block diagram and simulation environment enables modeling of the high-level control loop architecture presented in Figure 3.2 of Chapter 3. In addition, MATLAB's *Stateflow* control logic tool allows connecting FSM blocks within a Simulink model, and can therefore be used to model the algorithms of the components identified in the architecture of Figure 3.3. A MATLAB R2020b installation that included the Simulink and Stateflow add-ons was therefore used for the models of this chapter.

In order to produce working FSM models based on the proposed CLOT algorithm detailed in Chapter 4, an iterative approach must be taken where more advanced examples are built on top of a simpler OT foundation. MATLAB modeling will begin with a focus on establishing the control loop by ensuring a basic *identity* operation moves through a two-user system correctly. The system is then extended with the implementation of the buffer mechanism shown in Section 4.2 of Chapter 4. Once the buffered control loop is completed, the *insertData* and *deleteData* operations are added to demonstrate character-based OT functionality, including with buffer support. Finally, the system is extended to include support for several of the novel DOM operations introduced in Chapter 5, as well as a third user.

Finite State Machines based on concepts of distributed systems, string manipulations, and DOM manipulations have not been explored previously in MATLAB to the extent shown in this thesis. Examples are rare and many techniques had to be discovered through experimentation, often requiring workarounds after hitting the limits of what tools like Stateflow are capable of achieving. Implementation details from this chapter may therefore be useful for other researchers looking to apply FSMs to address problems in domains such as distributed databases, natural language processing, knowledge-based systems, and many others.

## 6.2 Basic OT FSM

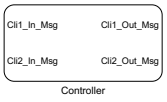
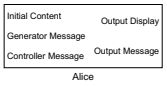
This section will demonstrate how the fundamentals of the CLOT algorithm and the proposed architecture can be modeled and simulated in MATLAB. The basic functionality of the FSMs is established in this section by transmitting a simple *identity* operation between a *Controller* instance and two *Client* instances, one named *Alice* and one named *Bob*.

### 6.2.1 High-Level Control Loop

Figure 3.2 presented a high-level control loop architecture, which was then separated into additional components in Figure 3.3. The high-level architecture of Figure 3.2 was modeled using MATLAB's Simulink environment as shown in Figure 6.1. In this figure, the rounded *Events Generator* and *Controller* blocks contain Stateflow FSM diagrams known as *Charts*, while the rectangular *Alice* and *Bob* blocks are *Simulink Library* instances containing reusable charts of the *Client* FSM logic. The high-level Simulink diagram was named *OT_Control_Loop*, while the reusable *Client Chart Library* was named *OT_Client_Lib*. Table 6.1 summarizes the most important visual elements present in the Simulink diagrams of this thesis.

In Simulink, *Display* blocks are identified with a gray background and give an up-to-date view of dynamic system values as the control loop simulation executes. In Figure 6.1, the display blocks are populated with the values from a completed simulation. For example, the *Initial Document Content* block is a *String Constant* block that can be set to any string value. The content of `OTConst.initialDocEx1Chars` is used to access a string constant named *initialDocEx1Chars* defined in the *OTConst* class representing the desired initial document state (classes are described further in Section 6.2.2). The thin line emerging from the right side of the *Initial Document Content* block carries a string-based MATLAB *Signal*. At the *Signal Branch* point, the signal line heading towards the right reaches a display block, resulting with the value of the string constant corresponding to `OTConst.initialDocEx1Chars` appearing inside the block, namely "abc".

Table 6.1: Summary of common Simulink diagram symbols.

| Symbol | Description |
|---|---|
|  | *Scalar and Nonscalar Simulink Signal* line, which is the most common way of connecting Simulink blocks. |
|  | *Variable-size Simulink Signal* line, such as from the output of a *Simulink MATLAB Function* block. |
|  | *Stateflow Message* line. Messages are used to transmit data between *Stateflow Charts*. |
|  | *Signal Branch* allowing the same signal to reach multiple destinations. |
|  | *Manual Switch* block that determines which of the two input signals on the left side will reach the output signal line on the right side. |
|  | *String Constant* block for generating signals containing strings in the Simulink format (in this example, the single-character string "R"). |
|  | *Display* block for visualizing signal values such as Simulink strings (in this example, the string "abc"). |
|  | *ASCII-to-String* or *String-to-ASCII* block for converting a Simulink string to/from a `uint8` array of ASCII characters. |
|  | *Simulink MATLAB Function* block containing custom MATLAB code. |
|  | *Simulink Input Port* or *Simulink Output Port* for linking to signals at a different level of the system (in this example, port number 1 is shown). |
|  | *Stateflow Chart* containing an FSM diagram modeled using Stateflow. Input ports are listed on the left side of the rounded rectangle, while output ports are listed on the right. |
|  | *Simulink Library* block containing reusable Simulink components such as *Stateflow Charts*. Input ports are listed on the left side of the rectangle, while output ports are listed on the right. |

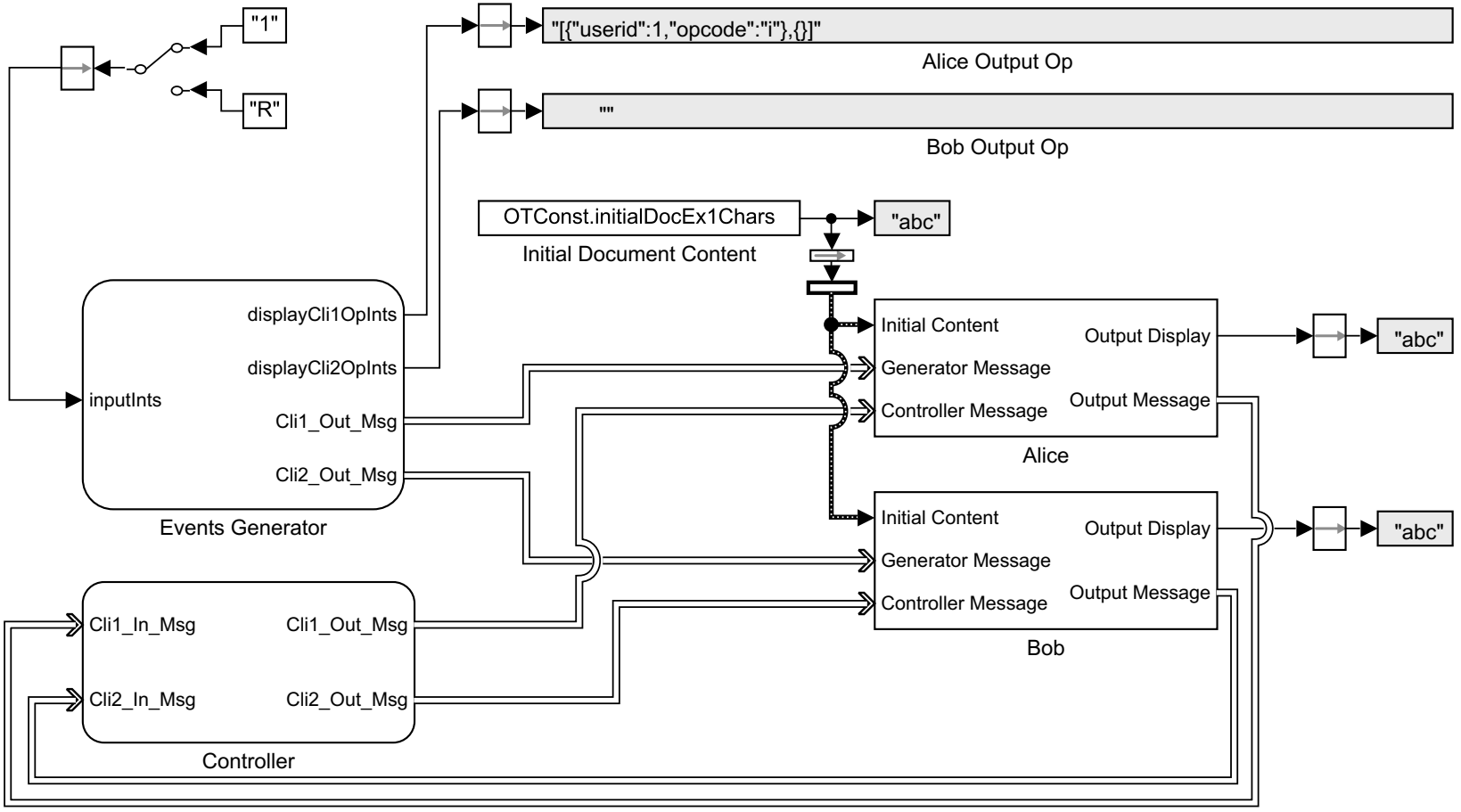



FIGURE 6.1: Two-user Simulink high-level control loop.

While such display blocks are useful for monitoring real-time values throughout the OT system directly from the Simulink diagram, the *Initial Document Content* must also reach all *Client* instances for use in their initialization. The signal line is therefore branched downward by a line that leads to a *String-to-ASCII* block, which converts a Simulink string signal into a `uint8` array of integers representing ASCII characters. The smaller white blocks containing gray right-pointing arrows are Simulink *ASCII-to-String* and *String-to-ASCII* blocks. Next reached by the downward-pointing signal line is a code-based *Simulink MATLAB Function* that appears in Figure 6.1 as an empty thicker-outlined block. Its purpose is to convert the received `uint8` array into a `uint32` array ideal for consumption via the *Initial Content* port of the *Client Chart Library* block of Alice and Bob. This variable-sized output signal of a *Simulink MATLAB Function* is denoted using a thick line containing white dots.

A *Manual Switch* block connected to the *inputInts* input port of the *Events Generator* block allows selecting between a fixed test number from the test suite (upper position with an integer in a string constant block) or a randomly-generated test (lower position with the string constant "R"). In Figure 6.1, the switch is shown to be selecting test "1", which corresponds to a test from a list of OT tests defined in a class named *OTGenerator*. The resulting operations to be sent to Alice and Bob, as determined by the *Events Generator*, are displayed in the corresponding *Output Op* display blocks shown in the top-right corner of the figure. The operations of test "1" are shown to consist of the *Client*'s unique identifier (in this case Alice's identifier of 1) and a simple "i" as the opcode. In this test, Alice is to generate a local *identity* operation, so Alice will receive the operation `[{"userid":1,"opcode":"i"},{}]` as a Stateflow *Message* from the *Events Generator* in order to begin the simulation. The message uses the *FullOp* JSON encoding described in Section 5.1.1, omitting the payload section for the simple identity operation example of this section. Messages are transmitted along paths denoted using *double lines*, such as from the output port named *Cli1_Out_Msg* of the *Events Generator* to the input port named *Generator Message* on Alice's *Client Chart Library* block. As will be shown, messages are sent from a Stateflow chart using the *send()* function and accessed from a *.data* property on a receiving chart's message port.

Once the operations are transmitted as separate messages to each *Client* involved, the latest document state of each *Client* is updated in the display blocks on the

far right side. In test "1", Bob must receive and apply the local *identity* operation performed by Alice. Since the *identity* operations of test "1" do not modify the initial string, both *Client* instances continue to output "abc" after the test has executed to completion. Section 2.2.1.2 defined "quiescence" as the point at which all operations have been executed at all sites and there are no messages remaining to be processed. In this system, quiescence is determined based on a predefined maximum number of "ticks" known to be valid across all tests, as will be explained in Section 6.2.3. At quiescence, an additional script verifies that both outputs are consistent, and that both outputs match the value required by the test in order to assert if the test is a "pass" or a "fail". The result is logged to the console and file system before the next available test is executed.

### 6.2.2 Classes for Modeling CLOT

Table 6.2 summarizes the main classes of this basic version of an OT system based on the CLOT algorithm of Chapter 4. Some of the classes, such as *FullOp*, were previously introduced in Section 5.1.1. All class names from the "Class" column were implemented as `.m` files containing static methods that can be called from the Stateflow Action Language.

It is important to note that the Stateflow Action Language that appears on Stateflow diagrams is a restricted version of the MATLAB Language used at the MATLAB console or in `.m` files (classes, functions etc.). This is partially due to the C-based compilation that occurs in the background whenever Simulink content is executed. While the MATLAB Language supports a weakly-typed programming model with inferred data types, the Stateflow Action Language generally requires variables to be strongly defined in a list created by using the *Model Explorer* interface of Simulink. The MATLAB Language has a useful string datatype, but Stateflow is limited to integer arrays, and must therefore represent strings as ASCII-encoded characters (*Simulink Strings* mentioned as moving across signal lines in Section 6.2.1 are also not permitted). MATLAB's powerful string manipulation libraries, as well as JSON and XML encoding/decoding libraries, are therefore accessed from *static methods* within `.m` files (or functions within `.m` files that aren't classes) called from the Stateflow Action Language of the Stateflow FSM diagrams. Stateflow Action Language cannot instantiate MATLAB classes, but merely call static methods within them. Stateflow diagrams therefore contain

TABLE 6.2: List of classes required for a basic OT FSM simulation.

| Class | Description |
|---|---|
| **ClientNameEnum** | This enumeration class is used to associate display-friendly *Client* names like "Alice" to IDs like "1" (or vice versa). |
| **FullOp** | This class creates *FullOp* objects. The basic (payload-less) *FullOp* format is defined as having the structure [ +{} , { } ], where +{} denotes "one or more Op objects", and is concluded with an "extras" structure { } for the timestamp (TS). |
| **MultiOp** | This class contains static methods to manipulate concatenated *FullOp* objects as characters. |
| **Op** | This class creates *Op* objects with their required properties like "userid" and "opcode". |
| **OpCodeEnum** | This enumeration class contains a list of permitted operation types. |
| **OTConst** | This class defines numerous useful constants for reuse across multiple files. |
| **OTGenerator** | This class enables the creation of fixed and random operations, as required by the *Events Generator*. |
| **OTUtil** | This class contains assorted useful static utility methods for string manipulation, variable casting etc. grouped into one reusable location. |

significant amounts of code for casting or wrapping variables, such as by using `coder.extrinsic()` or `uint8()` before assigning a value to Stateflow-level data, or allocating memory for variables by using `zeros()`. In addition, an array `arr` can be trimmed by using the MATLAB syntax of `arr(1:x)`, where `x` may be a constant such as `MAX_UINT8` or `MAX_UINT32` containing the respective integer limits. In this way, the desired FSM behavior can be modeled and debugged using Stateflow while, at the same time, giving access to MATLAB's advanced text processing functionality, producing useful visualizations at the Simulink layer (such as up-to-date display blocks), while also producing a human-readable console output as the simulation executes.

Figure 6.2 shows a class diagram that builds on the relationships between *MultiOp*, *FullOp* and *Op* previously introduced in Figure 5.1. These three classes are mostly concerned with JSON-based manipulations required by the Stateflow models. For example, the `toChar()` method of the *Op* class is used to obtain the JSON-based encoding of an *Op* object, while the `getOpClientId()` method of *FullOp* uses JSON parsing to extract a *Client ID* value for a given array of characters representing a *FullOp*. For this simplified version of the system, the *OpCodeEnum* class is shown to contain only the *identity* operation "i", while *ClientNameEnum* exposes just two users. The class *OTUtil* contains the numerous static methods for wrapping or converting between the previously-mentioned ways of representing a string in MATLAB, where "MXA" is used to denote a *mxArray* representation, which is a datatype returned from calls to certain MATLAB libraries [131]. The large number of constants in *OTConst* have been omitted, with only `initialDocEx1Chars` shown as an example here since the constant previously appeared in Figure 6.1.

### 6.2.3 Events Generator FSMs

The high-level control loop diagram of Figure 6.1 shows how a *Manual Switch* block decides the input to the *Events Generator* chart's *inputInts* port. An input of "R" means that the *Events Generator* is to continuously generate new random tests during system execution, while a numeric input implies the *Events Generator* is to execute a specific test number from a predefined suite of OT scenarios. The Stateflow implementation of the *Events Generator* must therefore transmit the requested generated or predefined operations to each user specified within the operations, where the operations are obtained from calls to the static methods of the *OTGenerator* class.

Figure 6.3 shows the Stateflow model of the *Events Generator*. Table 6.3 lists the common symbols found on Stateflow charts throughout this thesis. Stateflow transition lines use the syntax "`message []{}`", where not all sections may be present. If the `message` section is included, the transition continues evaluation if the queue corresponding to the given message name is non-empty. Guard conditions for the transition to be taken are denoted between `[]`. If the transition is taken, subsequent actions to be performed are denoted between `{}`. Actions that require lengthy code can use an ellipsis ("`...`") to indicate that a line of code

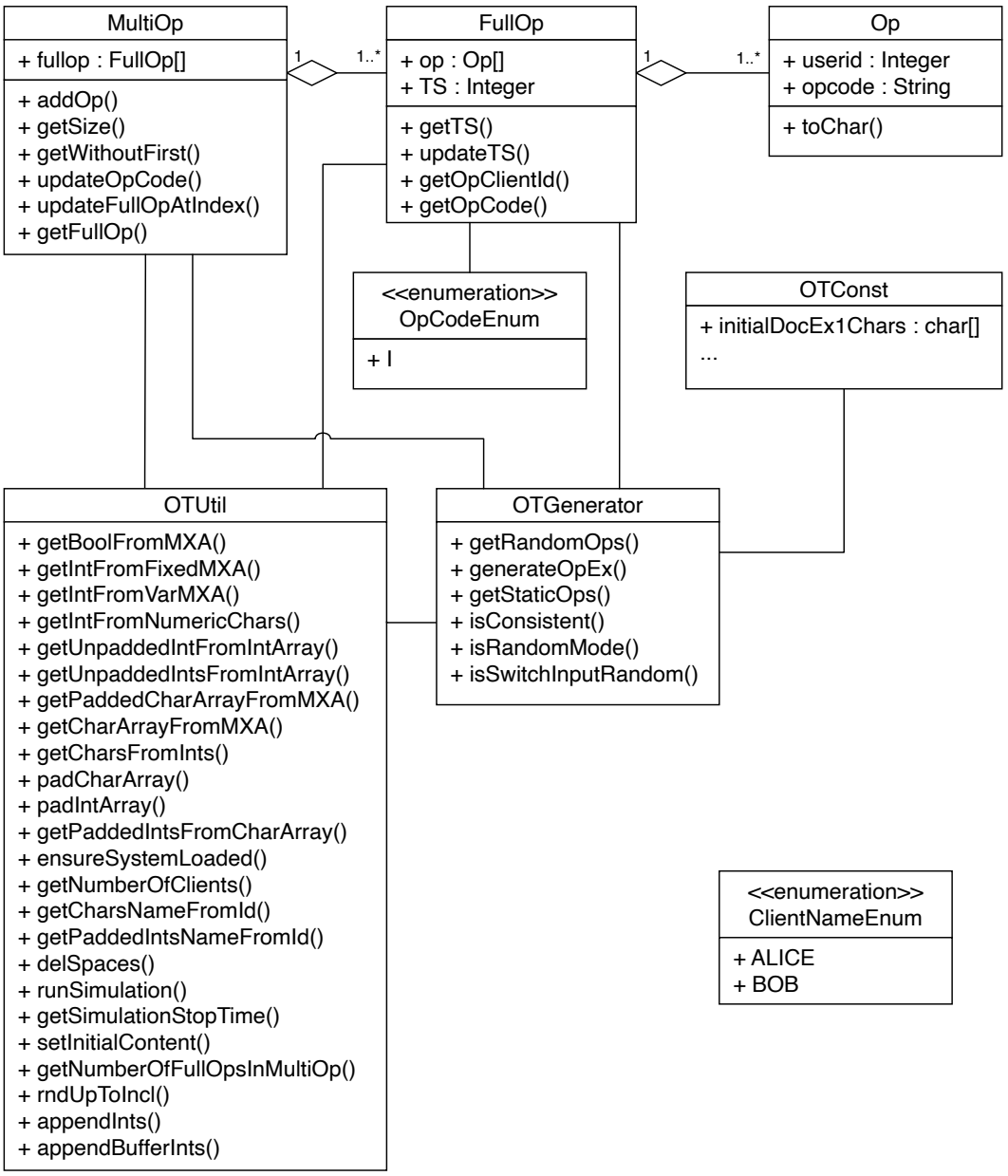
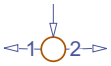
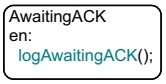



FIGURE 6.2: Classes required for a basic OT FSM simulation.

continues on the following line, or can reference code in other locations such as *Graphical Functions* or functions in `.m` files. Commented lines begin with a "`%`" symbol.

In the top left of Figure 6.3, it can be seen how the position of the *Manual Switch* is determined by using the boolean static method `OTGenerator.isSwitchInputRandom()` on the default transition from the *Entry Point*. If the random position is selected on the switch, the static method `OTGenerator.getRandomOps()` is called to retrieve a *MultiOp* containing randomly generated operations designed

TABLE 6.3: Summary of common Stateflow diagram symbols.

| Symbol | Description |
|---|---|
|  | *State Transition Line*, which may be straight or curved. |
|  | *Entry Point* with default transition into the chart. |
|  | *Junction* where, based on the shown numbering, the left transition is evaluated before the right transition. *Junctions* can also be used to terminate *Graphical Functions*. |
|  | A *State* named *AwaitingAck* where the *entry* action type is denoted using `en:` and is shown to call the `logAwaitingACK()` *Graphical Function*. |
|  | A *Parent State* named *SendingACKToClient* containing a nested FSM. A faint preview of the inner diagram is optionally shown beneath the name of the state. |
|  | *Graphical Function* named *logAwaitingACK* containing a Stateflow diagram representing a function. |
|  | *Simulink Function* named *stop* containing a Simulink diagram representing a function. |

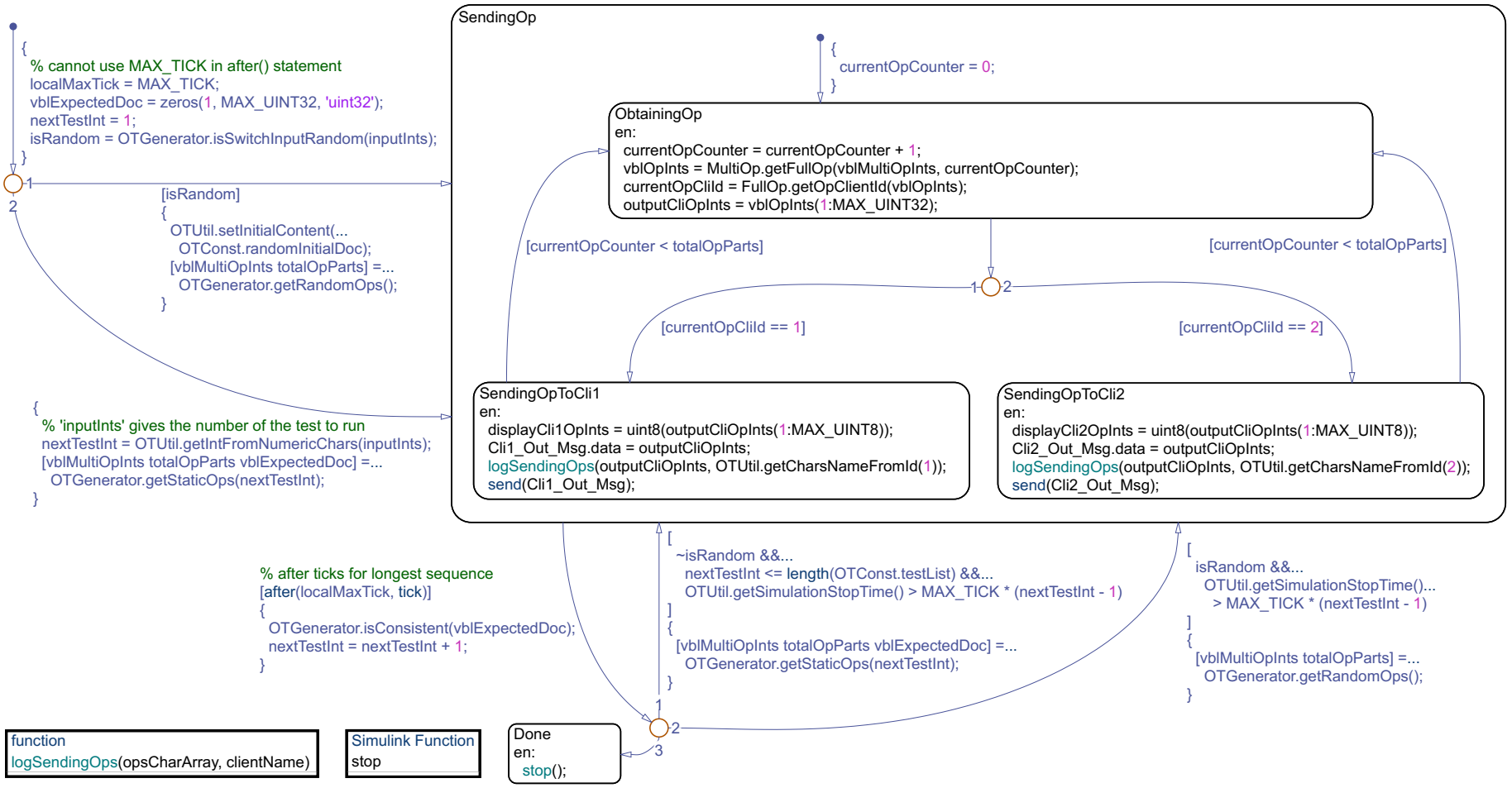



FIGURE 6.3: Two-user Events Generator state of Controller.

to exercise the system. Since the method returns two values, the syntax `[vblMultiOpInts totalOpParts]` is used, where the prefix `vbl` is used to indicate a *Variable Size* integer array to enable passing string data from class-based methods into Stateflow. The *Variable Size* property is configured by using *Model Explorer* for such Simulink data.

If a test number is specified by the *inputInts* contents based on the position of the manual switch, the `OTGenerator.getStaticOps()` static method is used to retrieve the *MultiOp* corresponding to the requested test number, which is assembled based on string-based constants defined in *OTConst*. The large *SendingOp* state of Figure 6.3 then loops through the *FullOp* instances contained within the obtained *MultiOp* and ensures that the correct *Client* receives each corresponding *FullOp*. This FSM-based loop increments a `currentOpCounter` and uses the `totalOpParts` integer value to determine if there are additional *FullOp* items left to process. For each *FullOp*, the `FullOp.getOpClientID()` static method is used to obtain the *Client* number, and either state *SendingOpToCli1* or *SendingOpToCli2* is reached to carry out the message transmission by using the `send()` function of MATLAB for sending messages between Stateflow charts. In the case of test "1", the *Cli1_Out_Msg* port of the *Events Generator* is used in order to reach the *Generator Message* port on the *Client Chart Library* instance representing Alice, as was shown in the Simulink diagram of Figure 6.1. The state *SendingOpToCli1* also shows how the `displayCli1OpInts` variable is set in order for the *FullOp* containing `[{"userid":1,"opcode":"i"},{}]` to reach the Simulink display block via the output of the same name of the *Events Generator* chart. In this example, the *MultiOp* contains a single *FullOp*, but other tests may include several instances in order to address multiple users and create conflict scenarios resolved by OT (Figure 6.33 at the end of this chapter shows such a scenario).

The state *SendingOpToCli1* also contains a call to `logSendingOps()`. The *function* box in the bottom-left of Figure 6.3 is the corresponding *Graphical Function* whose purpose is to generate useful console output during system execution. The contents of this function are shown in Figure 6.4. As the example shows, graphical functions of this thesis commonly apply MATLAB functions such as `char()` (cast integers into ASCII-based characters) and `deblank()` (remove trailing whitespace) in order to obtain useful output strings for display via the console using `disp()`. The execution logs resulting from the use of such functions are important for determining system correctness and will be analyzed in Chapter 7.

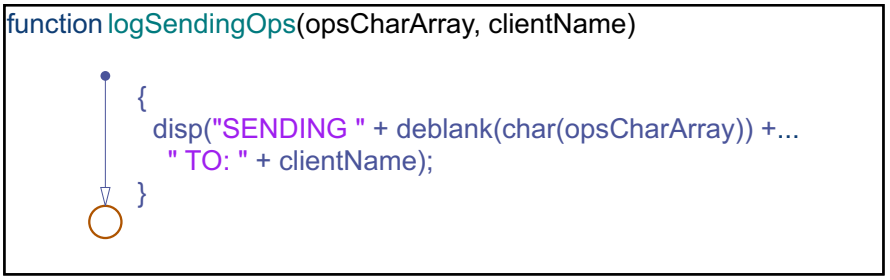



FIGURE 6.4: Contents of `logSendingOps()` graphical function.

The lower part of Figure 6.3 contains a *Junction* that determines if the next transition generates a new test (the `isRandom` path), incrementally selects the next test (∼`isRandom`, where "∼" denotes "not" in MATLAB), or ends system execution when there are no additional tests to execute (system termination is accomplished by using a *Stop Block* within the *Simulink Function* named `stop`). All Simulink models of this thesis are configured to use a "Fixed-step discrete" solver, so Simulink maintains a "tick" counter that increments by a value of 1 as a unit of time during system execution. Since none of the tests exceed the number of ticks in the Simulink workspace-level constant `MAX_TICK`, reaching this value (or a multiple thereof) is used as an indicator for quiescence in a distributed system. The constant `MAX_TICK` is reassigned to `localMaxTick` and used with Stateflow's `after()` state action type to determine when system consistency is evaluated. The static method `OTGenerator.isConsistent()` is called to perform the assertion and prevents further system execution if the resulting document content of all *Client* instances is not equivalent or does not match the final content specified by the test. If the test passes, the operations of the next test are again retrieved with calls to *OTGenerator*.

### 6.2.4 Client FSMs

The *Alice* and *Bob* blocks that were previously introduced in Figure 6.1 make use of the same *Chart Library* containing a high-level *Client State Machine* chart. The Simulink diagram of Figure 6.5 shows the high-level library inputs and outputs. Inputs consist of an *Initial Content* port for the *Initial Document Content*, a *Generator Message* port for messages from the *Events Generator*, and a *Controller Message* port for messages from the *Controller*. The input ports are mapped to variables `inputVblDocInts`, `Local_Change` and `StoC_Msg`, respectively. Output

ports include an *Output Message* port for `CtoS_Msg` messages going to the *Controller*, as well as two ports containing the document state in two different formats: *displayDocInts* is used by the Simulink display block via the *Output Display* port, while *vblDocInts* is accessed programmatically by `OTGenerator.isConsistent()` for exact comparisons at quiescence.

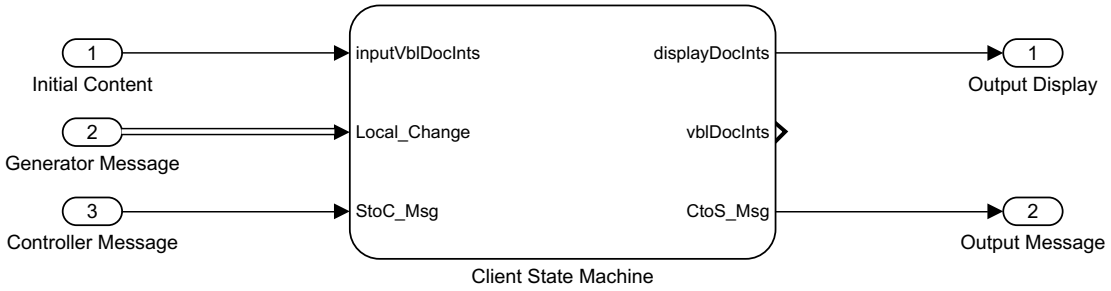



FIGURE 6.5: Simulink Client Chart Library defining Client State Machine input and output ports.

Figure 6.6 shows how the *Client State Machine* block of Figure 6.5 was modeled as a Stateflow-based FSM. The *Synced* and *AwaitingACK* states first identified in the two-state approach of Section 3.5 remain present, while the *ApplyingLocalOps*, *ApplyingRemoteOps*, *SendingOpsToController* and *ApplyingRemoteOpsWithoutACK* states introduced in Chapter 4 are also visible in the model. In addition, the default transition within the *Running* state leads to an *Initializing* state. The outer *Running* state of Figure 6.6 contains a self-transition to allow *Client* instances to "reset" when transitioning between multiple tests making up a predefined test suite. This is accomplished with the `MAX_TICK` approach mentioned in Section 6.2.3. This way, the *Initializing* state is revisited for each test in order to reset several boolean variables.

Figure 6.7 shows the actions performed by the `defaultTransitionAction()` graphical function, which is executed when exiting the *Initializing* state (denoted using the `ex:` state action type). The comment that appears as the first line of the action section of Figure 6.7 mentions the use of a "Mask", which is a MATLAB feature that allows passing parameters from the parent Simulink diagram (Figure 6.1) to the *Simulink Library* instance. In this system, it is used for configuring a unique `Client ID` which reaches the Stateflow layer as the variable *clientId*. The `inputVblDocInts` variable, which was established to contain the *Initial Document Content* in Figure 6.5, is trimmed and cast for use within the *Client* as `vblDocInts`. In addition, by updating `displayDocInts`, which was previously identified as an output in Figure 6.5, the corresponding display block on the right

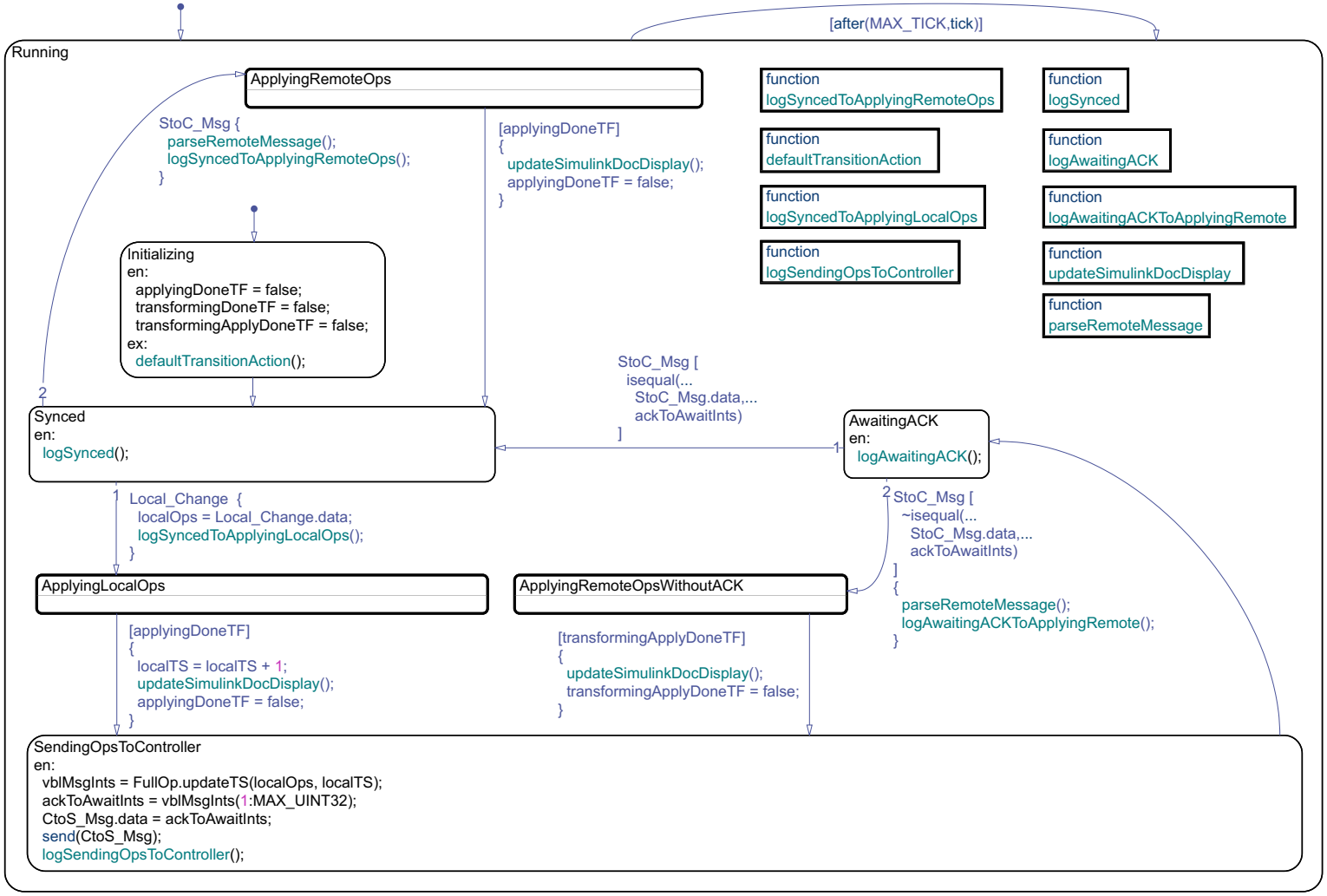



FIGURE 6.6: Basic Client State Machine based on Synced and AwaitingACK states.

side of the Simulink diagram of Figure 6.1 is updated at runtime to reflect the *Initial Document Content* received by the *Client*. Besides containing reusable code, graphical functions also help manage the complexity and cleanliness of the large *Client* FSM. Numerous logging-related graphical functions can be seen in the top-right corner of Figure 6.6, all of which follow a similar `disp()`-based pattern previously shown in Figure 6.4 in order to create useful console output as the system executes. Chapter 7 will discuss the resulting console output in greater detail.

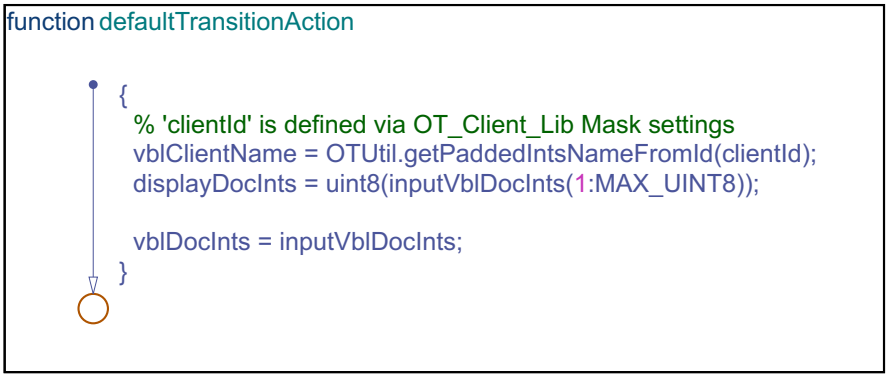



FIGURE 6.7: Contents of basic `defaultTransitionAction()` graphical function.

Once initialized, the *Client* reaches the default *Synced* state, where it remains until it receives a message from either the *Events Generator* or the *Controller* via the corresponding port of the *Client Chart Library* previously shown in Figure 6.5. A `Local_Change` message from the *Events Generator* is used to simulate a *Client* having performed a local operation (for example, an *identity* operation; more useful operations such as typing a letter will be explored in later sections). Having received such a message, the `localOps` variable of the *Client* is set to the message contents by accessing the message's `.data` property. The FSM then transitions to the *ApplyingLocalOps* state shown in Figure 6.8. Therein, the `opsToApply` variable is set to the `localOps` value before the reusable state called *ApplyingOps* is executed. *ApplyingOps* is shown in Figure 6.9.

The *ApplyingOps* state contains an inner `UpdatingLocalVariables` state that first allocates memory using the MATLAB `zeros()` function, and then extracts the opcode from the operation using `FullOp.getOpCode()`. Since *identity* operations do nothing when applied to the *Client*'s document, the basic *ApplyingOps* version shown here simply checks that the opcode within the operation is "i" (based on `OpCodeEnum`) and exits. The boolean variable `applyingDoneTF` is set to

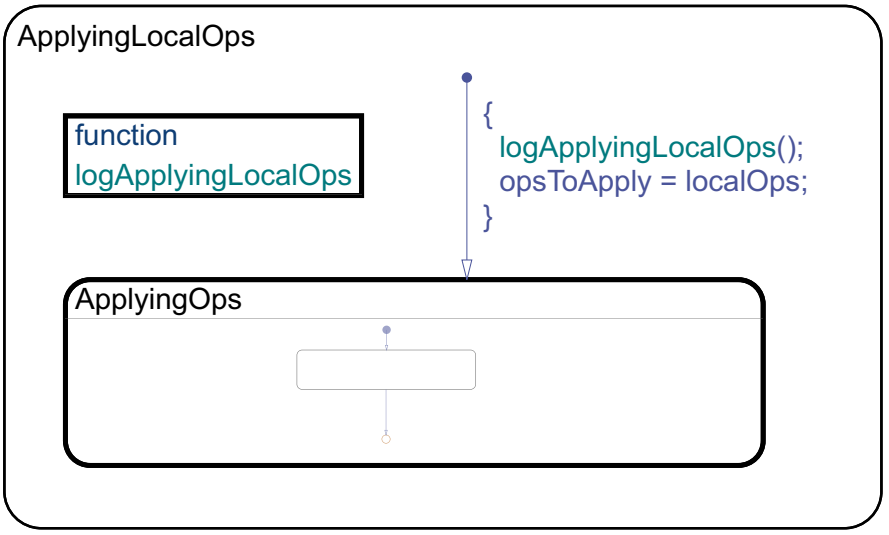



FIGURE 6.8: Basic Client ApplyingLocalOps state.

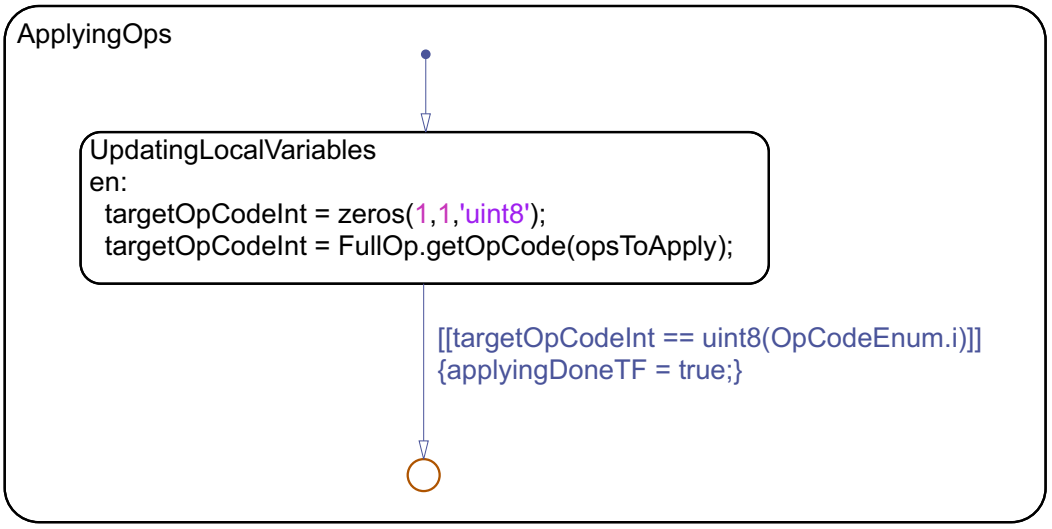



FIGURE 6.9: Basic Client ApplyingOps state.

"true" in order to allow the transition into the *SendingOpsToController* state to be taken within the *Running* state of Figure 6.6 (variables ending in `DoneTF` are used throughout the Stateflow design for this purpose). The timestamp value `localTS` is incremented as part of the transition action, and the `updateSimulinkDocDisplay` graphical function again updates `displayDocInts` for visual observation from the high-level Simulink diagram using the same technique previously shown in Figure 6.7. The *SendingOpsToController* state is then reached, which begins by updating the timestamp within the operation by using `FullOp.updateTS()`. In the next line, the integer array representing the resulting operation is set to `ackToAwaitInts` for later use when checking if the operation has been acknowledged by the *Controller*. The operation array is also set to `CtoS_Msg` and passed to the

`send()` function of Simulink, thereby making use of the *Client*'s *Output Message* port seen in Figure 6.5 in order to reach the *Cli1_In_Msg* port of the *Controller* seen in Figure 6.1. The *Client* then enters the *AwaitingACK* state via the transition on the far-right side of Figure 6.6.

The *Controller* is explained in Section 6.2.5 below. In this simplified model, a *Client* may receive either their own operation back (which is matched using `isequal()` to the previously-stored `ackToAwaitInts` to recognize an acknowledgement) or a new operation from a different user. In the former case, the *Client* simply transitions to the *Synced* state and awaits new local or remote operations. In the latter case, the *Client* must resolve a local conflict to apply the new operation and re-attempt to send its unacknowledged local operation, which is accomplished by transitioning to the *ApplyingRemoteOpsWithoutACK* state. The `parseRemoteMessage()` graphical function called during this transition serves to update `remoteTS` and `remoteOps`, as shown in Figure 6.10.

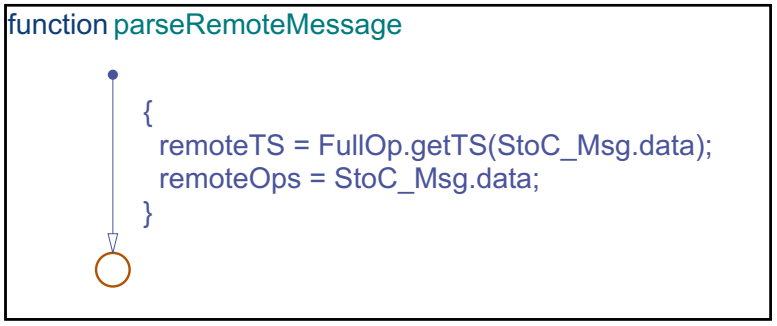



FIGURE 6.10: Contents of `parseRemoteMessage()` graphical function.

Figure 6.11 shows how the transformation sequence of Algorithm 4.3 was modeled using Stateflow. The timestamp `localTS` is first incremented so that the resulting operation is ready for sending by reusing the *SendingOpsToController* state once execution of *ApplyingRemoteOpsWithoutACK* has completed. The graphical functions shown in the bottom-left corner of the figure are all logging related and therefore follow previously-established logging patterns (Figure 6.4).

Section 2.2.1.2 defined *target* and *reference* operations in an OT transformation. The *TransformingForLocalApply* and *TransformingForSending* states are identical in content, but the `targetOps` and `refOps` variables are reversed before the states are executed in order to obtain both desired transformations. The *ApplyingOps* state of Figure 6.9 is reused by setting `opsToApply` to the transformed `primeOps` value obtained from *TransformingForLocalApply*.

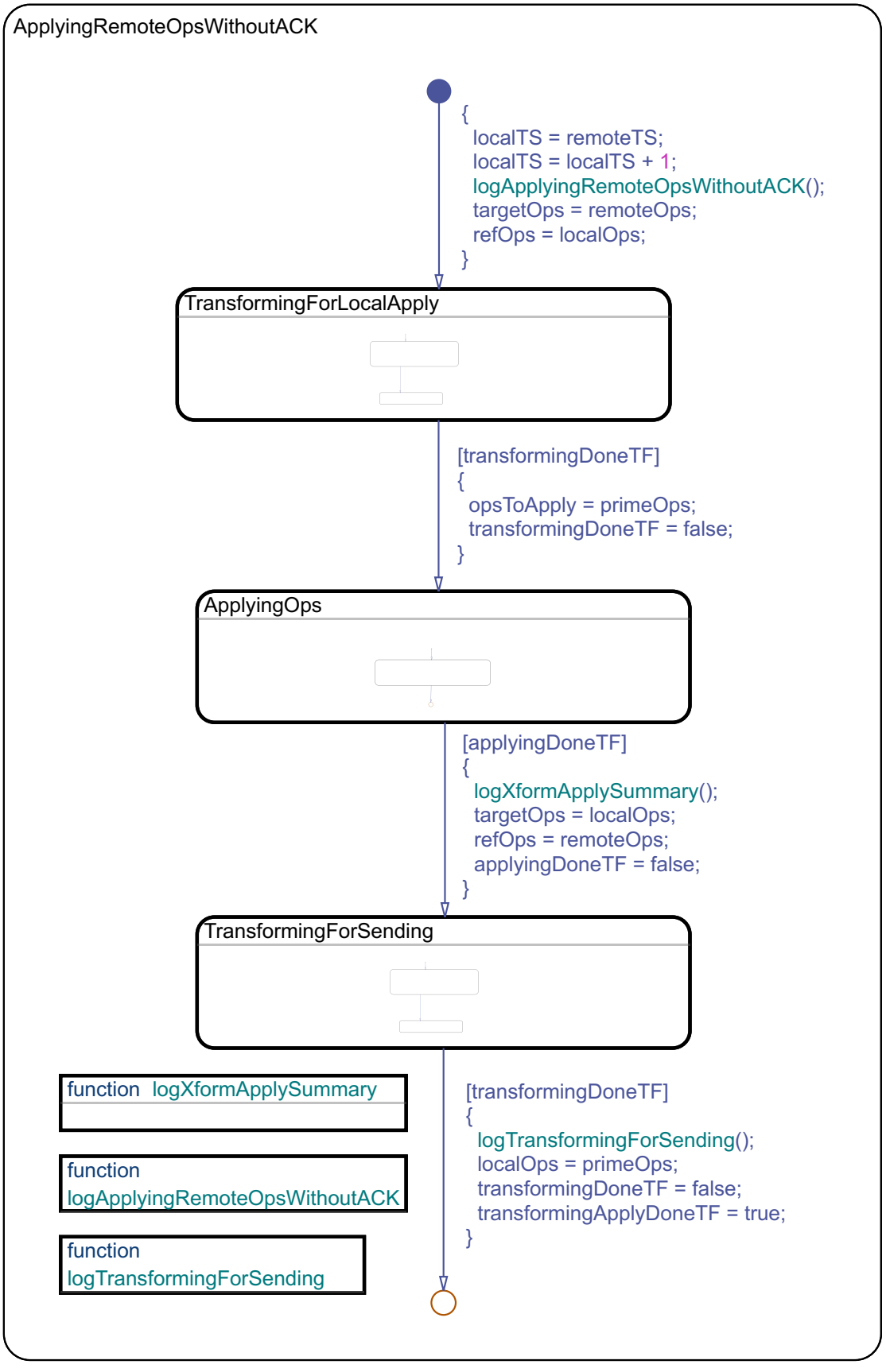



FIGURE 6.11: Basic Client ApplyingRemoteOpsWithoutACK state.

The contents of *TransformingForLocalApply* (and therefore also *TransformingForSending*) are given in Figure 6.12. `FullOp.getOpCode()` is used to determine the type of transformation based on the opcodes present. The transformations in the case of the *identity* operation are negligible so the target operation is simply set as *primeOps* from the inner *CompletingTransform* state. Later sections of this chapter will evolve the *TransformingForLocalApply* state to include more meaningful transformations.

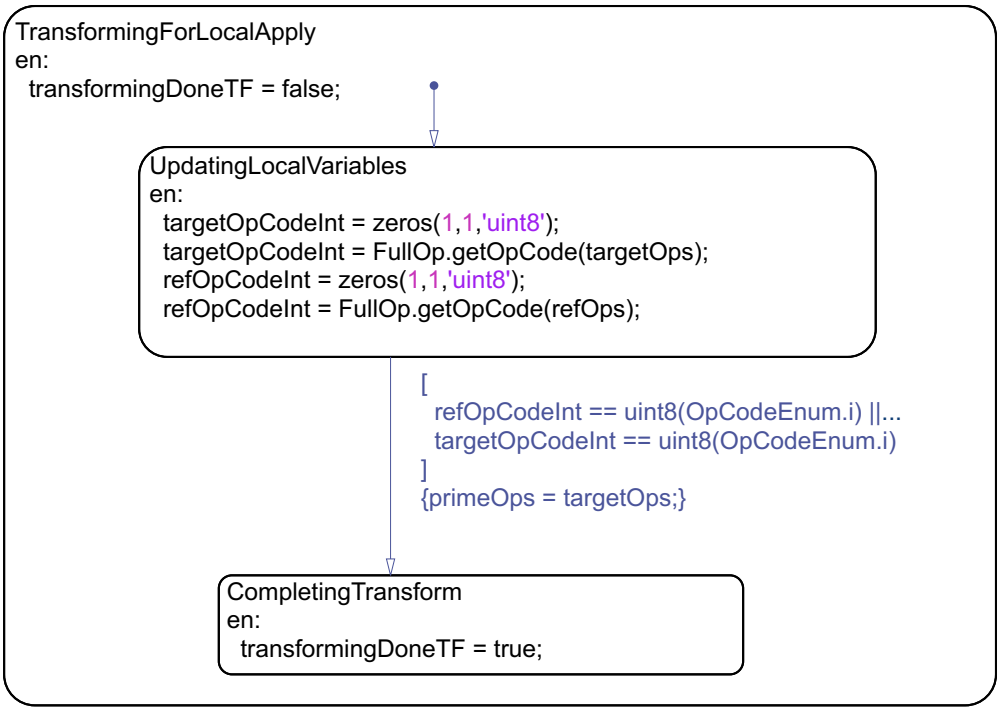



FIGURE 6.12: Basic Client TransformingForLocalApply state.

While in the *Synced* state, the *Client* may receive a new operation as a `StoC_Msg` from the *Controller*, thereby causing it to transition into the *ApplyingRemoteOps* state in order to apply the remote operation directly, as shown in Figure 6.13. Similar to the *ApplyingLocalOps* state of Figure 6.8, the `opsToApply` variable is updated to the value of `remoteOps` before the reusable state *ApplyingOps* of Figure 6.9 is executed.

### 6.2.5 Controller FSMs

By building on the description from Section 4.3 and the associated "serverless" assumptions, the *Controller* was modeled as shown in Figure 6.14. The *ControllerRunning* state features a similar initialization approach to the *Client*, whereby an

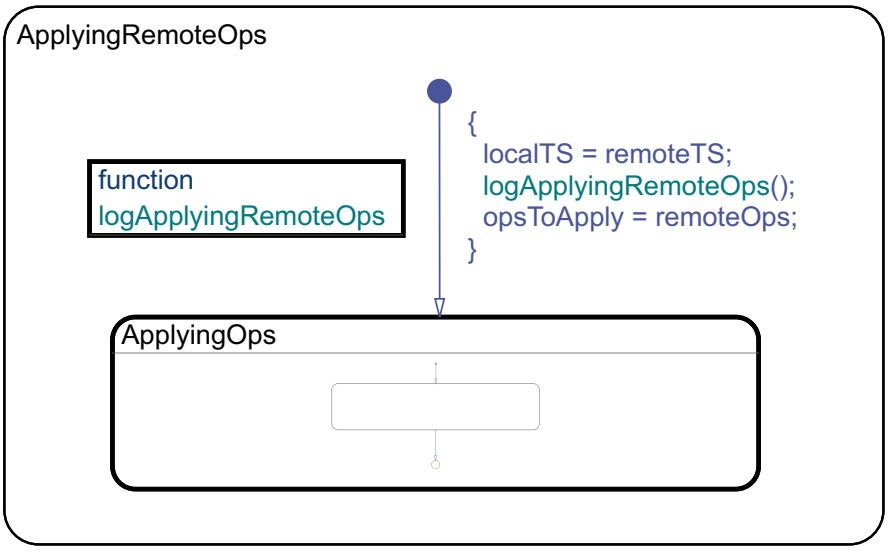



FIGURE 6.13: Basic Client ApplyingRemoteOps state.

*Initializing* state is used to configure variables before the default state of *Listening* is entered.

The top-left corner of the *ControllerRunning* state identifies code that is executed during (`du:`) its entire execution. At any time, the arrival of a *Cli1_In_Msg* or a *Cli2_In_Msg* on the ports of the *Controller* chart will cause the messages to be forwarded to a queue named `Msg_Queue`.

While in the *Listening* state, the presence of a message in `Msg_Queue` will cause the transition to be taken towards the junction near the bottom-left corner of Figure 6.14. During this transition, the received operation data is first set to `localOps`. The received timestamp is then extracted using `FullOp.getTS()` into the `remoteTS` variable. The sender of the operation is determined using `FullOp.getOpClientId()` and stored as an integer in `currentClientId`. The sender's name is also stored as a character array in `currentClientName` for use by the logging-related graphical functions.

The junction of Figure 6.14 then checks if the `remoteTS` timestamp value is larger than previous known values in order to reach the `PersistingNew` state, where the `localTS` value is updated. Reaching this state implies that the operation has been accepted into the *Controller*'s local history. The operation can therefore be acknowledged to the sending *Client*, which is achieved in the *SendingACKToClient* state shown in Figure 6.15.

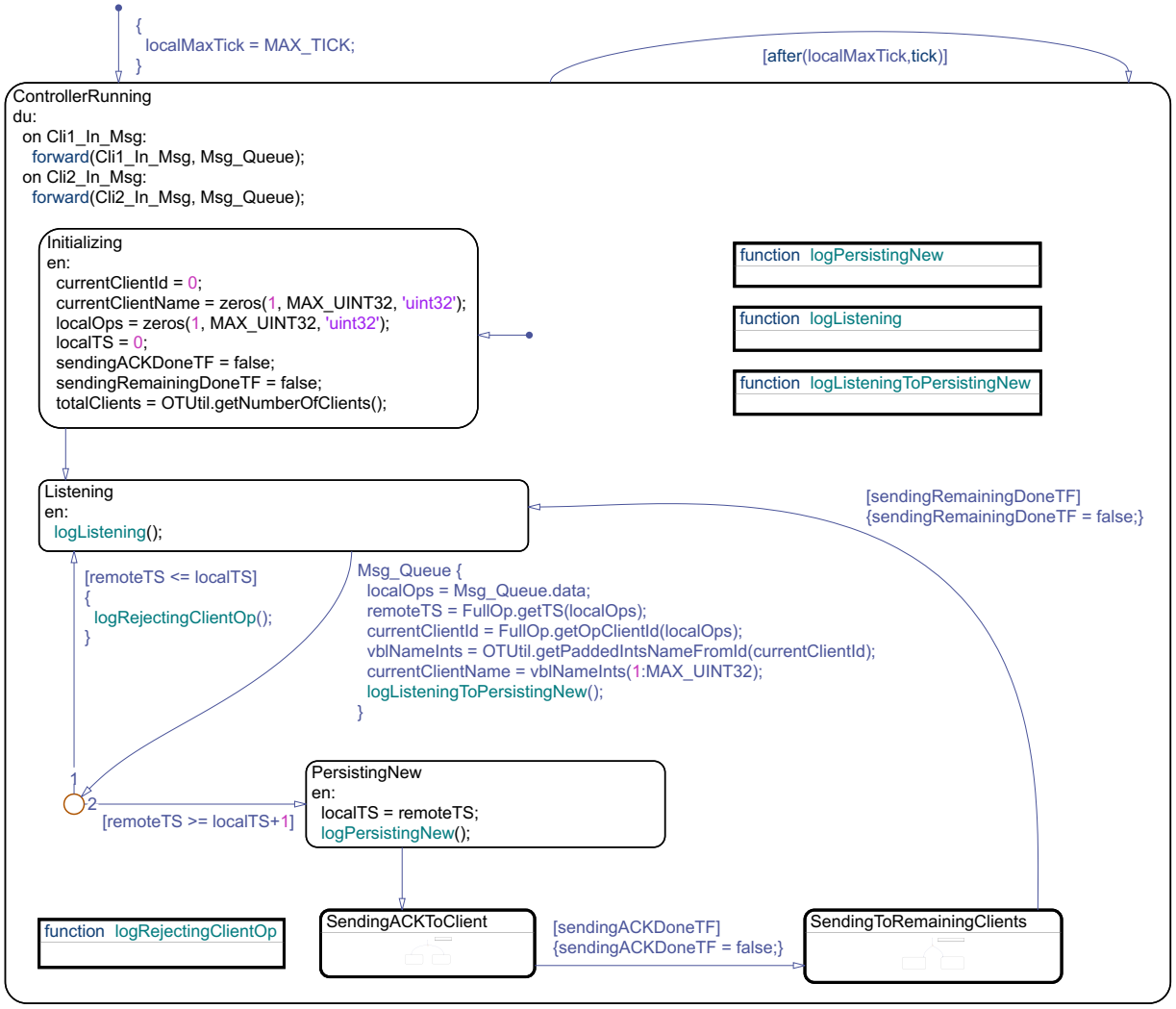



FIGURE 6.14: Two-user Controller state.

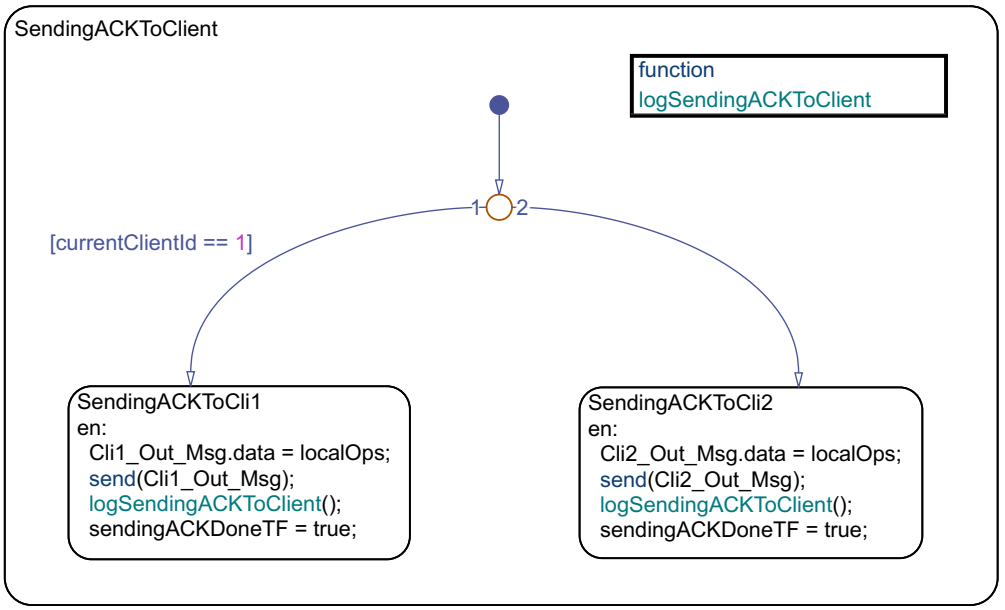



FIGURE 6.15: Two-user SendingACKToClient state of Controller.

The *SendingACKToClient* state of Figure 6.15 shows how the `send()` operation identifying the appropriate output port is selected based on the previously-set `currentClientId` value. For example, the state *SendingACKToCli1* will use the message port `Cli1_Out_Msg` shown in the high-level Simulink diagram of Figure 6.1, which leads to the `Controller Message` port on the *Client* Alice. The message data sent by the *Controller* contains the entire operation, and Section 6.2.4 showed how the operation gets matched with what each *Client* knows was sent last in order to recognize an acknowledgement.

The accepted operation must also be sent by the *Controller* to the other users participating in the session, which is accomplished by the *SendingToRemainingClients* state shown in Figure 6.16. The approach is similar to the *SendingACKToClient* state, except that the messages are sent to the opposite user of this two-user system.

In this *Controller* model, *SendingACKToClient* and *SendingToRemainingClients* are sending the same message, which could easily be accomplished with a single broadcast state that sends to both users. However, having two separate states remains true to the design of Section 4.3, which allows supporting custom acknowledgement messages that are more efficient than re-sending the entire operation and therefore more desirable in real-world implementations.

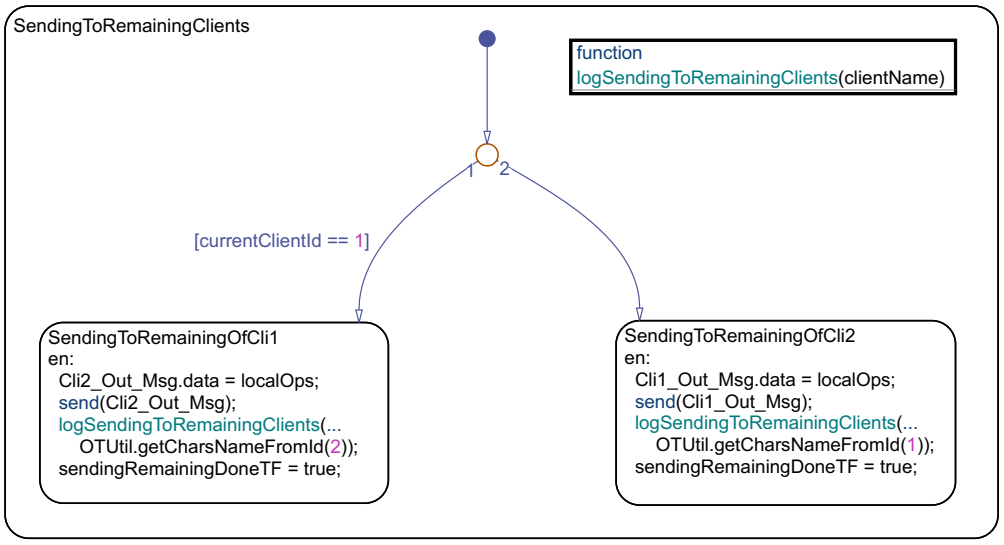



FIGURE 6.16: Basic SendingToRemainingClients state of Controller.

## 6.3 Basic OT FSM with Buffer

Section 3.5 established the need for a *AwaitingWithBuffer* state, and the design was extended in Section 4.2 with the states *ApplyingBufferedLocalOps*, *Awaiting-WithBuffer*, and *CreatingLocalOpFromBuffer*, as well as with the decision node for ensuring that all operations from the buffer are sent. This section will extend the basic *Client Chart Library* of Section 6.2.4 to add the buffer functionality, thereby including all main components of the CLOT integration algorithm. As this change is specific to the *Client* chart, the high-level Simulink models described in Section 6.2 (including the *Events Generator* and *Controller* charts) remain unchanged. The *Client* diagram of Section 6.2.4 is therefore updated to include the buffer as shown in Figure 6.17.

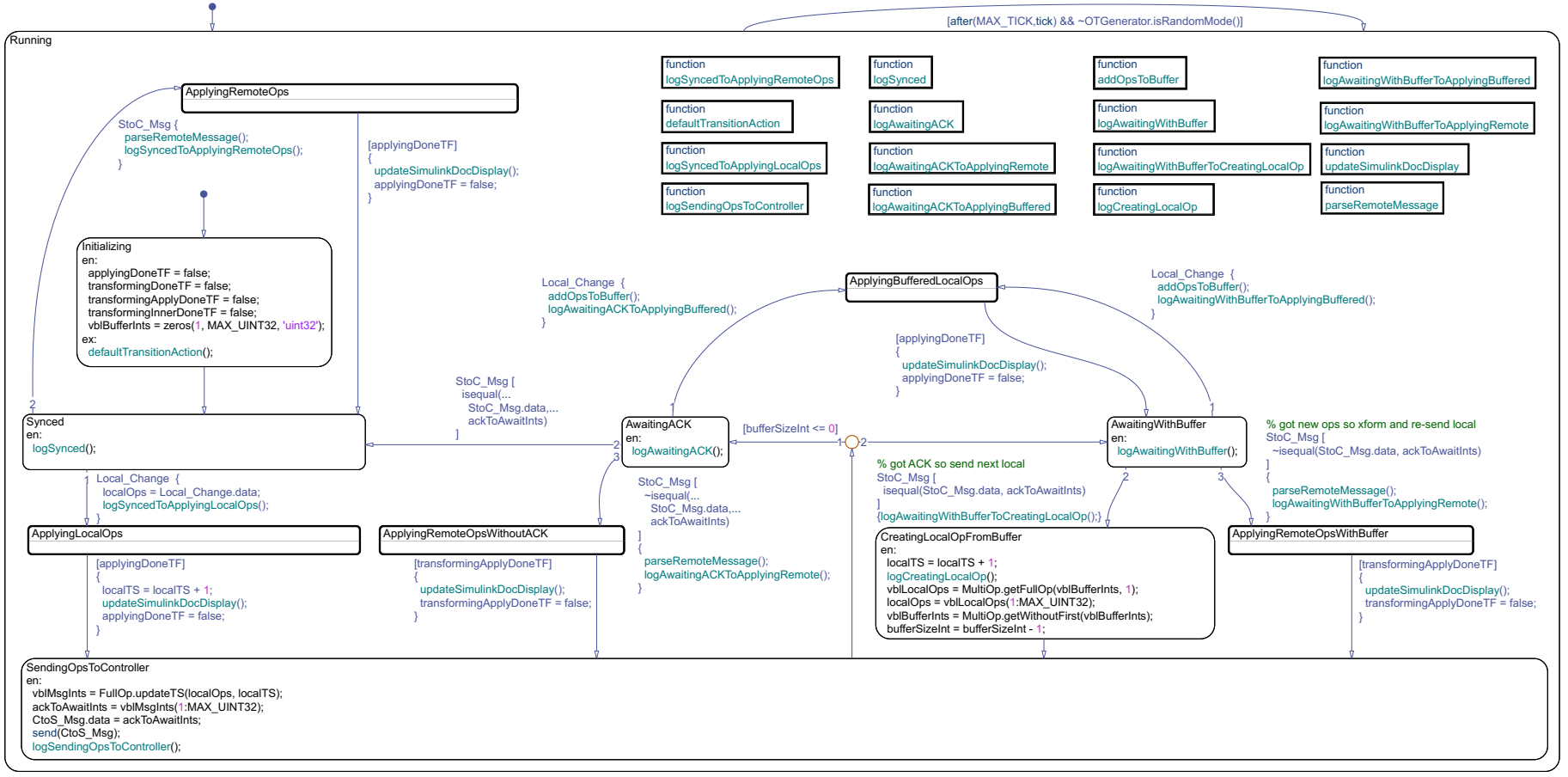



FIGURE 6.17: Basic Client State Machine based on Synced, AwaitingACK and AwaitingWithBuffer states.

The new transition out of *AwaitingACK* checks for the presence of a `Local_Change` message from the *Events Generator* and calls the `addOpsToBuffer()` graphical function, which is shown in Figure 6.18. As previously revealed in the class diagram of Figure 6.2, the *MultiOp* class contains a list of static methods to allow managing a buffer of operations directly from Stateflow. This implies that *MultiOp* provides methods to accept parameters of integer arrays and return integer arrays representing arrays of operations as ASCII characters, where each operation is encoded using the JSON-based *FullOp* format. The `addOpsToBuffer()` graphical function first sets `newLocalOps` to the contents of the received `Local_Change` message. It then makes use of `MultiOp.addOp()` to add the `newLocalOps` to the `vblBufferInts` variable, while the variable `bufferSizeInt` is used to track the operation buffer size.

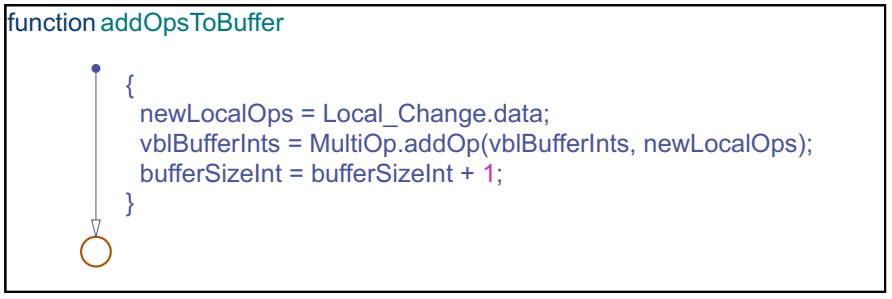



FIGURE 6.18: Contents of `addOpsToBuffer()` graphical function.

The first step of Algorithm 4.4 is therefore performed by `addOpsToBuffer()` on the transition line leaving the *AwaitingACK* state in order to allow the *ApplyingBufferedLocalOps* state to follow the same structure previously used in Figure 6.8 for *ApplyingLocalOps* (as well as Figure 6.13 for *ApplyingRemoteOps*), where `opsToApply` is instead set to `newLocalOps`. Once the new operation has been applied to the local document in this fashion, the transition is taken into the *AwaitingWithBuffer* state, since the *Client* is still awaiting the acknowledgement for the original operation before the newly buffered operation can be sent. The presence of another `Local_Change` message while in *AwaitingWithBuffer* will repeat the `addOpsToBuffer()` and *ApplyingBufferedLocalOps* sequence to ensure that all local events from the *Events Generator* are immediately processed and displayed to the *Client* (an important property of OT systems as per Section 2.2).

While in the *AwaitingWithBuffer* state, the presence of a `StoC_Msg` message containing an operation equal to the previously-sent operation (`ackToAwaitInts`) implies an acknowledgement and allows the next operation from the buffer to be

prepared for sending. This is achieved by the *CreatingLocalOpFromBuffer* state, which increments the `localTS` value and obtains the oldest operation from the buffer by using `MultiOp.getFullOp()`. The operation is set to `localOps`, which is then used by the *SendingOpsToController* state for creating a message that reaches the *Controller*'s input port corresponding to the *Output Message* port of the *Client Chart Library*. The operation is also removed from the buffer by using `MultiOp.getWithoutFirst()` and the `bufferSizeInt` variable is adjusted for this reduction. The transition out of the *SendingOpsToController* state leads to a junction that returns back to the *AwaitingACK* state if the last buffered operation was sent, or returns to *AwaitingWithBuffer* if additional operations remain to be sent from the buffer.

The third possible transition out of the *AwaitingWithBuffer* state of Figure 6.17 occurs with the arrival of a `StoC_Msg` message containing a new remote operation, resulting with a transition to the *ApplyingRemoteOpsWithBuffer* state. As previously shown in Section 4.2, several loops are required in order to transform the remote operation against the existing buffer before the remote operation can be applied and a new local operation can be created for re-sending via *SendingOpsToController*. This scenario will be shown using FSMs in the next section since the explanation of the FSM-based design benefits from supporting more than just an *identity* operation in the OT system.

## 6.4 Buffered OT FSM with Insert and Delete

Now that the main components of the CLOT algorithm have been modeled, this section will extend the *Client* to support character-based *insertData* and *deleteData* operations in addition to the *identity* operations shown in the OT systems of Section 6.2 and Section 6.3. No changes are required to the buffered top level *Client Chart Library* FSM of Figure 6.17, the high-level Simulink model of Figure 6.1, the *Controller* of Figure 6.14, or the *Events Generator* of Figure 6.3, although calls to the *OTGenerator* can now return operations that include the new operation types. Only minor changes are required to the class diagram of Figure 6.2, such as the *OpCodeEnum* class gaining the *insertData* and *deleteData* properties, and the *Op* class gaining a *con* (content) property for handling a character and a position value now required for insertion and deletion operations (which the *FullOp* class allows retrieving via `FullOp.getOpCon()`). The *Op* class

creates an instance of a new *OpCon* class for storing operation content properties as character arrays (in order to remain flexible for future operation requirements) and enabling their retrieval as a JSON-based character array. A more complete class diagram is provided after the introduction of additional operations for supporting DOM synchronization in Section 6.5, where the DOM-based versions of *insertData* and *deleteData* are also discussed.

Section 6.2.4 reviewed how a *Client* can leave the *Synced* state by receiving a `Local_Change` message from the *Events Generator* that must first be applied to the local document via *ApplyingLocalOps* (Figure 6.8), or by receiving a `StoC_Msg` containing a remote message to be applied locally via *ApplyingRemoteOps* (Figure 6.13). These two states, along with the *ApplyingBufferedLocalOps* state mentioned in Section 6.3 (and the *ApplyingRemoteOpsWithBuffer* state described at the end of this section), all make use of the same *ApplyingOps* state to make this modification. Since `FullOp.getOpCode()` can now return *insertData* and *deleteData* operation types, the *ApplyingOps* state of Figure 6.9 has been modified to include a junction to evaluate such possibilities, as shown in Figure 6.19.

While processing *identity* operations remains straightforward, the *ApplyingInsertData* state within the *ApplyingOps* state must perform the necessary string manipulations for inserting the requested character at the correct position within the character array of `vblDocInts` containing the latest document content of the *Client*. The JSON-based *insertData* encoding of Section 5.2.4 showed how the *opcon* section contains an array of the structure `opcon: [letter, position]`. Simulink constants are therefore defined using *Model Explorer* to represent these array locations based on the corresponding index, namely `insDelDataConLetterIdx` with a value of `0` and `insDelDataConPosIdx` with a value of `1`. By using the JSON parsing functions of `FullOp.getOpCon()`, the character to be inserted is obtained from the message as `targetOpConLetterInt` and its position as `targetOpConPosInt`.

Trimming and concatenation techniques of the MATLAB Language are used to obtain a new `vblDocInts` array that includes the new letter. Trimming was previously shown in Section 6.2.2, while concatenation occurs with the syntax `['a' 'b' 'c']` to get `'abc'`, where `a`, `b` and `c` are integers or integer arrays representing characters in *ApplyingInsertData* (namely, `targetOpConLetterInt` is in the `b` position and the presence of the ellipsis also permits the concatenation to take

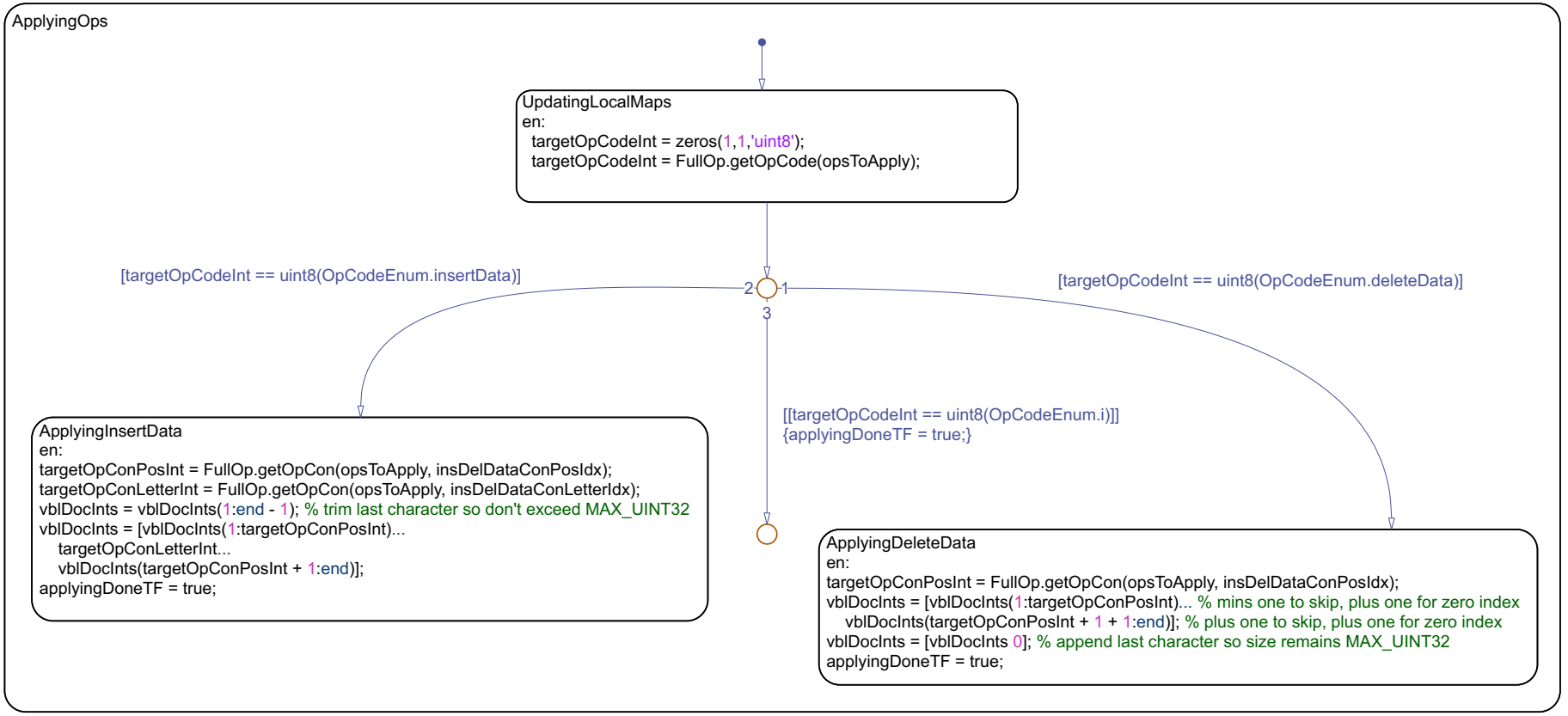



FIGURE 6.19: Client ApplyingOps state with insertData and deleteData support.

place). Similarly, the *ApplyingDeleteData* state removes a letter from `vblDocInts` at position `targetOpConPosInt`.

Although their syntax may not be intuitive, the previously-described trimming and concatenation techniques allow more of the required logic to be kept at the Stateflow diagram level. However, more traditional substring functionality can be accessed by calling static methods or functions from `.m` files, which will be employed increasingly with the growing complexity of the system. For example, this can be seen with the calls to `getOpCon()` from `FullOp.m`, which parses the operation content by using JSON libraries that are otherwise not easily accessible from Stateflow Action Language directly.

The *ApplyingRemoteOpsWithoutACK* state, which is the conflict resolution state reached when a *Client* receives a new remote operation instead of an acknowledgement of their own operation, has not changed and remains modeled as per Figure 6.11. However, the inner *TransformingForLocalApply* state, which was previously shown in Figure 6.12, must now support the four transformations identified in Section 2.2.1.2 in order to process possible conflicts with the *insertData* and *deleteData* operations. Figure 6.20 shows the updated *TransformingForLocalApply* state, where `FullOp.getOpCode()` is used to determine the type of transformation based on the *target* and *reference* opcodes present in `targetOpCodeInt` and `refOpCodeInt`, respectively. The obtained opcodes are then compared with the opcodes in `OpCodeEnum` using the gates of the transition lines exiting the junction.

A transformation between two operations of type `OpCodeEnum.insertData` results with the system transitioning into the *TransformingInsertDataVsInsertData* state shown in Figure 6.21, where the logic of Algorithm 2.1 can be seen implemented as a Stateflow FSM. The `FullOp.getOpCon()` static method is used to retrieve the positions of both operations, while `FullOp.getOpClientId()` retrieves the numeric *Client* identifiers. The *IncreasingPosition* state is reached if the insertion within the received *target* operation occurred at a position after the local *reference* operation, in which case the resulting transformed *target* operation is incremented by 1. The replacement in the characters representing the operation is made by using the `FullOp.updateOpCon()` static method and the result is stored in `primeOps` before it is applied locally using the *ApplyingOps* state within *ApplyingRemoteOpsWithoutACK* (Figure 6.11).

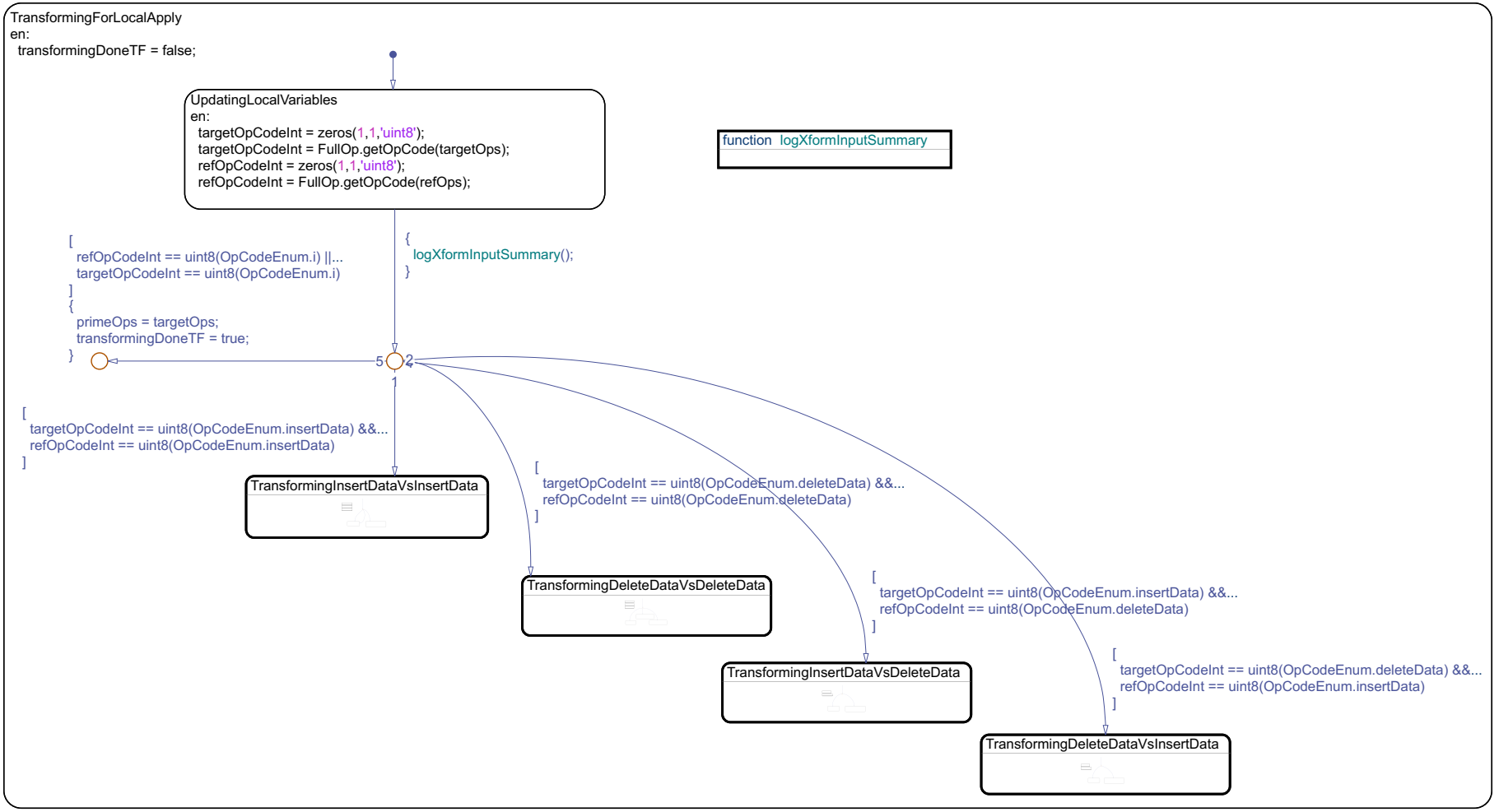



FIGURE 6.20: Client TransformingForLocalApply state with insertData and deleteData support.

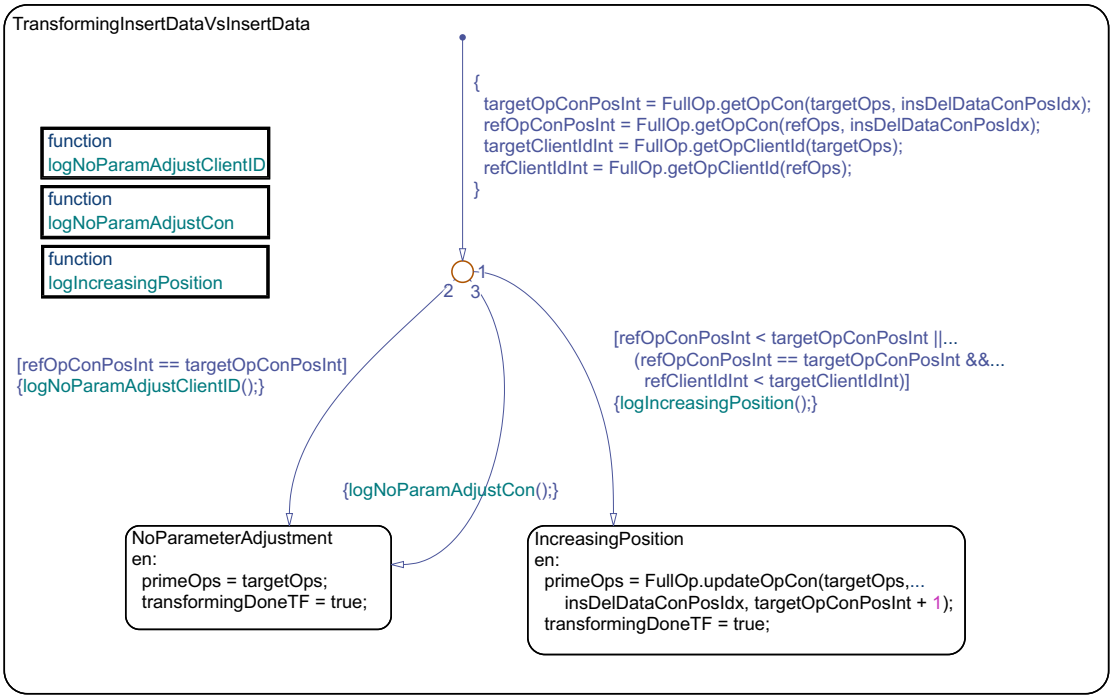



FIGURE 6.21: Client TransformingInsertDataVsInsertData state.

Similarly, Figure 6.22 shows the transformation logic between two *deleteData* operations (Algorithm 2.2) modeled as the *TransformingDeleteDataVsDeleteData* state within the *TransformingForLocalApply* state of Figure 6.20. This includes the ability to return an identity operation by reaching the inner *SettingIdentityOp* state. The states *TransformingInsertDataVsDeleteData* and *TransformingDeleteDataVsInsertData* follow the same pattern to implement Algorithm 2.3 and Algorithm 2.4, respectively.

Section 6.3 mentioned how the *ApplyingRemoteOpsWithBuffer* state is reached when a *Client* is awaiting an acknowledgement for an initial operation but has created at least one other local operation (which must be buffered for sending) and then received a new remote operation instead of the acknowledgement for the initial operation. Before the remote operation can be applied, it must be transformed against the initial local operation as well as all buffered operations. A new version of the initial operation must also be obtained and sent to the *Controller*. Algorithm 4.5 of Section 4.2 detailed this process, while Figure 6.23 and Figure 6.24 show how this algorithm was adapted into a FSM using Stateflow. The *ApplyingRemoteOpsWithBuffer* state is split into two halves, with a small lightning bolt symbol added to show the continuation of the interrupted transition line.

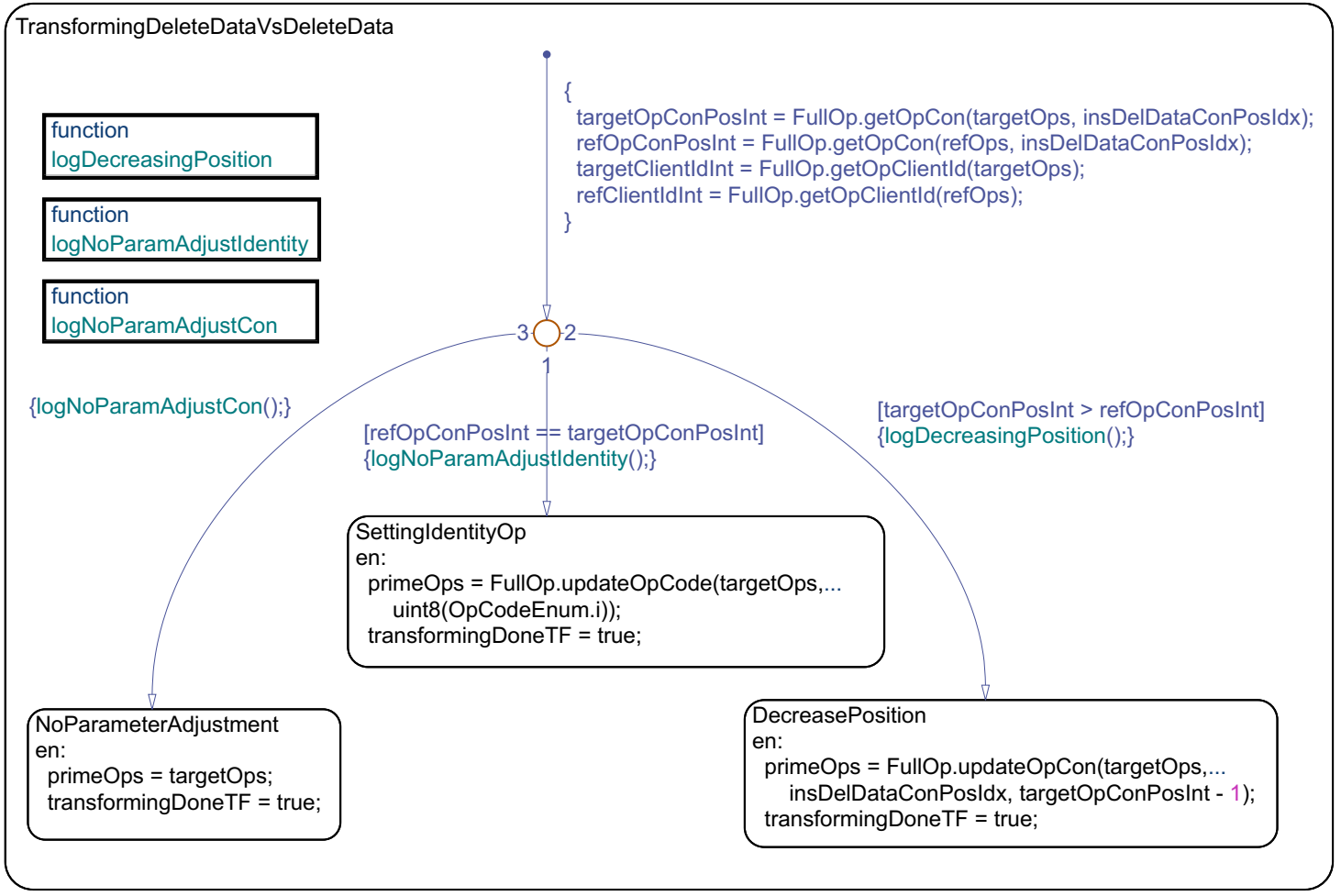



FIGURE 6.22: Client TransformingDeleteDataVsDeleteData state.

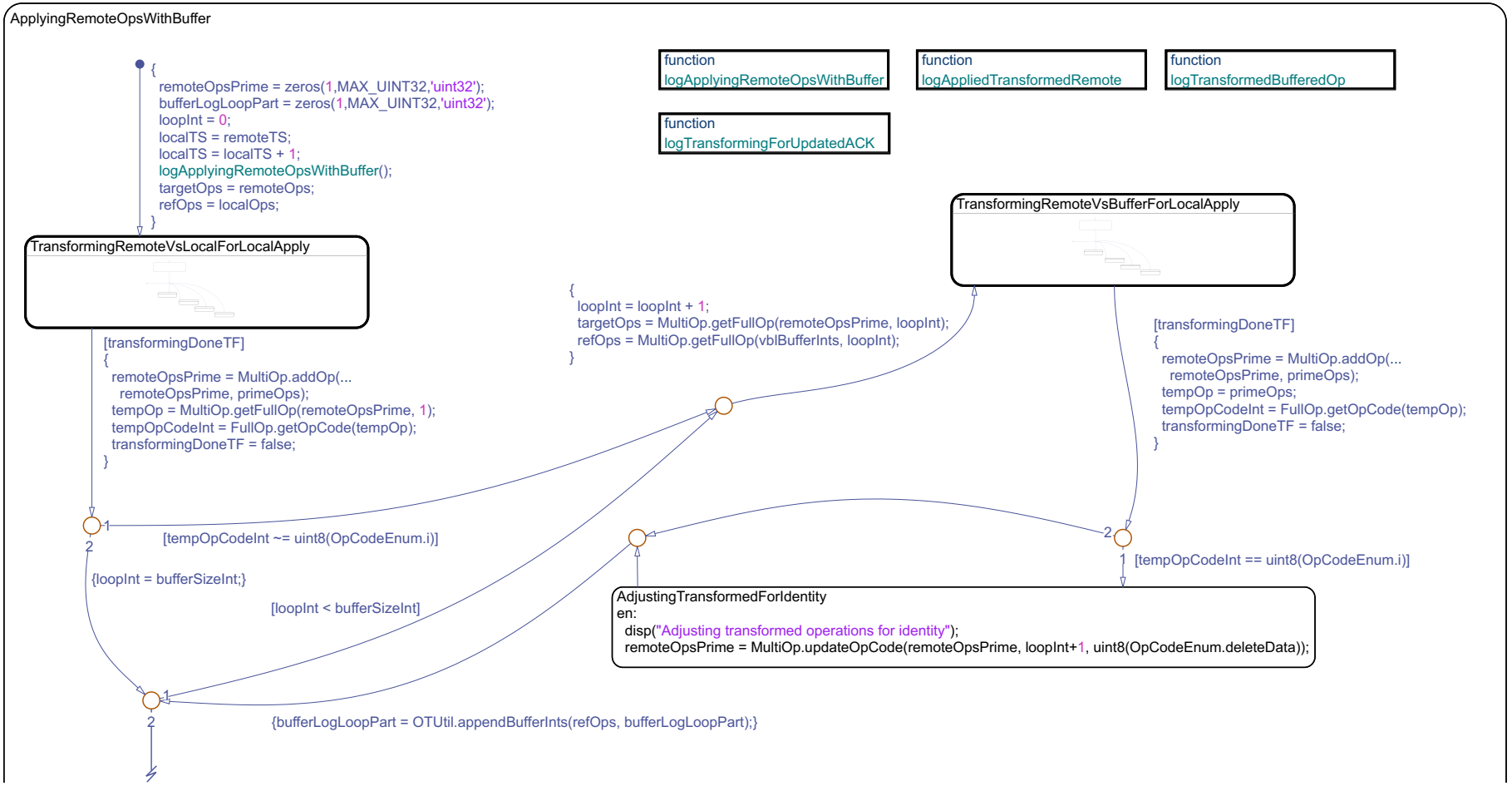



FIGURE 6.23: Top half of ApplyingRemoteOpsWithBuffer state of Client.

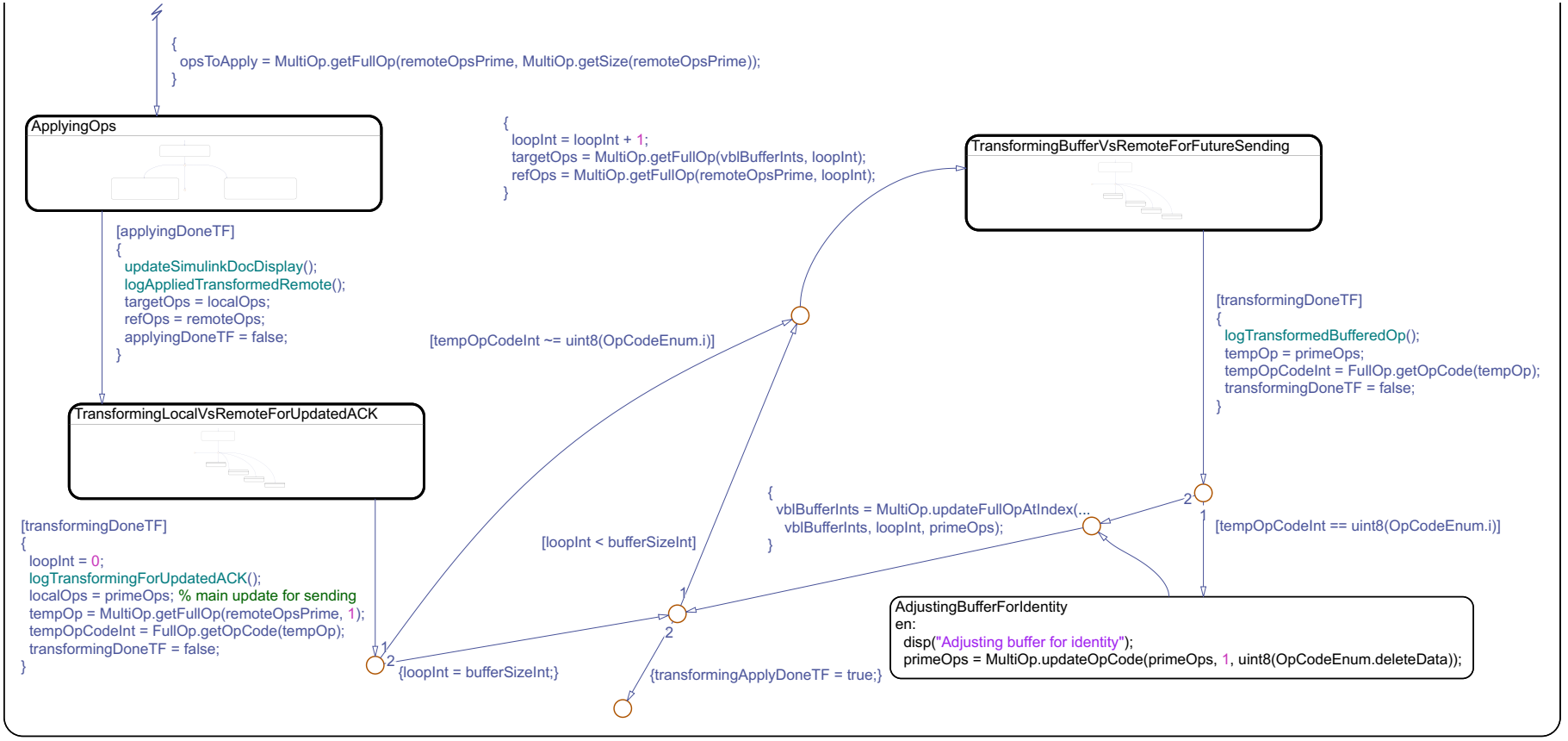



FIGURE 6.24: Bottom half of ApplyingRemoteOpsWithBuffer state of Client.

Figure 6.23 shows the top half of the *ApplyingRemoteOpsWithBuffer* state. Similar to the *ApplyingRemoteOpsWithoutACK* state described in Section 6.2.4 (Figure 6.11), the `localTS` value is first updated and incremented based on the remote operation. The remote operation is then transformed against the unacknowledged local operation in the *TransformingRemoteVsLocalForLocalApply* state, whose contents are the same as the *TransformingForLocalApply* and *TransformingForSending* states of Section 6.2.4 (Figure 6.12). In fact, there are four such states within the *ApplyingRemoteOpsWithBuffer* state, namely *TransformingRemoteVsLocalForLocalApply* and *TransformingRemoteVsBufferForLocalApply* in the top half (Figure 6.23), but also *TransformingLocalVsRemoteForUpdatedACK* and *TransformingBufferVsRemoteForFutureSending* in the bottom half (Figure 6.24).

The `primeOps` resulting from the transformation performed by *TransformingRemoteVsLocalForLocalApply* are added to a `remoteOpsPrime` array. If the resulting operation is an *identity* operation (which can occur as the result of a *deleteData* vs. *deleteData* transformation), there is no need to transform it against the operations in the buffer (this is an optimization on Algorithm 4.5). The `opsToApply` variable is set to the transformed operation (as per the top of Figure 6.24) and the system transitions to the *ApplyingOps* state, whose contents were also previously seen in Section 6.2.4. Returning to Figure 6.23, if the *TransformingRemoteVsLocalForLocalApply* state did not result with an *identity* operation, a loop is entered whereby the remote operation is incrementally transformed against all operations of the *vblBufferInts* array based on an increasing `loopInt` index, as previously shown in Algorithm 4.5. The `MultiOp.addOp()` static method is used to append the resulting `primeOps` to the `remoteOpsPrime` array in order to form the intermediate stages of the transformed remote operation. As an additional improvement on Algorithm 4.5 for supporting *insertData* and *deleteData*, the junction leading to the state *AdjustingTransformedForIdentity* is then used to preserve any operations that were returned as *identity* operations during the previous transformation and restore them as *deleteData* operations. The `bufferLogLoopPart` variable is then set in order to assemble a useful console log message for later output via the `logAppliedTransformedRemote()` graphical function. The fully-transformed version of the initial operation in `remoteOps` will therefore end up as the last element of the `remoteOpsPrime` array. Once there are no additional operations in `vblBufferInts` and the loop has completed, the `opsToApply` variable is set and the `ApplyingOps` state is reached.

Figure 6.24 shows how the transition leaving the `ApplyingOps` state prepares the `targetOps` and `targetRef` variables for input to the *TransformingLocalVsRemoteForUpdatedACK* state, where the target operation is now the local operation for which the acknowledgement is being awaited. The result of the transformation is therefore an updated *localOps* variable which is to be sent to the *Controller* once the *ApplyingRemoteOpsWithBuffer* state has completed. A similar loop structure to Figure 6.23 is then entered in order to transform all buffered operations in `vblBufferInts` against the intermediate operations previously stored in `remoteOpsPrime` with use of the *TransformingBufferVsRemoteForFutureSending* state, still adjusting for any *identity* operations encountered via the *AdjustingBufferForIdentity* state. By using `MultiOp.updateFullOpAtIndex`, the content of `vblBufferInts` is therefore updated to contain the buffer operations for future sending to the *Controller* by the *CreatingLocalOpFromBuffer* state of Figure 6.17 once the updated `localOps` operation has been sent and acknowledged.

## 6.5 Advanced OT FSM

This chapter first showed how an OT system can be modeled based on the proposed CLOT algorithm, and then showed how its hierarchical design allowed for the gradual introduction of new operations and transformations. In this final section, the system will be upgraded to support some of the more challenging DOM-based operations introduced in Chapter 5, as well as a third instance of the *Client*.

### 6.5.1 HTML DOM-Based OT FSMs

The focus of this section is to update the *Client Chart Library* to model two highly-representative DOM-based operations defined in Chapter 5, namely *insertNode* and *giveData*, as well as the associated algorithms and data structures. The corresponding *deleteNode* and *receiveData* operations, along with the *moveNode* and attribute-related operations, can be derived based on the patterns established from the more challenging *insertNode* and *giveData* operations explored here.

The *giveData* operation is especially noteworthy as it can be used to achieve DOM node splitting, a capability that is essential for proper OT intention preservation when hierarchical DOM-based edits are involved. As was described in

Section 5.2.6, actions such as bolding of text will create a nested *strong* node, which breaks up the previous text node. Text-based operations that were targeting positions in the un-split version of the text node must be carefully transformed to find the new locations of those targeted positions, which may require "jumping over" several new nodes.

#### 6.5.1.1 Functions and Classes

One of the strengths of using MATLAB for modeling the proposed OT system and architecture is the included cross-platform access to its powerful libraries. Since XML and HTML files are frequently encountered when working with scientific data, MATLAB includes built-in access to an embedded Java version of the `org.apache.xerces.dom` library [132]. However, in order to allow results from this MATLAB Language component to reach the Stateflow Action Language, several `.m` file functions needed to be loaded using the `coder.extrinsic()` feature of Stateflow [131]. Calls to the XML library, as well as resulting XML-related variables such as *realDOM* and *realMap* introduced in Chapter 5, must therefore remain at this *extrinsic* layer of MATLAB Language code. Careful management of array indexes is also required, as calls to Java methods use zero-based indexing, while the MATLAB Language and Stateflow Action Language use one-based indexing.

Table 6.4 provides a list containing a subset of the new `.m` file functions, along with a brief description of each function. In order to help differentiate from static methods and graphical functions, `.m` file function names do not end with `()` in this thesis. Not included in the table are functions whose purpose can be deduced based on the similarity of their name to functions on the given list, namely *applyDeleteData*, *applyGiveData* and *applyInsertNode* (see *applyInsertData*), *clientApplyingRemoteOpsWithoutACK* (see *clientApplyingOps*) and *xformGiveDataVsDeleteData* (see *xFormInsertNodeVsDeleteData*). Additional details for all functions, including the input and output variables and their corresponding data types, are provided in Appendix A.

The final class diagram of the DOM-based CLOT system is shown in Figure 6.25. While the *MultiOp* class remains unchanged from the previous iteration of Figure 6.2, the static method list of *FullOp* has been enhanced significantly to support the more elaborate JSON structure, along with the retrieval and updating of its

TABLE 6.4: Subset of functions required for DOM-based OT FSM Simulation.

| Function | Description |
|---|---|
| **applyInsertData** | Apply *insertData* operations to the extrinsic state of the given *Client*. |
| **clientApplyingOps** | Update the *Client*'s extrinsic state from the *ApplyingOps* state. |
| **clientDefaultTransition** | Initialize the *Client*'s extrinsic state. |
| **diffDOM** | Obtain a FullOp of differences in the *Client*'s extrinsic state. |
| **genMapFromDOM** | Obtain a *OTMap* object from the DOM node input parameter. |
| **genTransMap** | Generate a transitional *OTMap* object from an existing *OTMap* and a *FullOp*. |
| **getApplySummary** | Log a summary after an operation was applied from a function such as *applyInsertData*. |
| **getSyncedSummary** | Log the *Client*'s extrinsic state from the *Synced* state. |
| **getRealDOM** | Retrieve an integer array of the *realDOM* for a given *Client ID*. |
| **getXformApplySummary** | Log a summary after a transformation result was applied. |
| **xFormInsertNodeVsDeleteData** | Transform a target *InsertNode* operation vs. a reference *DeleteData* operation. |

various components. The previously-seen *getTS()*, *updateTS()*, *getOpClientId()*, and *getOpCode()* static methods remain. The remainder of the *FullOp* static methods have been introduced to help enable the DOM synchronization functionality of this more advanced OT system. *FullOp* now also has the *Payload* and *PayloadAux* components previously introduced in Figure 5.1 of Section 5.1.1. The *Op* class supports the *loc* property with instances of the *OpLoc* class, while the *OpCon* class introduced in Section 6.4 is also shown. *OTConst* has been extended with DOM-based constants required by the numerous new tests in *OTGenerator*, while *OTUtil* remains the same. The *OTMap* class has been introduced to enable the

"map" functionality presented in Section 5.1.2, where node objects (in this case, objects of type `ElementNSImpl` from Xerces) are maintained in an array structure. The *OTXML* class is also new and provides useful utility functions for working with XML. These two new classes are used by functions such as *genMapFromDOM* and *genTransMap* mentioned in Table 6.4. Finally, *OpCodeEnum* now lists the targeted DOM operation types, *ClientNameEnum* now includes *Carol* as the third user (discussed further below in Section 6.5.2), and *DOMNodeTypeEnum* has been added to contain *ELEMENT_NODE* (type 1), *TEXT_NODE* (type 3) and *DOCUMENT_NODE* (type 9) for easier matching of DOM node types as support for new types is introduced.

#### 6.5.1.2 Initialization Updates

Other than the changes required for the introduction of a third *Client* (Section 6.5.2 below), the control loop design based on the CLOT integration algorithm remains as established in the previous sections of this chapter, with the top level *Client* diagram of Figure 6.17 staying the same. However, the contents of many of the inner states and graphical functions have continued to evolve in order to support the new DOM-based requirements.

The *Running* state of Figure 6.17 began by entering the *Initializing* state, which initialized several variables before calling the `defaultTransitionAction()` graphical function. As shown in Figure 6.26, the updated `defaultTransitionAction()` graphical function now extrinsically calls the *clientDefaultTransition* function introduced in Table 6.4. The beginning of the corresponding `clientDefaultTransition.m` file is given in Listing 6.1 and is explained below.

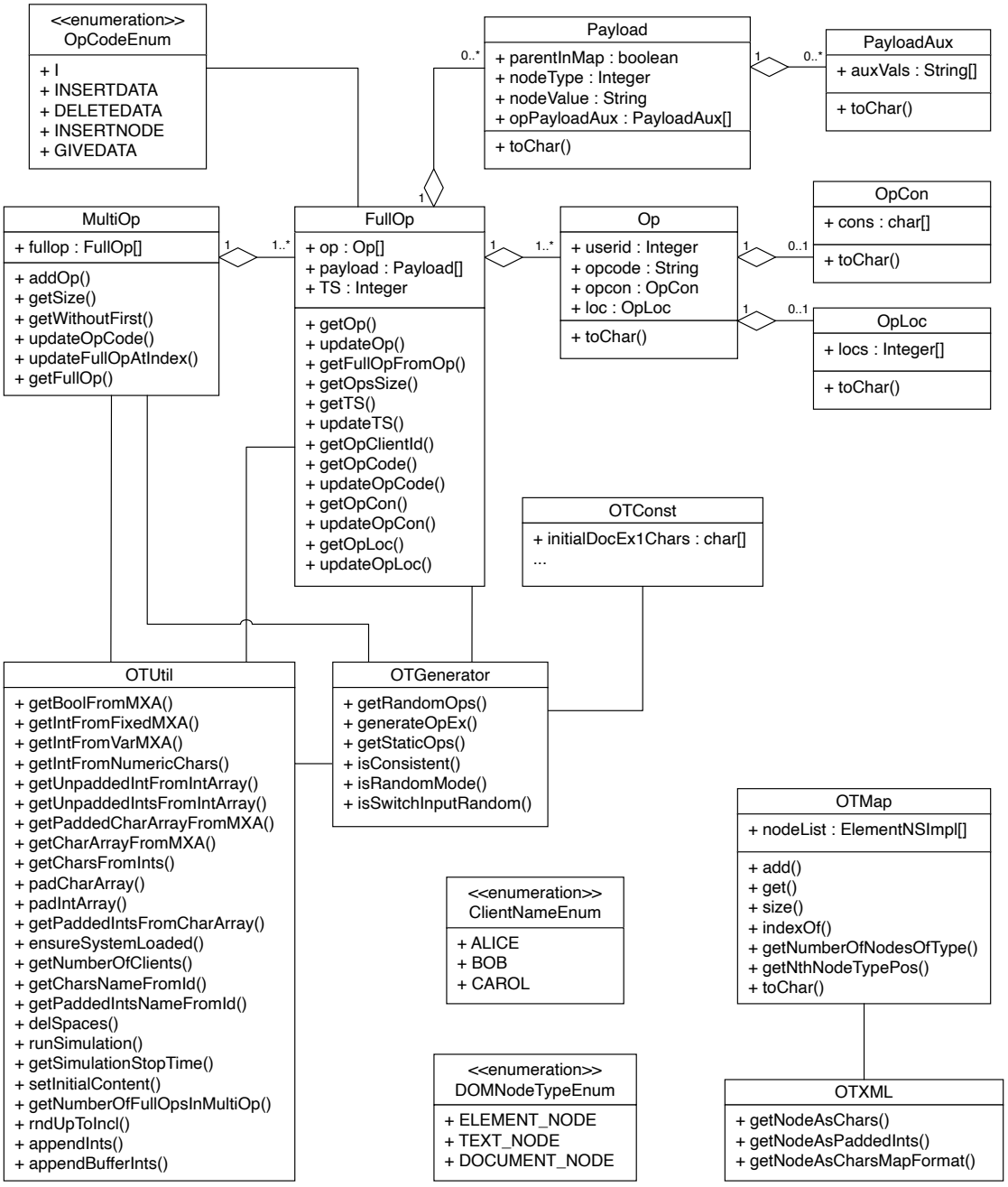



FIGURE 6.25: Classes used for DOM-based model.

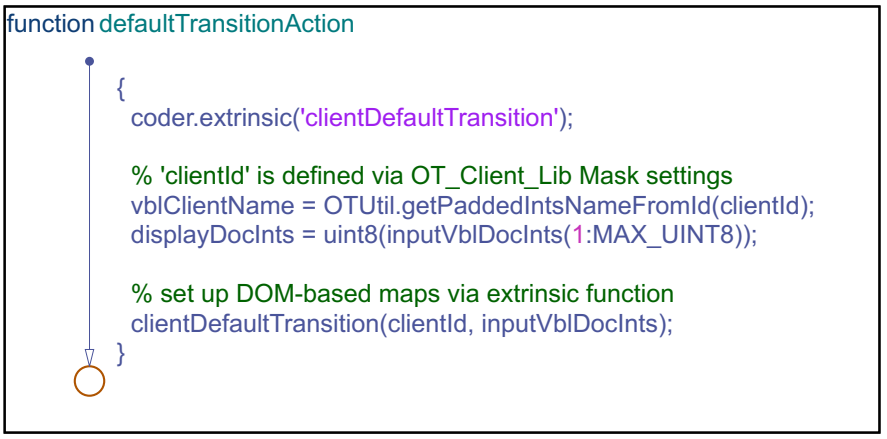



FIGURE 6.26: Contents of DOM-based `defaultTransitionAction()` graphical function.

```
1  function clientDefaultTransition(cliId, inputDocIntArray)
2    global realDOM realMap;
3    inputDocChars = OTUtil.getCharsFromInts(inputDocIntArray);
4    xmlStream = java.io.StringBufferInputStream(inputDocChars);
5    docNode = xmlread(xmlStream);                        % Document Node (Type 9)
6    docRootEleNode = docNode.getDocumentElement();       % Element Node (Type 1)
7
8    realDOM{cliId} = docRootEleNode.cloneNode(true);
9    realMap{cliId} = genMapFromDOM(realDOM{cliId});
10   ...
11 end
```

LISTING 6.1: Partial code of `clientDefaultTransition.m` function using MATLAB Language.

Most comments have been omitted from the `clientDefaultTransition.m` section shown in Listing 6.1, with the full header of the function provided in Appendix A. The purpose of this function is to carry out the requirements of Algorithm 5.1 presented in Chapter 5. In order for variables like `realDOM` and `realMap` to be accessible across extrinsic calls to functions from Stateflow, such variables were defined as `global` arrays on line 2, where the *Client ID* is used as an index into each array. Although not ideal from a data isolation perspective since all *Client* instances are accessing variables from the same shared MATLAB workspace, this workaround allowed managing the multiple instances of DOM-related values required.

The `inputVblDocInts` variable is passed from the `defaultTransitionAction()` graphical function of Figure 6.26 to the *clientDefaultTransition* function, where

it is received as `inputDocIntArray`. This variable contains an integer array of the characters within the *Initial Document Content* block of the Simulink layer, as was previously covered in Section 6.2.4. Line 3 of Listing 6.1 uses the `OTUtil` class to cast the received integer array into a useful character array, which line 4 then converts into a Java-based `StringBufferInputStream` object that is passed into MATLAB's Xerces-based `xmlread()` function to obtain a *document node* on line 5. The root node of the *document* is then obtained as an *element node* in the variable `docRootEleNode` on line 6.

Section 5.1.2 discussed how DOM-based *element nodes* have properties such as `tagName`. Xerces exposes the `tagName` property as the method `docRootEleNode.getTagName()`, for example. With the availability of proper DOM node objects now established, Listing 6.1 makes use of the `cloneNode()` method of Xerces on line 7 to ensure a fresh deep clone of `docRootEleNode` is obtained for each *Client* as their initial `realDOM`. Line 8 then uses the `genMapFromDOM` function to obtain an `OTMap` object (based on the "map" array described in Section 5.1.2) from the *Client*'s `realDOM` object. The remainder of Algorithm 5.1 is implemented in `clientDefaultTransition.m` in a similar fashion, although no "change observer" is required since operations are introduced by the *Events Generator* rather than an event from a web browser.

#### 6.5.1.3 FSM Updates

The frequently-reused *ApplyingOps* state, which was first modeled in Figure 6.9 and then updated to include character-based *insertData* and *deleteData* operations in Figure 6.19, must now be updated to include the DOM-based versions of those operations, along with support for the new *insertNode* and *giveData* operations. By using Figure 5.2 from Section 5.3 as a guide, the DOM-based version of *ApplyingOps* was modeled using Stateflow as shown in Figure 6.27.

Algorithm 5.3 previously showed how *prevMap* and *transMap* require updating before applying an operation. The `clientApplyingOps .m` file function achieves this as shown in Listing 6.2, which is called as the first action on the default transition from the entry point inside *ApplyingOps*. The *genTransMap()* algorithm from Section 5.1.2.1 (Algorithm 5.2 for creating DOM nodes based on their description within the *FullOp*) resides in its own `.m` file of the same name and is called on line 4. Since the *FullOp* is applied based on the use of an *Events Generator*, the

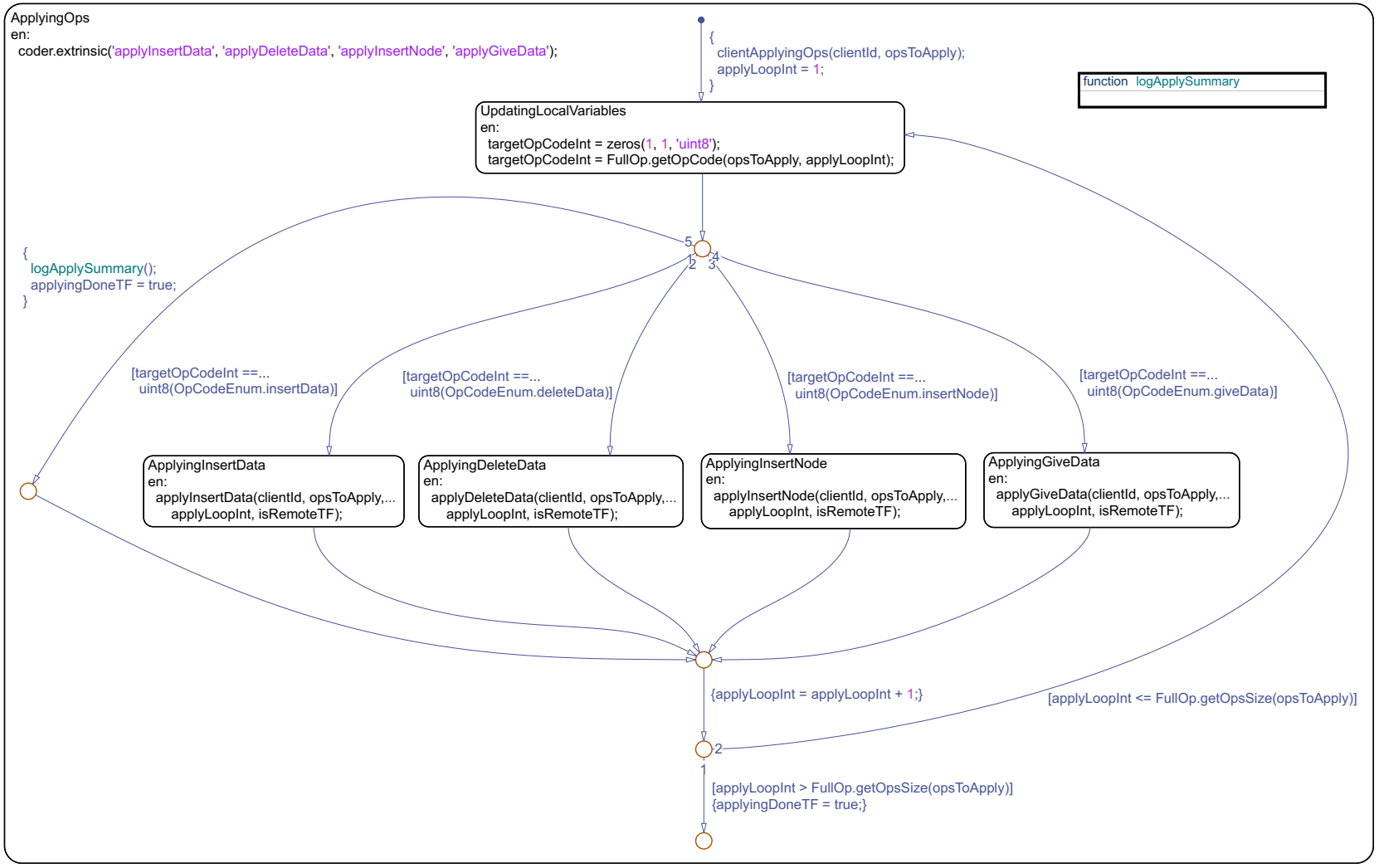



FIGURE 6.27: Client ApplyingOps state with DOM-based insertNode and giveData support.

resulting variable is named `localTransMap` to help distinguish it from a `remoteTransMap` used by the updated *ApplyingRemoteOpsWithoutACK* state described below. A boolean named `isRemoteTF`, which defaults to *false*, is used to make this distinction from the Stateflow layer, such as when calling any of the four operation types which now have dedicated functions to perform their respective DOM manipulations on the extrinsic *Client* state (all except the *identity* operation, which uses the transition on the far left side with a junction that helps with diagram line management). For example, Listing 6.3 shows how the DOM is modified by the *applyInsertNode* function to insert a new node. Before the section of code shown, the variables `parentNode`, `loc`, and `nodeToInsert` were defined based on the input *FullOp* and previously-determined transitional map, where `parentNode` is the node object into which `nodeToInsert` is to be inserted by using the `loc` position identified by the *insertNode* operation. The `getChildNodes.item()` and `insertBefore()` calls make use of the Xerces implementation of the DOM4 specification [33].

```
function clientApplyingOps(cliId, localOpsIntArray)
  global realDOM realMap prevMap localTransMap;
  prevMap{cliId} = genMapFromDOM(realDOM{cliId});
  localTransMap{cliId} = genTransMap(cliId, realMap{cliId}, localOpsIntArray);
  ...
end
```

Listing 6.2: Partial code of `clientApplyingOps.m` function using MATLAB Language.

```
...
% get the node currently at the desired location within the parent
nodeToShift = parentNode.getChildNodes.item(loc);

% insert the new node before it
parentNode.insertBefore(nodeToInsert, nodeToShift);
...
```

Listing 6.3: Partial code of `applyInsertNode.m` function using MATLAB Language.

Chapter 5 established how a *FullOp* can contain multiple *Op* objects to handle DOM changes resulting from operations such as *giveData*. The *ApplyingOps* state

must therefore loop over and apply all operations within the input *FullOp*. Figure 6.27 shows how the static method `getOpsSize()` from the *FullOp* class is used along with the `applyLoopInt` counter in order to iterate over all *Op* objects.

The *ApplyingRemoteOpsWithoutACK* state was previously described in Section 6.2.4 (Figure 6.11) as being responsible for handling a conflict by transforming a received operation against a local unacknowledged operation before applying it locally, and then transforming the previously-made local change against the received operation for re-sending to the *Controller*, as per Algorithm 4.3. However, Section 5.1.2.4 identified the sanitization behavior of real-world web applications and proposed the use of a *diff()* function to determine the local changes after applying the remote operations. Figure 6.28 shows the top half of the updated *ApplyingRemoteOpsWithoutACK* state, while Figure 6.29 shows the bottom half, where the lightning bolt symbol on each diagram indicates the continuation in the transition line connecting the two halves.

The top half of the DOM-based *ApplyingRemoteOpsWithoutACK* state shown in Figure 6.28 again begins by incrementing the timestamp value `localTS` later used when sending the updated local set of operations to the *Controller*. The *clientApplyingRemoteOpsWithoutACK* `.m` file function then simply receives the *FullOp* data from the `StoC_Msg` message for an extrinsically-accessible version of `remoteOps` and generates a corresponding *remoteTransMap*. The nested loop structure of the algorithm is coordinated using the `remoteLoopInt` and `localLoopInt` variables such that one operation is accessed from the received `remoteOps` at a time and transformed in the *TransformingForLocalApply* state against every operation within `localOps` as per Algorithm 5.6. The resulting `updatedRemoteOps` variable is then set to `opsToApply` and applied using the *ApplyingOps* state (Figure 6.27 above) in the bottom half of the *ApplyingRemoteOpsWithoutACK* state shown in Figure 6.29. The model diverges slightly from Algorithm 5.6 and shows both the *TransformingForSending* state (same content as the *TransformingForLocalApply* state) as well as a *Diff* state (which calls the *diffDOM* function of Table 6.4) connected in order to form the final `localOps` variable for sending to the *Controller* via the higher-level *SendingOpsToController* state. This way, both approaches can be compared when modeling and debugging new operations and transformations. However, the Web-Based Collaborative Platform described in Chapter 8 must rely solely on the *diff()* approach (Algorithm 5.5) in order to address the sanitization process described in Section 5.1.2.4.

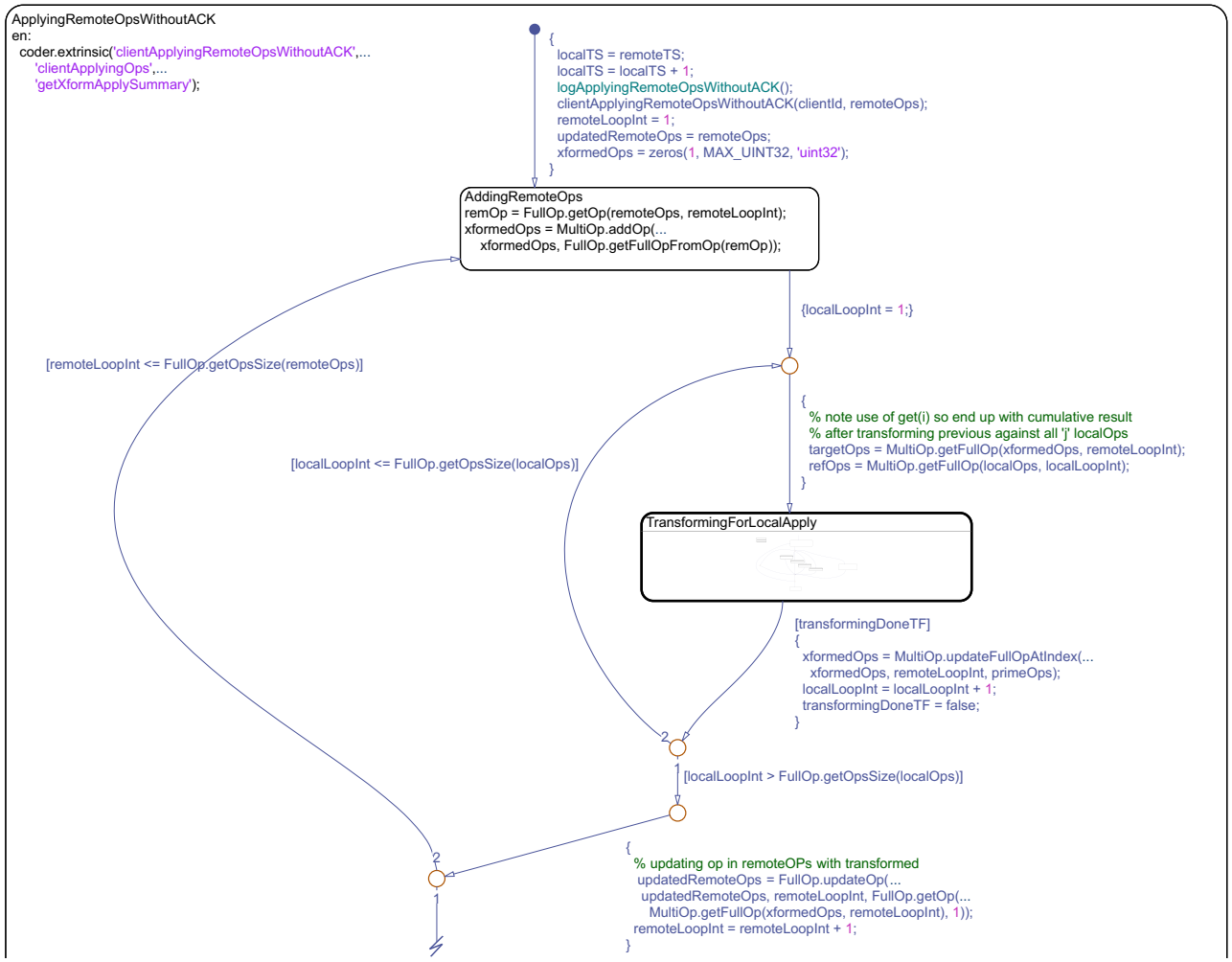



FIGURE 6.28: Top half of client ApplyingRemoteOpsWithoutACK state with DOM-based insertNode and giveData support.

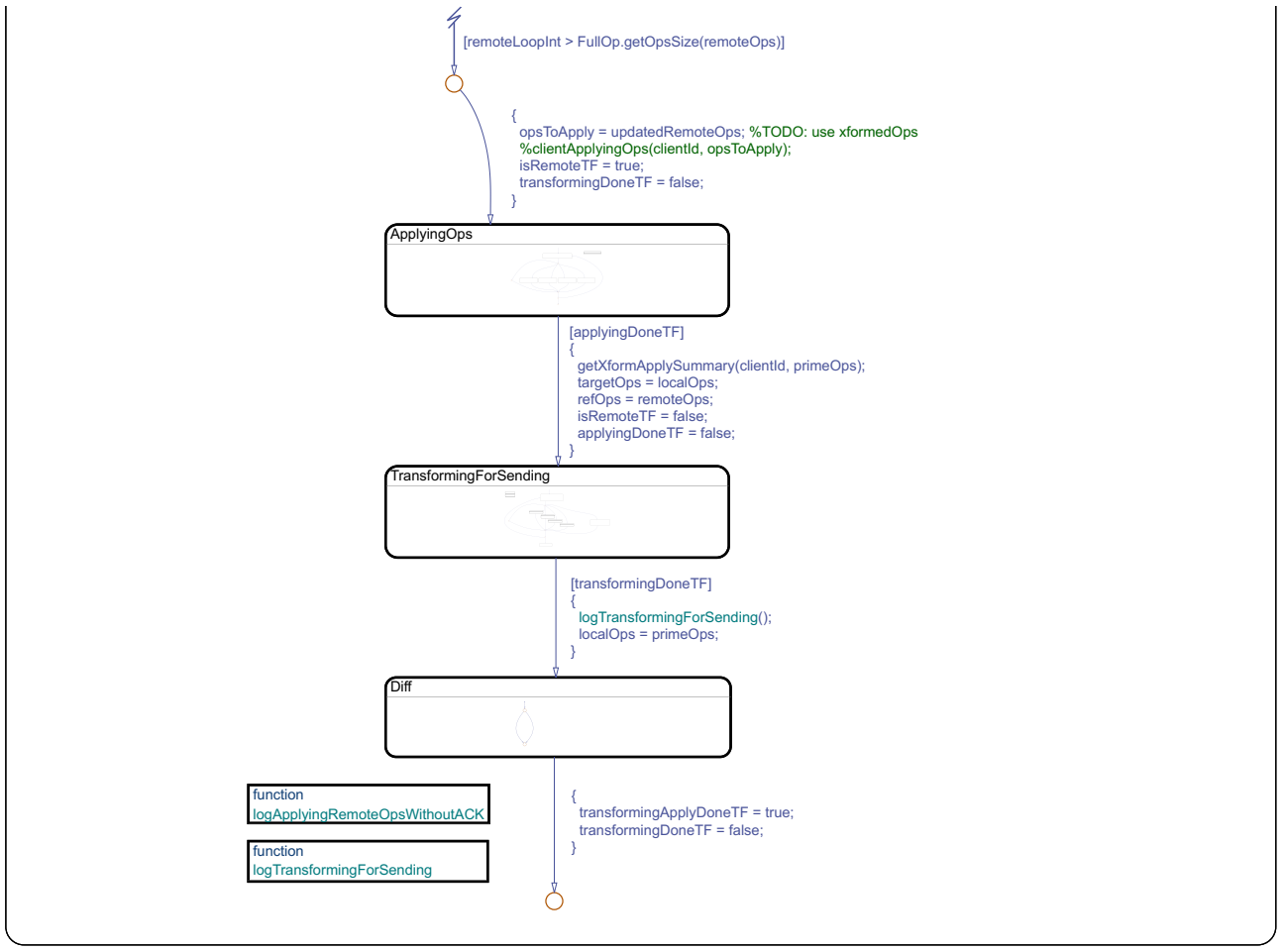



FIGURE 6.29: Bottom half of client ApplyingRemoteOpsWithoutACK state with DOM-based insertNode and giveData support.

The updated *TransformingForLocalApply* state is shown in Figure 6.30. As an example of a DOM-based transformation modeled using FSMs, Figure 6.31 shows the model corresponding to Algorithm 5.10 where a *deleteData* operation is transformed against an *insertData* operation. The *TransformingDeleteDataVsInsertData* state contains the two-junction design previously modeled in Figure 5.3 of Chapter 5, where the first junction checks if the operations are in the same parent node and immediately completes the state otherwise, thereby avoiding the additional transformation logic. Calls to static methods from the *FullOp* class are used for extracting and updating the `loc` and `opcon` values of the target *deleteData* operation. The constants `locParentIdx` (value of 0 via *Model Explorer*) and `insDelDataConPosIdx` (value of 1 via *Model Explorer*) help identify reusable array indexes within operations based on the operation structures defined in Section 5.2. Alternatively, the *TransformingForLocalApply* state (Figure 6.30) now contains the states *TransformingInsertNodeVsDeleteData* and *TransformingGiveDataVsDeleteData*, which respectively call *xFormInsertNodeVsDeleteData* and *xFormGiveDataVsDeleteData* `.m` file functions to perform the necessary attribute adjustments extrinsically (directly on the JSON content of the operations) without having to expose every attribute as a static method to the Stateflow layer. The *ApplyingRemoteOpsWithBuffer* state introduced in Section 6.4 can be updated in a similar fashion for DOM-based support by reusing the *ApplyingOps* state (Figure 6.27) and the *TransformingForLocalApply* state (Figure 6.30) detailed in this section, although this is left as future work.

### 6.5.2 Three-User Models

Figure 6.1 previously showed how a distributed system based on the CLOT algorithm was modeled in Simulink in order to enable two *Client Chart Library* instances to interact with messages from the *Events Generator* and the *Controller*. In this way, operations are sent from the *Controller* to *Client* instances in a real-time feedback control loop that ensures changes among multiple users remain synchronized even for complex hierarchical data types.

Figure 6.32 introduces a third copy of the *Client Chart Library* as a *Client* named *Carol*, along with the necessary new ports and *Stateflow Message* lines connecting *Carol* to the corresponding charts. No changes are required to the *Client Chart Library*, and the `clientId` parameter is configured to the value `3` by using the

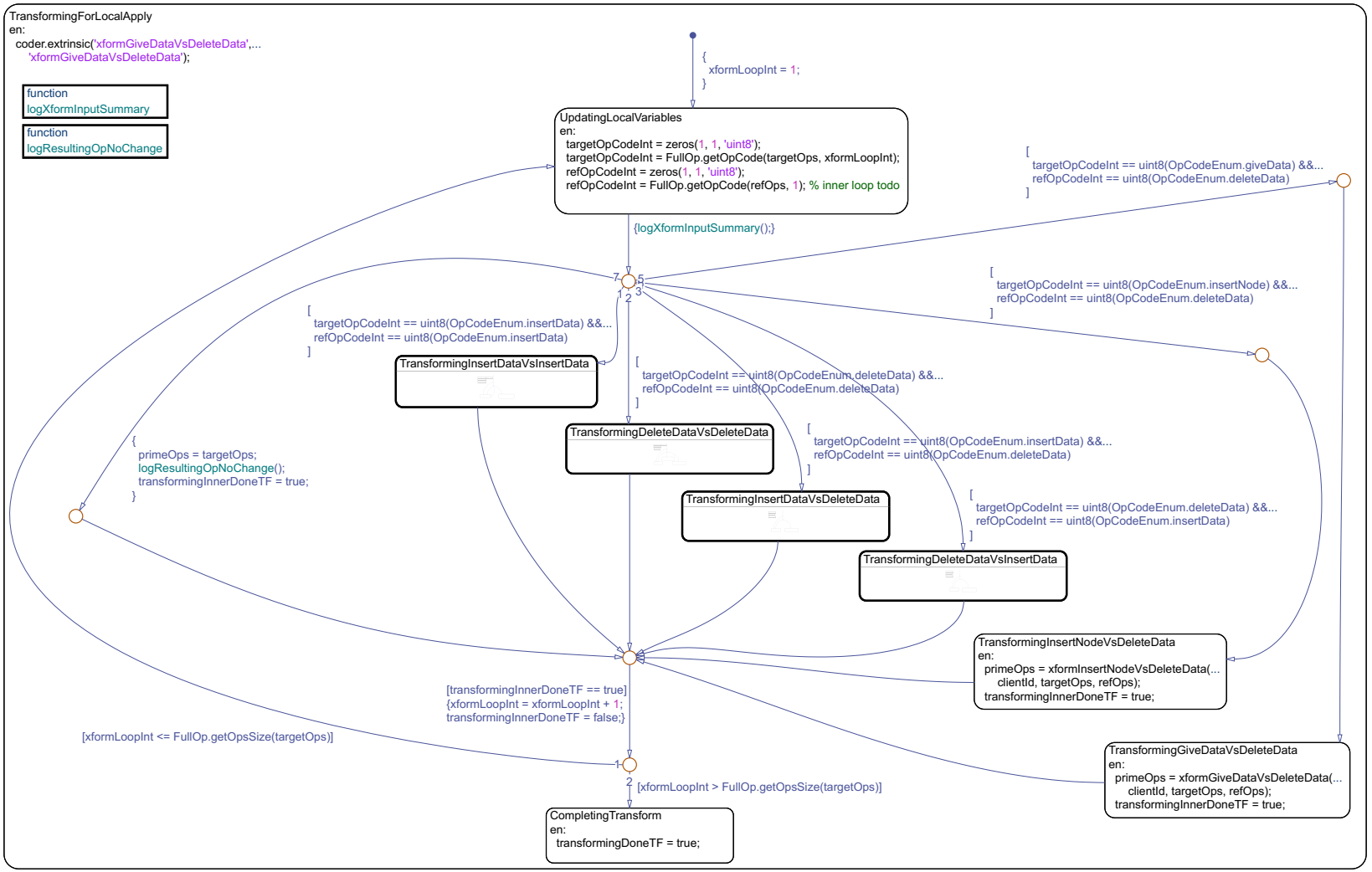



FIGURE 6.30: Client TransformingForLocalApply state with DOM-based insertNode and giveData support.

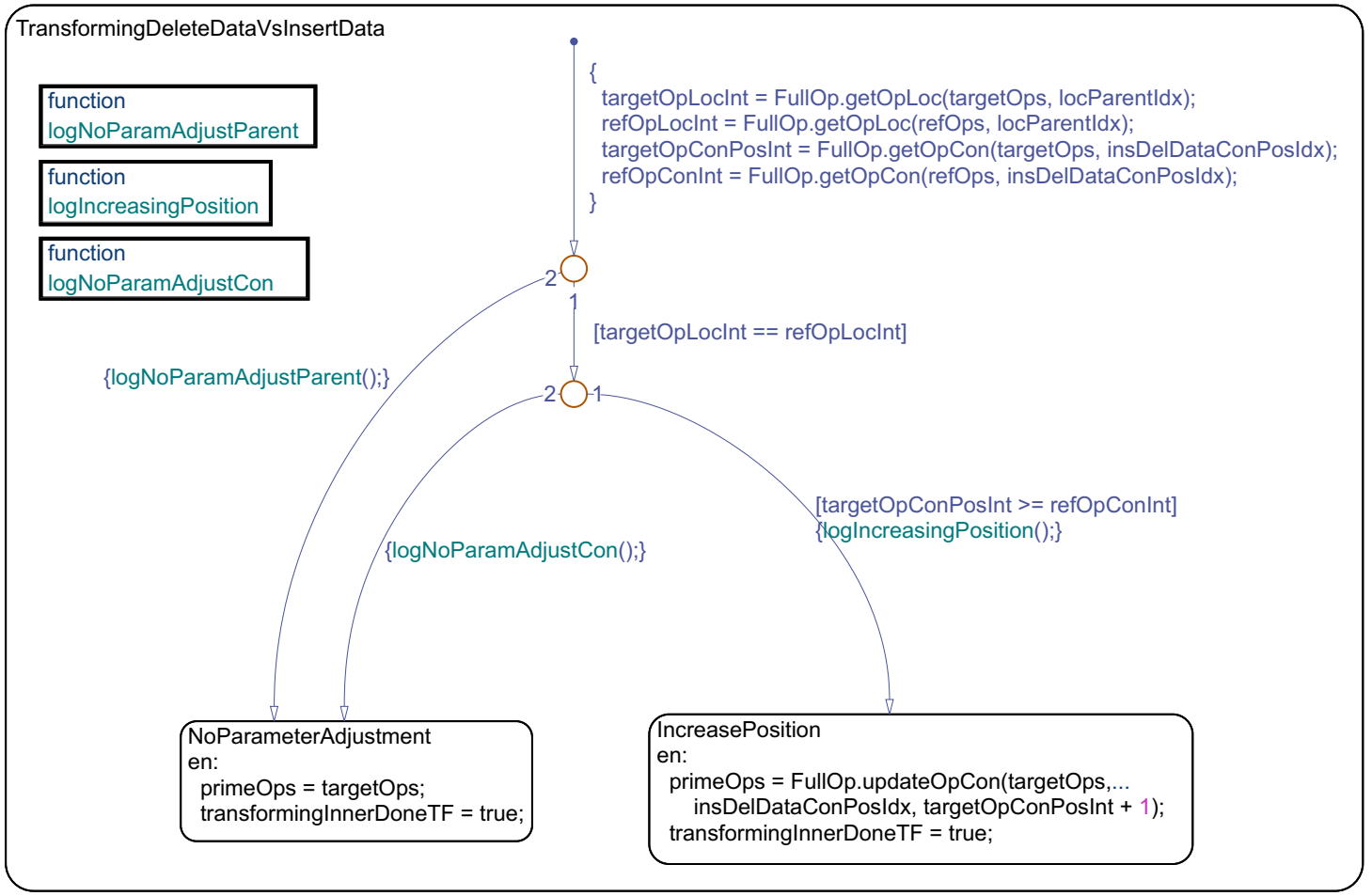



FIGURE 6.31: Client TransformingDeleteDataVsInsertData state with DOM-based support.

"Mask" interface of the block (see Section 6.2.4). Section 7.4 will explain the input and output values within the display blocks of this figure.

However, the *Events Generator* and the *Controller* charts require minor updates in order to communicate with the new *Client*. For the *Events Generator*, the introduction of a *SendingOpToCli3* state into Figure 6.3 is not difficult to achieve. The result of `FullOp.getOpClientId()` is simply compared against the new value of `3` and the pattern of *SendingOpToCli1* and *SendingOpToCli2* is repeated to identify the unique message parameters.

Similarly, the *Controller* only requires the introduction of `Cli3_in_Msg` to the design of Figure 6.14 and the corresponding state added to the *SendingACKToClient* state shown in Figure 6.15. Additional logic is required in the *SendingToRemainingClients* state for ensuring that the accepted operation is only broadcast to the correct two *Client* instances. The FSM-based approach used is shown in Figure 6.33, where the *SendingToClient3* state is reused in an attempt to minimize duplication of Stateflow Action Language statements.

By building on the key algorithms and FSM models introduced so far in this thesis, an evaluation of the real-time architecture will be explored through model execution and simulation in Chapter 7.

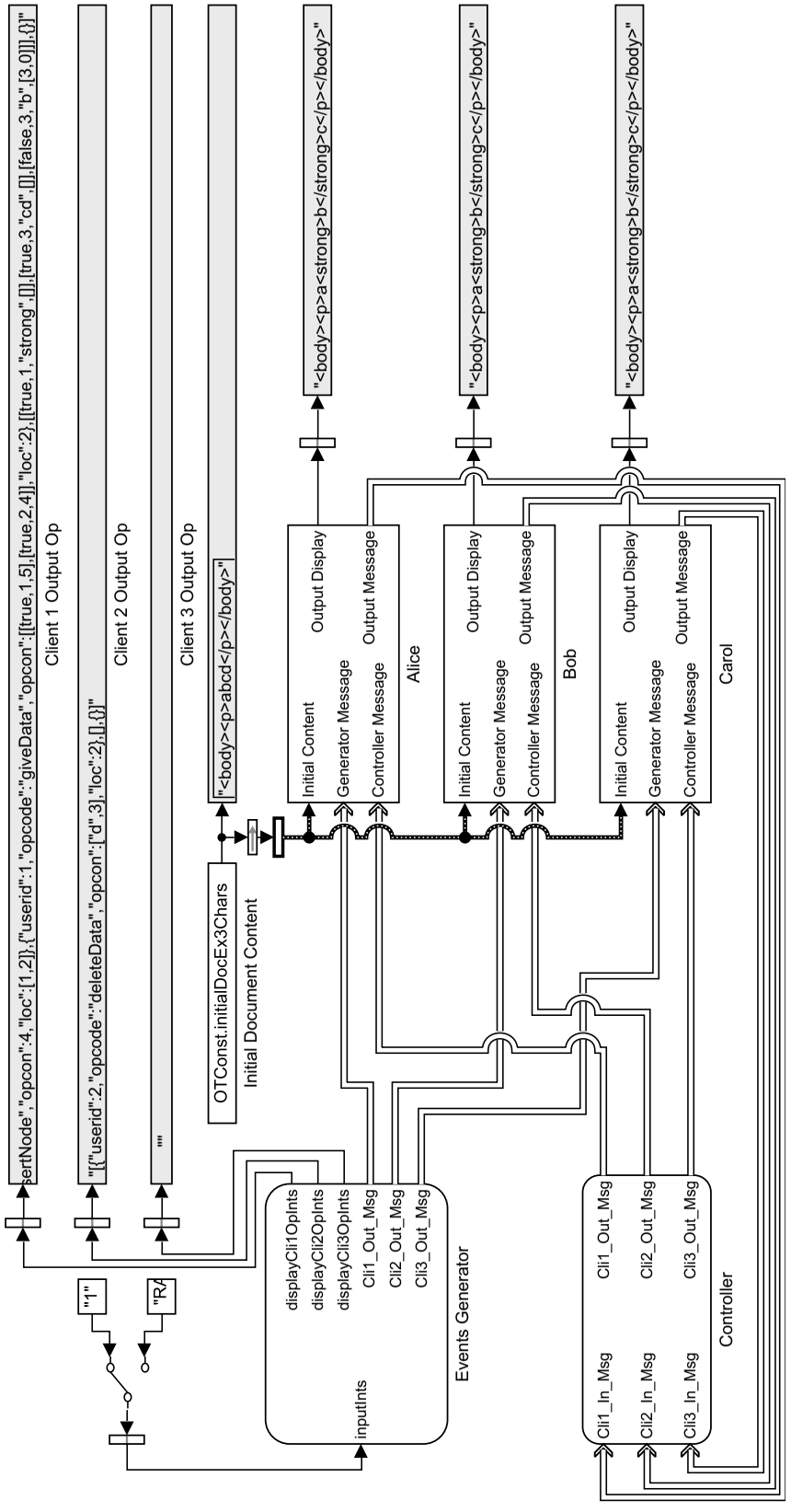



FIGURE 6.32: Three-user Simulink high-level control loop.

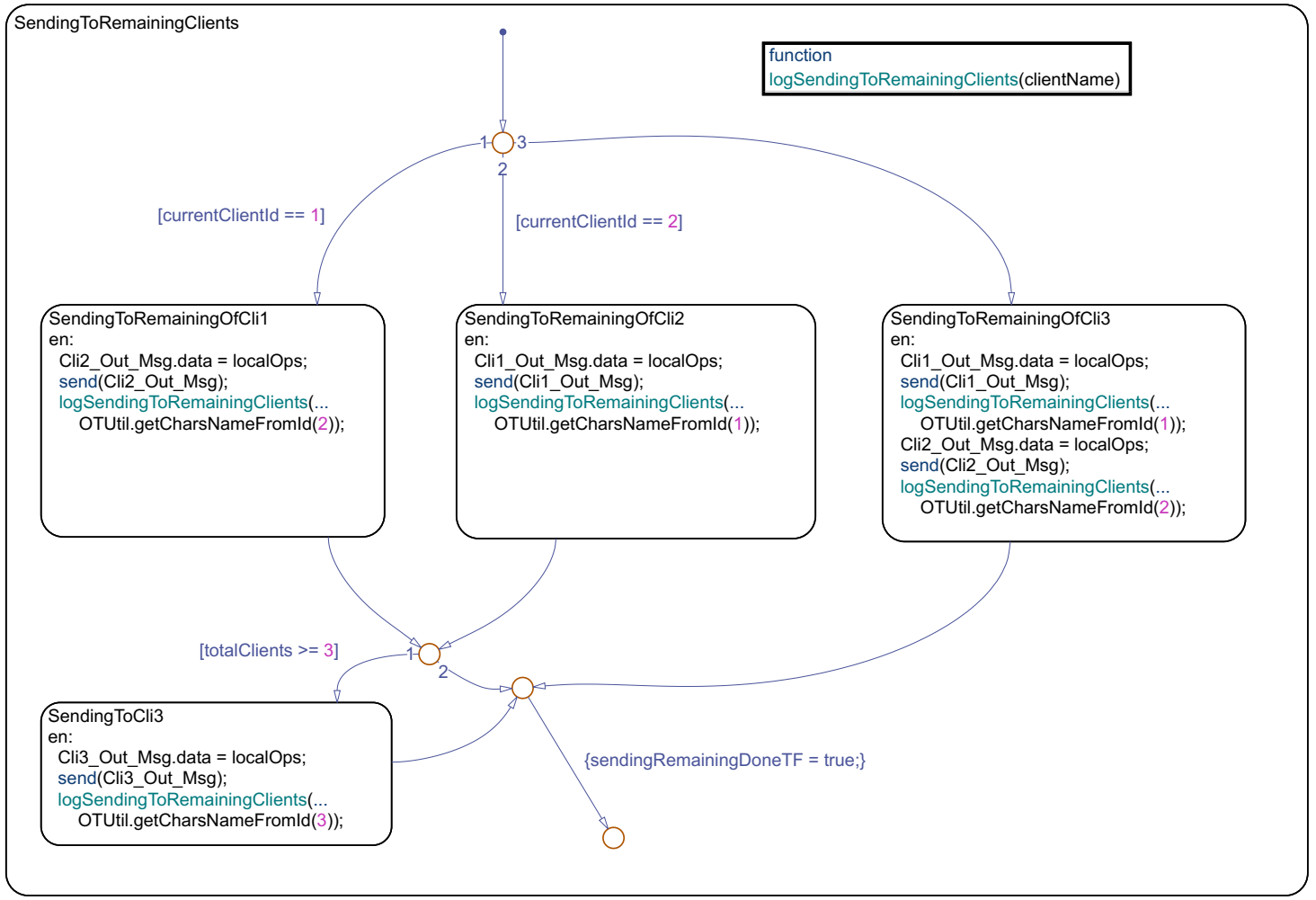



FIGURE 6.33: Three-user SendingToRemainingClients state of Controller.

# Chapter 7

# OT Simulation Results

This chapter describes the simulation results obtained by executing a variety of integration tests in order to evaluate the OT system that has been modeled in Chapter 6. By carefully observing the execution of all FSMs within the feedback-loop architecture, this chapter will show that the algorithms of the proposed "serverless" collaboration system execute correctly and enable new intention-preserving operations for concurrently supporting advanced DOM-based actions such as splitting text nodes. It will be shown how the causality, convergence, and intention preservation requirements of the OT system are met in a variety of realistic co-editing situations, thereby establishing a high level of confidence in the reliable execution of the FSM models, all of which is essential for the development of the Web-Based Collaborative Platform and the collaborative web applications it enables.

## 7.1 Basic OT FSM

Section 6.2 presented a simplified model of the CLOT algorithm by focusing on *identity* operations moving through a two-user system. More specifically, Section 6.2.3 described how the execution of test "1" causes Alice to receive the *FullOp* of `[{"userid":1,"opcode":"i"},{}]` from the *Events Generator*. With the *Events Generator* omitted as an actor, the main sequence of events defining this test is summarized in Figure 7.1 and simply has the previously-mentioned *FullOp* move through the system from user Alice to user Bob. While this first test is designed to be as simple as possible and does not include a conflict, it reaches

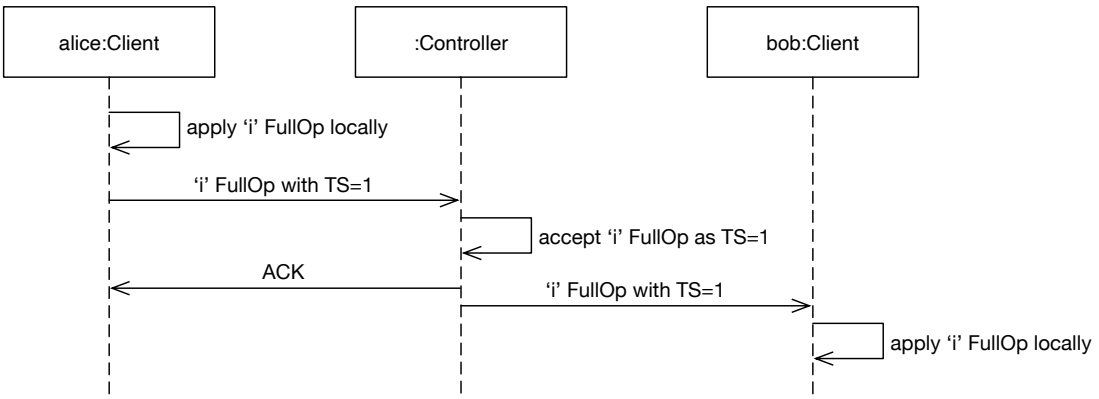



FIGURE 7.1: Message sequence chart for a basic identity operation scenario (test "1").

a large number of significant system states, and is therefore useful for establishing the basic flow of events in the system.

In the *OTGenerator* class, the test is defined as per Listing 7.1, with the comments revealing what the constants from the *OTConst* class resolve to. The `initDoc` character array defined on line 1 is used to populate the Simulink *Initial Document Content* block (see Section 6.5.1.2). Line 2 defines the *FullOp* that will be introduced into the system by the *Events Generator*. The syntax used to define `ops` allows for easy appending of additional *FullOp* character arrays by duplicating the line, thereby forming a *MultiOp* (this will be shown in later examples). The `expectedResult` variable is optionally included in order to perform a final test assertion to determine if the result of the execution is "correct".

```
1 initDoc = 'OTConst.initialDocEx1Chars'; % abc
2 ops = [ops, OTConst.identityEx1Chars];  % [{"userid":1,"opcode":"i"},{}]
3 expectedResult = 'abc';
```

LISTING 7.1: MATLAB Language code executed by Events Generator to initiate a basic identity operation scenario (test "1").

Figure 6.4 of the *Events Generator* provided an example of how output strings were assembled and sent to the Simulink console using the `disp()` function during the execution of the FSM-based system. The resulting console output therefore updates as the states and transitions of the FSM are visited, providing valuable insight into the functioning of the system. The execution of the `disp()` statement in Figure 6.4 results with line 1 of Listing 7.2. Lines 2 and 3 show that both Alice and Bob have correctly initialized with the same initial document content of "abc" (`initDoc` of Listing 7.1 correctly reached the `inputVblDocInts` of each *Client*

instance as per Section 6.2.4) and are both in the *Synced* state. Line 4 confirms that the *Controller* has also initialized by reaching the *Listening* state. On Line 5, Alice is processing the `Local_Change` message from the *Events Generator*, which contains a *FullOp* that Alice must first apply locally. The *Events Generator* can be considered to be instructing Alice's *Client* to perform an operation that updates the local document, much like a user typing a letter on a keyboard causes a WYSIWYG editor to update the local document when the letter is displayed.

```
1  SENDING [{"userid":1,"opcode":"i"},{}] TO: Alice
2  Alice in Synced with: abc
3  Bob in Synced with: abc
4  Controller in Listening
5  Alice typed and received Local_Change with [{"userid":1,"opcode":"i"},{}] while
       in Synced
6  Alice in ApplyingLocalOps, Alice.localTS is now 0
7  Alice sent CtoS_Msg with [{"userid":1,"opcode":"i"},{"TS":1}] via
       SendingOpsToController
8  Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"i"},{"TS
       ":1}] while in Listening
9  Controller determined TS from Alice is new and reached PersistingNew,
       Controller.localTS is now 1
10 Alice in AwaitingACK
11 Controller sent StoC_ACK to Alice via SendingACKToClient
12 Alice in Synced with: abc
13 Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"i"},{"TS":1}]
       to Bob via SendingToRemainingClients
14 Bob received StoC_Msg with [{"userid":1,"opcode":"i"},{"TS":1}] while in Synced
15 Bob in ApplyingRemoteOps, Bob.localTS is now 1
16 Controller in Listening
17 Bob in Synced with: abc
18 CONSISTENT AND CORRECT!
```

Listing 7.2: Snippet of console output during FSM execution of a basic identity operation scenario (test "1").

Alice applies the *FullOp* on line 6 via the *ApplyingLocalOps* state, which has no effect on the local document content due to its *identity* nature. Stateflow visually highlights active states using a blue border, and Figure 7.2 shows how the simulation user interface displays Alice as executing the *ApplyingLocalOps* state within the *Running* state of the *Client* model (the state machine was previously shown in Figure 6.6). Alice then sends the *FullOp* to the *Controller* on line 7 with an incremented timestamp (TS) value of 1. While Alice then transitions to the

*AwaitingACK* state (line 10), the *Controller* consumes the *FullOp*, recognizes the TS as being new, and accepts the *FullOp* by transitioning to its *PersistingNew* state (lines 8-9). The *SendingACKToClient* state of the *Controller* then causes a message to be sent to Alice such that Alice recognizes her previously-sent operation as being acknowledged and returns to the *Synced* state with the local document remaining as "abc" (line 12). The *Controller* then sends Alice's *FullOp* to the other session participant, Bob, on line 13, before returning to the *Listening* state (line 16). Bob's receiving of the *FullOp* causes him to transition out of his *Synced* state and into the *ApplyingRemoteOps* state, where his local document is updated to include the operation (no change in this case). Bob then returns to the *Synced* state, where it is revealed that his local document still contains "abc" (line 17).

Upon verifying the final document content of both *Client* instances, the *Events Generator* displays that both have converged on the same final content as per the *consistency model* of Sun et al. [4]) (`CONSISTENT`), and that the final content matches the content asserted by the test's `expectedResult` value (`CORRECT`). The `expectedResult` value from the test's definition in *OTGenerator* therefore also accounts for the intention preservation requirements of the OT system so that, as much as possible, both users are satisfied with the final result (Section 2.2.1.4). By optionally not providing the `expectedResult` value, some tests, including randomly-generated tests, will not check for correctness and are considered to "pass" based on consistency alone. Of course, the *identity* operation used in the tests of this section (and Section 7.2) will never modify the document content. Simply checking the final document content of both users at quiescence to determine that these tests "pass" is therefore of little value. Instead, the careful analysis of the full console log is required to ensure the desired behavior of the FSMs is observed, as was done above.

To observe a simple conflict scenario between two *identity* operations, the next test will follow the sequence defined by Listing 7.3, where Bob produces his own *identity* operation in addition to Alice's operation of the previous example.

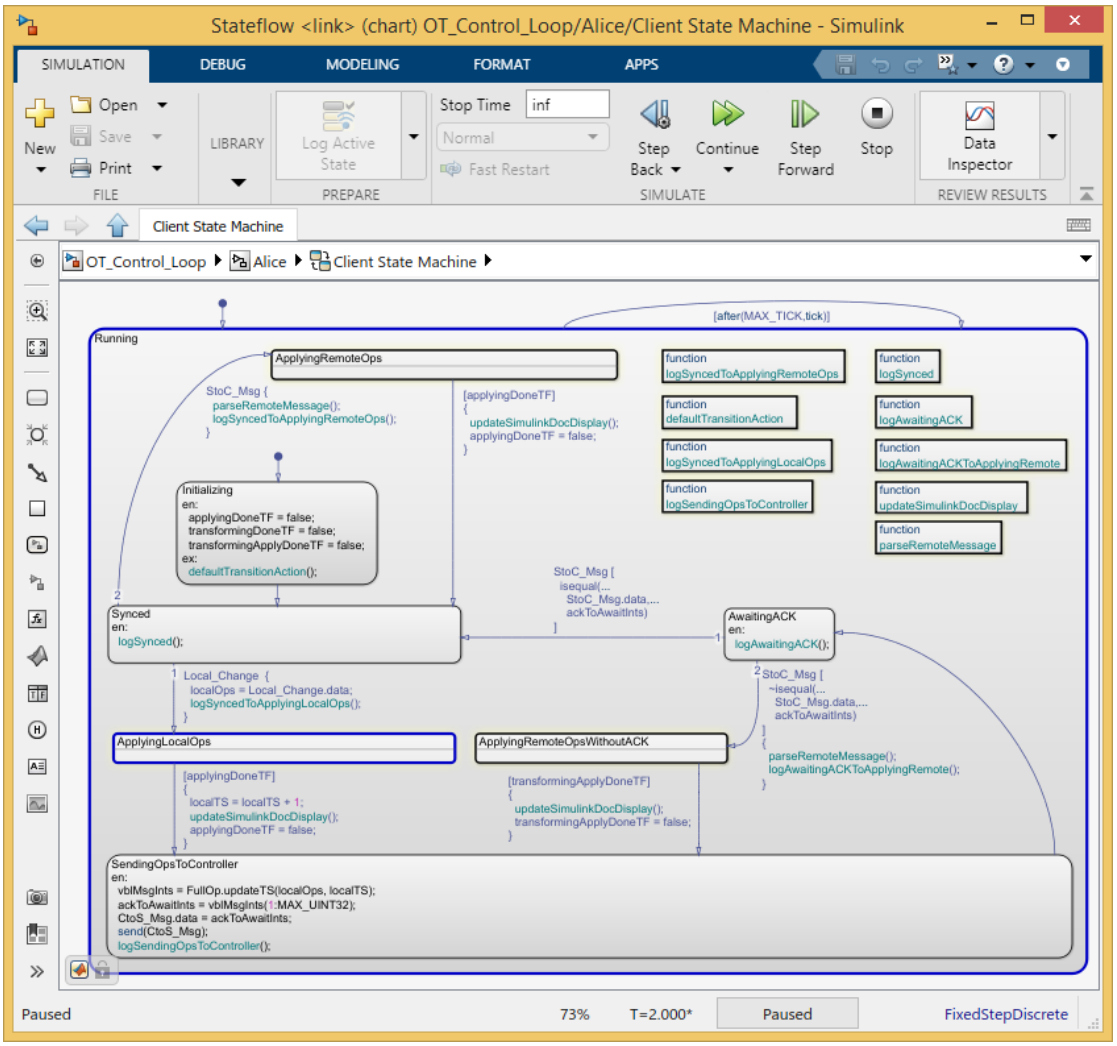



FIGURE 7.2: Alice in ApplyingLocalOps state during Stateflow execution of a basic identity operation scenario (test "1").

```
1 initDoc = 'OTConst.initialDocEx1Chars'; % abc
2 ops = [ops, OTConst.identityEx1Chars];  % [{"userid":1,"opcode":"i"},{}]
3 ops = [ops, OTConst.identityEx2Chars];  % [{"userid":2,"opcode":"i"},{}]
4 expectedResult = 'abc';
```

LISTING 7.3: MATLAB Language code executed by Events Generator to initiate a basic transformation scenario between two identity operations (test "2").

Listing B.1 of Appendix B features the complete console log of the execution of test "2". Listing 7.4 duplicates the more significant section (line 22 to line 32) of this larger listing, but the lines leading up to it are briefly discussed first. The corresponding sequence of events is summarized in Figure 7.3, where `'i'` is used for brevity to refer to an `'i' FullOp`.

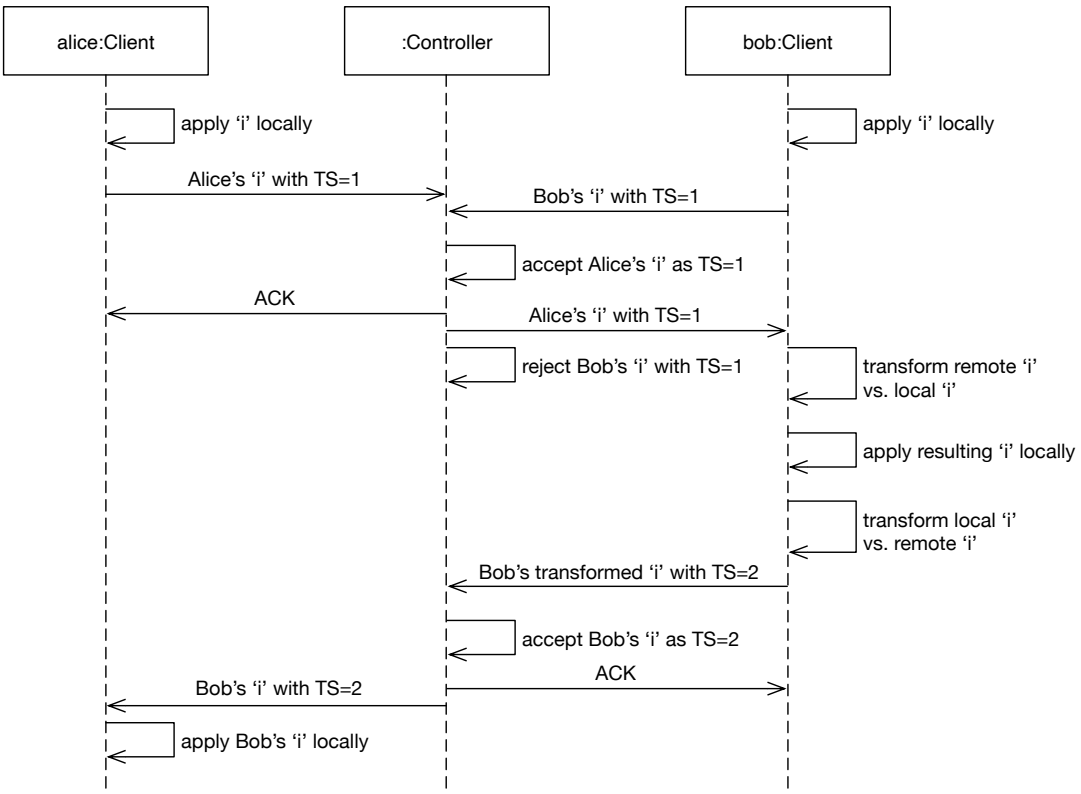



FIGURE 7.3: Message sequence chart for a basic transformation scenario between two identity operations (test "2").

On line 6, Alice's `localTS` begins with the value of 2 due to the test suite from which this particular test execution was extracted, and the value can be seen incremented to 3 in the *FullOp* that Alice sends to the *Controller* on line 10. The message from the *Events Generator* is received by Bob on line 8 and applied in the following line as Alice did in the previous example. However, Alice's *FullOp* (with a timestamp of 3) is accepted by the *Controller* first (line 12), so Alice receives her acknowledgement and returns to the Synced state on line 16. Bob, however, sent his *FullOp* on line 14, also with a timestamp of 3, only to receive Alice's *FullOp* on line 19. Bob therefore enters the *ApplyingRemoteOpsWithoutACK* state.

On line 22, which can now be referenced from Listing 7.4, the *Controller* has received Bob's *FullOp*, and, upon comparing timestamp values, rejected it for reusing the timestamp of 3 (line 23). Meanwhile, Bob is carrying out the actions of the *ApplyingRemoteOpsWithoutACK* state, where he must transform the received identity operation from Alice against his local identity operation and apply the (negligible) result to his local document, as shown on line 25. Bob then transforms his local operation against Alice's operation (line 26) and sends the resulting *FullOp* (still containing an *identity* operation) to the Controller with the updated

timestamp of 4 (line 27). This *FullOp* is accepted by the Controller (line 29) and acknowledged so that Bob can return to *Synced* (line 32). The remainder of the example in Listing B.1 shows how the *Controller* also sends Bob's updated *FullOp* to Alice, which Alice applies via her *ApplyingRemoteOps* state before also returning to the *Synced* state.

```
22 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"i"},{"TS":3}]
       while in Listening
23 Controller determined TS from Bob is old (REJECTED!)
24 Controller in Listening
25 Bob transforms remoteOps [{"userid":1,"opcode":"i"},{"TS":3}] against localOps
      [{"userid":2,"opcode":"i"},{}] to get [{"userid":1,"opcode":"i"},{"TS":3}]
      and applies it, Bob's document is now abc
26 Bob performed the transformation with reversed parameter sequence and obtained
      [{"userid":2,"opcode":"i"},{}]
27 Bob sent CtoS_Msg with [{"userid":2,"opcode":"i"},{"TS":4}] via
      SendingOpsToController
28 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"i"},{"TS":4}]
       while in Listening
29 Controller determined TS from Bob is new and reached PersistingNew, Controller.
      localTS is now 4
30 Bob in AwaitingACK
31 Controller sent StoC_ACK to Bob via SendingACKToClient
32 Bob in Synced with: abc
```

LISTING 7.4: Snippet of console output during FSM execution of a basic transformation scenario between two identity operations (test "2").

The Stateflow screenshot of Figure 7.4 shows Bob executing the *ApplyingRemoteOpsWithoutACK* state machine (based on Figure 6.11), which contains the nested *TransformingForLocalApply* state. The execution of the system is easy to follow by keeping Bob's state machines open along side those of Alice (Figure 7.2 above), as well as those of the *Events Generator*, *Controller*, and the high-level Simulink model (which updates display block values as per Figure 6.1). The two tests of this section, along with several other variations on them, were used to confirm that the basic models of Section 6.2 execute correctly, thereby ensuring that the upcoming sections have a solid foundation to build upon.

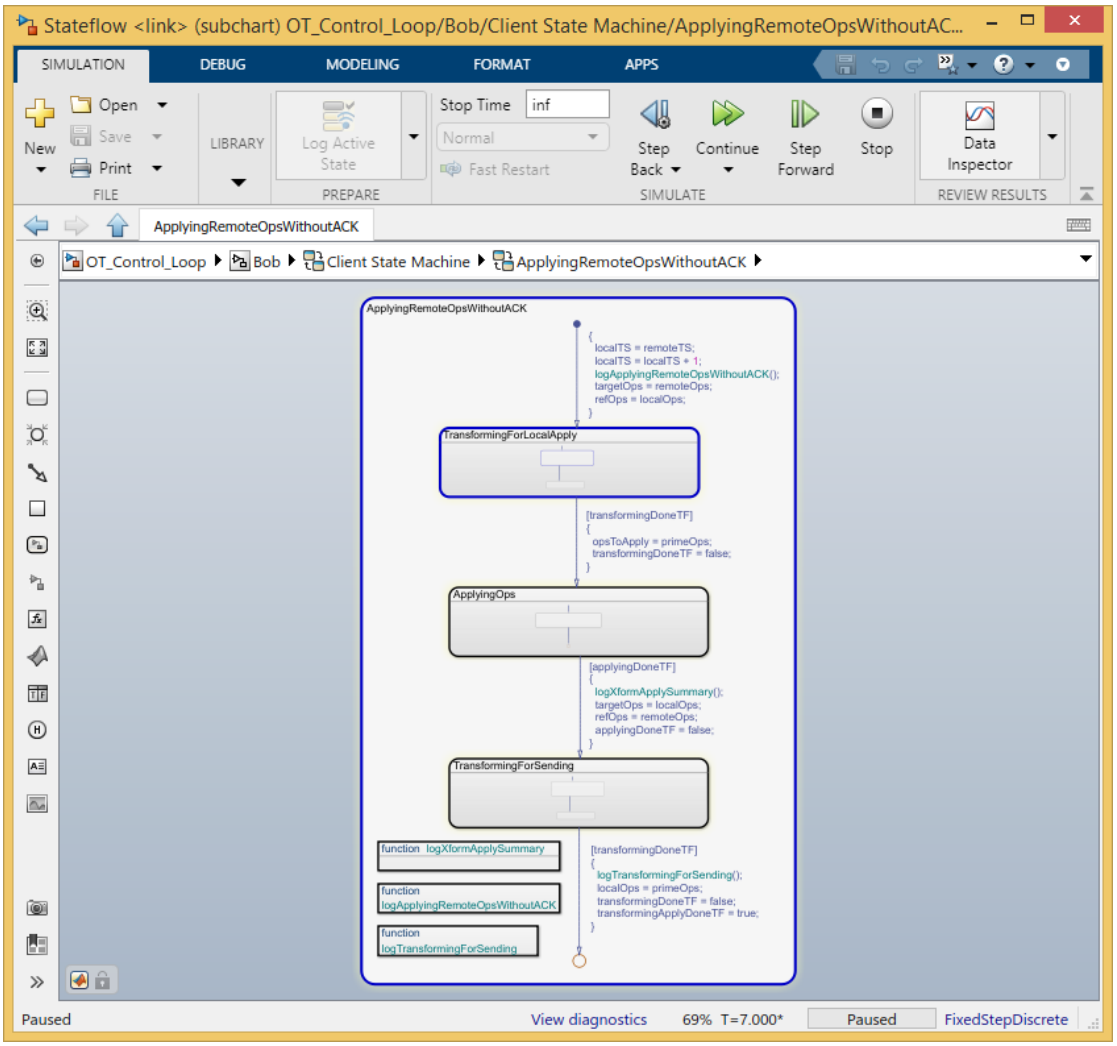



FIGURE 7.4: Bob executing the nested TransformingForLocalApply state within ApplyingRemoteOpsWithoutACK during Stateflow execution of a basic transformation scenario between two identity operations (test "2").

## 7.2 Basic OT FSM with Buffer

The basic flow of messages established in Section 7.1 will now be extended to evaluate the buffer-related states modeled in Section 6.3. While still relying on *identity* operations, Test "3" focuses on reaching three of the new states within the right side of the final top level *Client* diagram shown in Figure 6.17, namely *ApplyingBufferedLocalOps*, *AwaitingWithBuffer* and *CreatingLocalOpFromBuffer*. The simplest pattern of operations required to evaluate the buffer functionality involves the same user performing two operations immediately after one another, thereby causing the second operation to become buffered before the first is acknowledged, as illustrated in Figure 7.5. The *Events Generator* invokes this sequence for Alice with the code of Listing 7.5.

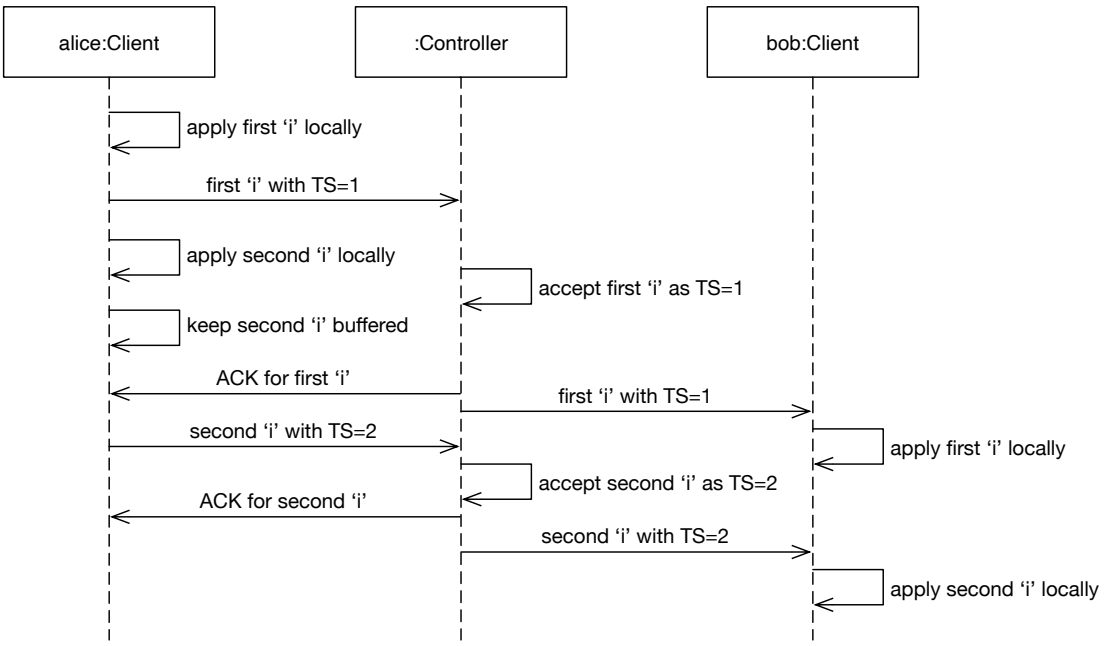



FIGURE 7.5: Message sequence chart for a basic buffer scenario with two identity operations (test “3”).

```
1 initDoc = 'OTConst.initialDocEx1Chars'; % abc
2 ops = [ops, OTConst.identityEx1Chars];  % [{"userid":1,"opcode":"i"},{}]
3 ops = [ops, OTConst.identityEx1Chars];  % [{"userid":1,"opcode":"i"},{}]
4 expectedResult = 'abc';
```

LISTING 7.5: MATLAB Language code executed by Events Generator to initiate a basic buffer scenario with two identity operations (test “3”).

Listing B.2 of Appendix B begins in a fashion similar to Listing 7.2 above where Alice is performing an *identity* operation that results with her reaching the *AwaitingACK* state. However, the second *identity* operation performed by Alice can be observed to have an effect in Listing 7.6, which extracts lines 12 to 23 of the full Listing B.2. By processing a `Local_Change` message while in the *AwaitingACK* state on line 13, Alice transitions to the *ApplyingBufferedLocalOps* state and stores this new *FullOp* in her buffer. Alice then transitions to the *AwaitingWithBuffer* state on line 19, where she can process the *StoC_ACK* message the *Controller* previously sent on line 12. Alice then reaches the *CreatingLocalOpFromBuffer* state on line 21 to prepare the message from the buffer for sending to the *Controller*, including incrementing its timestamp. The *SendingOpsToController* state is then reused on line 23 to perform the sending of this second *FullOp* containing an *identity* operation. The remainder of Listing B.2 sees Alice return to *AwaitingACK* to await this previously-buffered operation, which Alice receives, allowing her to

return to the *Synced* state. In this test, Bob is simply receiving and applying both of Alice's operations via his *ApplyingRemoteOps* state.

```
12 Controller sent StoC_ACK to Alice via SendingACKToClient
13 Alice typed and received signal Local_Change with buffer op [{"userid":1,"
       opcode":"i"},{}] while in AwaitingACK
14 Alice in ApplyingBufferedLocalOps with buffered op [{"userid":1,"opcode":"i
       "},{}]
15 Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"i"},{"TS":5}]
       to Bob via SendingToRemainingClients
16 Bob received StoC_Msg with [{"userid":1,"opcode":"i"},{"TS":5}] while in Synced
17 Bob in ApplyingRemoteOps, Bob.localTS is now 5
18 Controller in Listening
19 Alice in AwaitingWithBuffer with buffer size 1
20 Alice received signal StoC_ACK while in AwaitingWithBuffer
21 Alice in CreatingLocalOpFromBuffer, Alice.localTS is now 6
22 Bob in Synced with: abc
23 Alice sent CtoS_Msg with [{"userid":1,"opcode":"i"},{"TS":6}] via
       SendingOpsToController
```

LISTING 7.6: Snippet of console output during FSM execution of a basic buffer scenario with two identity operations (test "3").

In order to reach the *ApplyingRemoteOpsWithBuffer* state of Figure 6.17, the *Events Generator* sends the sequence of operations shown in Listing 7.7 as test "4". That is, Alice sends her operation to the *Controller*, after which Bob performs two local operations, where the first is rejected by the *Controller*, leaving Bob to handle a conflict scenario for his two local changes. Figure 7.6 summarizes the expected sequence of events in the system.

```
1 initDoc = 'OTConst.initialDocEx1Chars'; % abc
2 ops = [ops, OTConst.identityEx1Chars];  % [{"userid":1,"opcode":"i"},{}]
3 ops = [ops, OTConst.identityEx2Chars];  % [{"userid":2,"opcode":"i"},{}]
4 ops = [ops, OTConst.identityEx2Chars];  % [{"userid":2,"opcode":"i"},{}]
5 expectedResult = 'abc';
```

LISTING 7.7: MATLAB Language code executed by Events Generator to initiate a basic buffer scenario with three identity operations (test "4").

Listing 7.8 of Appendix B shows the full log of executing the simulation of this test, but the noteworthy section of lines 26 to 37 is included as Listing 7.8 below. Before line 26, Bob added his second operation to a local buffer (similar to Alice in

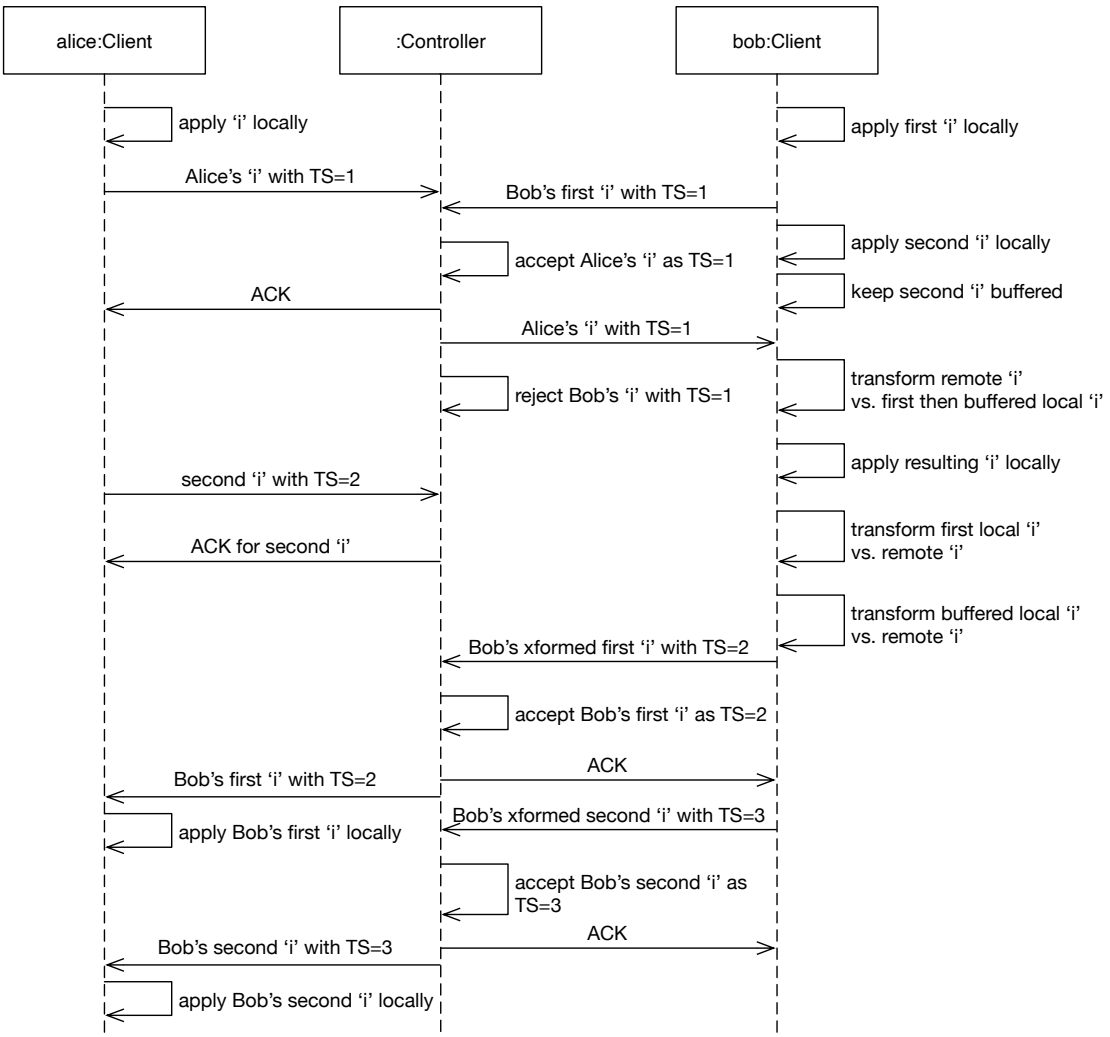



FIGURE 7.6: Message sequence chart for a basic buffer scenario with three identity operations (test "4").

the previous example). However, while in the *AwaitingWithBuffer* state, Bob received Alice's *FullOp* instead of acknowledgement of his own first operation. This causes Bob to transition to the *AwaitingRemoteOpsWithBuffer* state, which must execute Algorithm 4.5, as seen on lines 28 and 29. Figure 7.7 shows a screenshot of the state machine being executed in Stateflow, with Bob processing the inner *ApplyingOps* state at the time (Figure 6.23 and Figure 6.24 previously showed a version of *AwaitingRemoteOpsWithBuffer* in greater detail by separating it into two parts). After applying Alice's operation, the resulting version of Bob's first operation is sent to the *Controller* on line 31, and Bob returns to the AwaitingWithBuffer state since his buffer is not empty. By processing the *Controller*'s acknowledgement of his operation on line 36, Bob is able to send his buffered operation via *CreatingLocalOpFromBuffer*. The remainder of the test is similar to

the sequence of the previous example, with Bob returning to the *AwaitingACK* state before the *Synced* state and Alice directly applying both of his operations.

```
26 Bob in AwaitingWithBuffer with buffer size 1
27 Bob received signal StoC_Msg while in AwaitingWithBuffer (CONFLICT RESOLUTION
       WITH BUFFER!)
28 Bob in ApplyingRemoteOpsWithBuffer, remoteOps is [{"userid":1,"opcode":"i"},{"
       TS":7}], Bob.localTS is now 8
29     That is, Bob transforms remoteOps [{"userid":1,"opcode":"i"},{"TS":7}]
       against localOps [{"userid":2,"opcode":"i"},{}] to get [{"userid":1,"opcode
       ":"i"},{"TS":7}] and applied it, Bob's document is now abc
30 Bob performed the transformation with reversed parameter sequence and obtained
       [{"userid":2,"opcode":"i"},{}] as the updated ACK that awaiting
31 Bob sent CtoS_Msg with [{"userid":2,"opcode":"i"},{"TS":8}] via
       SendingOpsToController
32 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"i"},{"TS":8}]
        while in Listening
33 Controller determined TS from Bob is new and reached PersistingNew, Controller.
       localTS is now 8
34 Bob in AwaitingWithBuffer with buffer size 1
35 Controller sent StoC_ACK to Bob via SendingACKToClient
36 Bob received signal StoC_ACK while in AwaitingWithBuffer
37 Bob in CreatingLocalOpFromBuffer, Bob.localTS is now 9
```

LISTING 7.8: Snippet of console output during FSM execution of a basic buffer scenario with three identity operations (test "4").

## 7.3 Buffered OT FSM with Insert and Delete

To move beyond simple *identity* examples into more meaningful *apply* and *transformation*-related functionality, Section 6.4 modeled the inclusion of the character-based versions of the *insertData* and *deleteData* operations.

Based on examples from seminal OT research [3][61], Figure 2.6 of Section 2.2.1.2 illustrated a transformation where Alice and Bob begin with the document content of "xyz", into which Alice inserts the letter "a" at position 1 and Bob inserts the letter "b" at position 2, and both reach the final document of "xabyz". The code of Listing 7.9 is used to set up the *Events Generator* to simulate this example via the modeled CLOT-based system. The code references the reusable *FullOp* definitions from the *OTConst* class given in Listing 7.10. The expected sequence

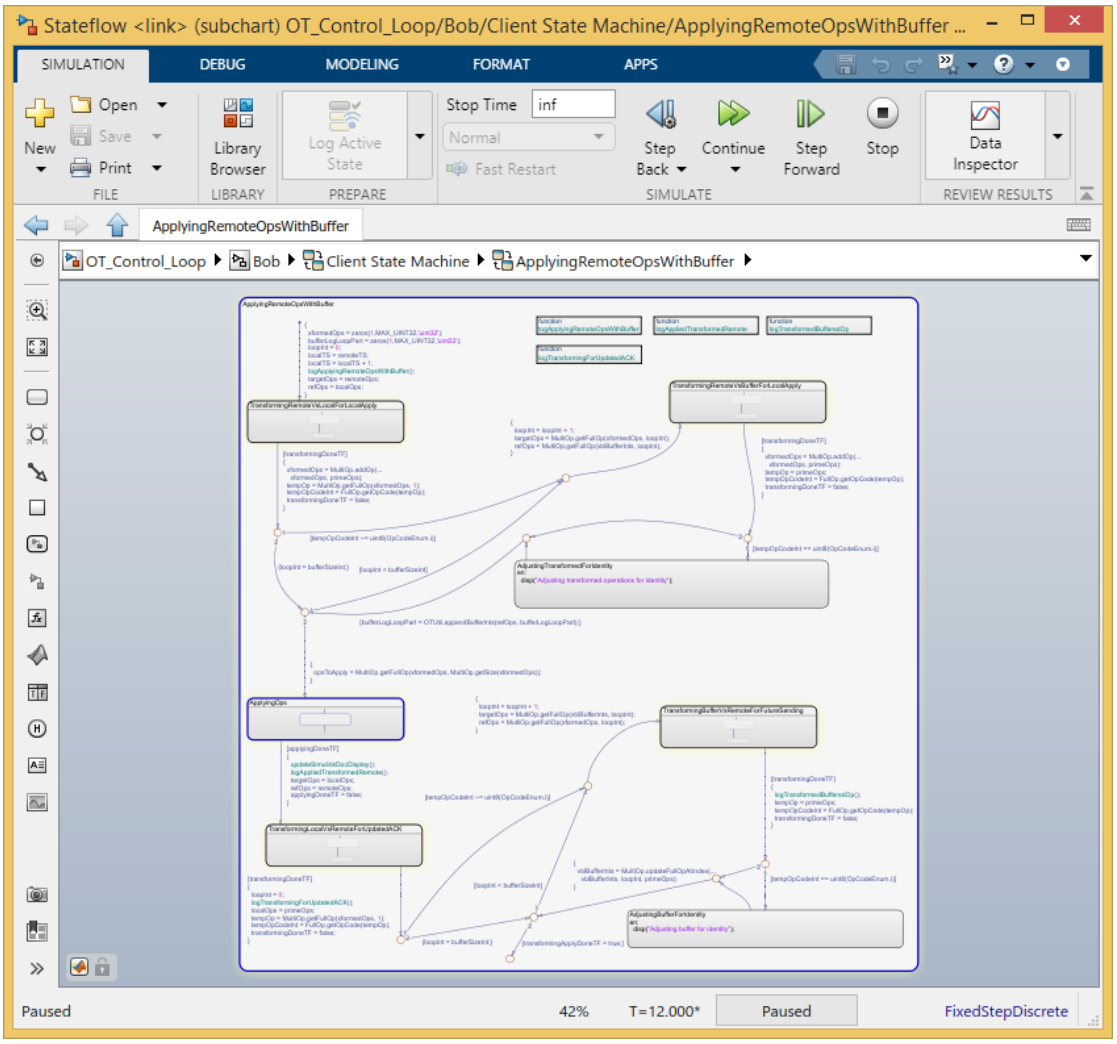



FIGURE 7.7: Bob in the ApplyingRemoteOpsWithBuffer state during Stateflow execution of a basic buffer scenario with three identity operations (test "4").

of events is therefore similar to test "2" (Figure 7.3 of Section 7.1) above, but with an *insertData FullOp* used instead of `'i'`.

```
1 initDoc = 'OTConst.initialDocEx2Chars'; % xyz
2 ops = [ops, OTConst.insDataEx1Chars];   % Alice insertData("a", 1)
3 ops = [ops, OTConst.insDataEx5Chars];   % Bob insertData("b", 2)
4 expectedResult = 'xaybz';
```

LISTING 7.9: MATLAB Language code executed by Events Generator to initiate a transformation scenario between character-based insertData operations (test "5").

```
1 insDataEx1Chars = ’[{"userid":1,"opcode":"insertData","opcon":["a",1]},{}]’];
2 insDataEx5Chars = ’[{"userid":2,"opcode":"insertData","opcon":["b",2]},{}]’];
```

LISTING 7.10: MATLAB Language code defining FullOp definitions used by a transformation scenario between character-based insertData operations (test “5”).

Listing B.4 of Appendix B contains the full execution log of the test “5” simulation. The listing shows how both Alice and Bob apply the *FullOp* instances from the *Event Generator* locally, where Alice obtains “xayz” (line 10) while Bob obtains “xybz” (line 15). However, since Alice’s *FullOp* was accepted first, Bob must enter the *ApplyingRemoteOpsWithoutACK* state to resolve the conflict, as shown in the snippet of Listing 7.11. Line 24 and 25 show how Bob prepares to transform Alice’s operation as the *target* and his own local operation as the *reference*. Line 29 summarizes the result on the transformation and its effect after being applied to Bob’s document (the desired “xaybz”). Line 30 and 31 show how the target and reference operations have been reversed to obtain Bob’s updated *FullOp* for his local change (the *Controller* can be seen rejecting Bob’s first attempt in lines 26-29).

```
24 Bob XFORM: Target op is {"userid":1,"opcode":"insertData","opcon":["a",1]},
       reference is {"userid":2,"opcode":"insertData","opcon":["b",2]}
25 Bob XFORM: insertData vs. insertData with no parameter adjustment
26 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"insertData","
       opcon":["b",2]},{"TS":17}] while in Listening
27 Controller determined TS from Bob is old (REJECTED!)
28 Controller in Listening
29 Bob transforms remoteOps [{"userid":1,"opcode":"insertData","opcon":["a",1]},{"
       TS":17}] against localOps [{"userid":2,"opcode":"insertData","opcon":["b
       ",2]},{}] to get [{"userid":1,"opcode":"insertData","opcon":["a",1]},{"TS
       ":17}] and applies it, Bob’s document is now xaybz
30 Bob XFORM: Target op is {"userid":2,"opcode":"insertData","opcon":["b",2]},
       reference is {"userid":1,"opcode":"insertData","opcon":["a",1]}
31 Bob XFORM: insertData vs. insertData where increase position of target
       insertion
32 Bob performed the transformation with reversed parameter sequence and obtained
       [{"userid":2,"opcode":"insertData","opcon":["b",3]},{}]
33 Bob sent CtoS_Msg with [{"userid":2,"opcode":"insertData","opcon":["b",3]},{"TS
       ":18}] via SendingOpsToController
```

LISTING 7.11: Snippet of console output during FSM execution of a basic transformation.

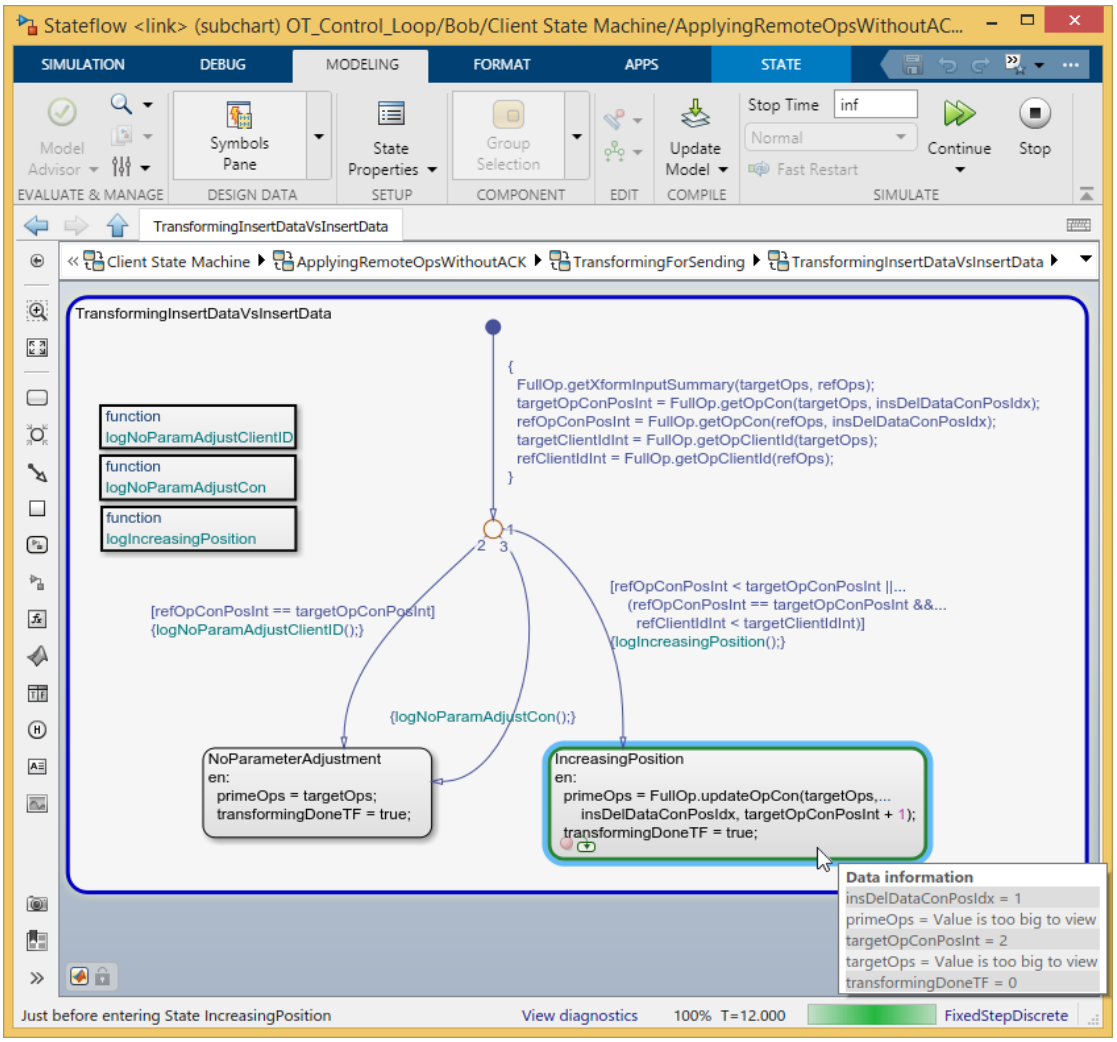



FIGURE 7.8: Bob in the ApplyingRemoteOpsWithBuffer state during Stateflow execution of a transformation scenario between character-based insertData operations (test "5").

Figure 7.8 shows the Simulink step-based debugging utilities being used to observe the value of position parameter `targetOpConPosInt` as `2` before entering the *TransformingInsertDataVsInsertData* state (the state machine was previously shown in Figure 6.21). Bob's updated *FullOp*, containing the transformed position parameter value of `3` after the execution of the state, is then sent to the *Controller* on line 33. The remainder of Listing B.4 shows how Alice goes on to receive Bob's operation, which she can apply directly, resulting with both clients reaching the *Synced* state with the consistent and correct result of "xaybz".

In order to test the system more thoroughly and observe more advanced buffer and transformation scenarios, the manual switch block of Figure 6.1 that selects the mode of operation of the *Events Generator* was set to the random position.

The algorithm of the *getRandomOps()* function of *OTGenerator* picks a random number of "slots" to which a *FullOp* can be assigned (up to four in the example shown below). As many parameters of the *FullOp* as possible are then randomized, including the *Client* that is to receive the *FullOp*, the operation type and the various letters and positions that can make up an operation. The *Events Generator* checks that all *Client* instances ended up in a consistent state after the random multi-slot scenario has finished executing. The simulation stops if any of the users obtained a different final document state, leaving the log available for later analysis and debugging. By leaving the *Event Generator* to execute for several hours, a high degree of confidence was obtained that the FSM-based algorithms are robust and free of transformation errors.

The sequence shown in Listing 7.12 and using the *FullOp* constants of Listing 7.13 was recreated from a randomized run as an example of a test generated by the *getRandomOps()* function. Test "6" therefore resulted with one local *FullOp* for Bob followed by three for Alice. The sequence of events evolves on that of test "4" (Figure 7.6 of Section 7.2) but goes beyond it by requiring Alice to buffer an additional operation and, of course, using *insertData* and *deleteData* operations requiring proper transformations and document content updates.

```
1 initDoc = 'OTConst.initialDocEx7Chars'; % tymlykekpymtkiksqsisnuamdmwmqagffnng
2 ops = [ops, OTConst.delDataEx8Chars];   % Bob deleteData("_", 33)
3 ops = [ops, OTConst.insDataEx20Chars];  % Alice insertData("g", 24)
4 ops = [ops, OTConst.delDataEx9Chars];   % Alice deleteData("_", 15)
5 ops = [ops, OTConst.delDataEx10Chars];  % Alice deleteData("_", 13)
6 expectedResult = 'tymlykekpymtkkqsisnuamgdmwmqagffng';
```

Listing 7.12: MATLAB Language code executed by Events Generator to initiate a randomized buffer scenario with character-based insertData and deleteData operations (test "6").

```
1 delDataEx8Chars = ['[{"userid":2,"opcode":"deleteData","opcon":["_",33]},{}]']
2 insDataEx20Chars = ['[{"userid":1,"opcode":"insertData","opcon":["g",24]},{}]']
3 delDataEx9Chars = ['[{"userid":1,"opcode":"deleteData","opcon":["_",15]},{}]']
4 delDataEx10Chars = ['[{"userid":1,"opcode":"deleteData","opcon":["_",13]},{}]']
```

Listing 7.13: MATLAB Language code defining FullOp definitions used by a randomized buffer scenario with character-based insertData and deleteData operations (test "6").

Listing 7.14 corresponds to lines 40 to 52 of Listing B.5 in Appendix B. Since Bob's *FullOp* reached the *Controller* before any of Alice's, Alice ended up in the *ApplyingRemoteOpWithBuffer* state. The listing shows how Alice proceeds to apply Bob's updated *FullOp* with the transformation summary on line 41. Alice also updates the local and buffered operations before reattempting to send the local operation as a *FullOp* to the *Controller* (line 51). The remainder of the log of Listing B.5 shows how Alice sends each operation from the buffer after the previous operation is acknowledged (line 59 and line 72), and how all three operations reach Bob to be applied directly via *ApplyingRemoteOps*. In the end, both Alice and Bob remain at rest in the *Synced* state with the same document content, and the content matches the expected content defined via Listing 7.12, thereby indicating that the test has passed.

```
40 ...
41     That is, Alice transforms remoteOps [{"userid":2,"opcode":"deleteData","
       opcon":["_",33]},{"TS":40}] against localOps [{"userid":1,"opcode":"
       insertData","opcon":["g",24]},{}] and buffer [{"userid":1,"opcode":"
       deleteData","opcon":["_",15]},{}] and buffer [{"userid":1,"opcode":"
       deleteData","opcon":["_",13]},{}] to get [{"userid":2,"opcode":"deleteData
       ","opcon":["_",32]},{"TS":40}] and applied it, Alice's document is now
       tymlykekpymtkkqsisnuamgdmwmqagffng
42 Alice XFORM: Target op is {"userid":1,"opcode":"insertData","opcon":["g",24]},
       reference is {"userid":2,"opcode":"deleteData","opcon":["_",33]}
43 Alice XFORM: insertData vs. deleteData with no parameter adjustment (incoming
       Ins is before local Del)
44 Alice performed the transformation with reversed parameter sequence and
       obtained [{"userid":1,"opcode":"insertData","opcon":["g",24]},{}] as the
       updated ACK that awaiting
45 Alice XFORM: Target op is {"userid":1,"opcode":"deleteData","opcon":["_",15]},
       reference is {"userid":2,"opcode":"deleteData","opcon":["_",34]}
46 Alice XFORM: deleteData vs. deleteData with no parameter adjustment
47     That is, Alice buffer[1] is now [{"userid":1,"opcode":"deleteData","opcon
       ":["_",15]},{}]
48 Alice XFORM: Target op is {"userid":1,"opcode":"deleteData","opcon":["_",13]},
       reference is {"userid":2,"opcode":"deleteData","opcon":["_",33]}
49 Alice XFORM: deleteData vs. deleteData with no parameter adjustment
50     That is, Alice buffer[2] is now [{"userid":1,"opcode":"deleteData","opcon
       ":["_",13]},{}]
51 Alice sent CtoS_Msg with [{"userid":1,"opcode":"insertData","opcon":["g
       ",24]},{"TS":41}] via SendingOpsToController
52 Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"insertData
       ","opcon":["g",24]},{"TS":41}] while in Listening
```

Listing 7.14: Snippet of console output during FSM execution of a randomized buffer scenario with character-based insertData and deleteData operations (test "6").

## 7.4 HTML DOM-Based OT FSM with Three Users

The previous sections of this chapter have shown how the *Events Generator* was used to evaluate the basic components of the CLOT integration algorithm, including the support for buffered operations and transformations of character-based insertions and deletions. One of the design objectives of the CLOT algorithm,

however, is to allow for OT systems with more advanced operations that act on the HTML DOM, thereby enabling the development of new forms of web-based collaboration tools.

Section 2.3 introduced approaches from existing research for addressing the challenges posed by OT-based DOM synchronization systems. Among other operations proposed in Section 5.2, Section 5.2.6 presented the challenging *giveData* operation syntax, which goes beyond the existing research approaches of Chapter 2 by supporting the splitting of DOM-based text nodes while preserving the intentions of participating users. Section 6.5 then developed the corresponding models required to simulate such DOM-based synchronization functionality.

To evaluate the DOM synchronization functionality of the system, test "7" creates a DOM-based OT conflict scenario. The scenario assumes that Alice and Bob both begin with a DOM of Listing 5.18 consisting of a *body* node containing a single *p* node. The *p* node has a single text node containing the string "abcd". Alice selects the letter "b" in her rich-text editor and presses the "bold" button in the editor's toolbar. Meanwhile, Bob deletes the letter "d" from the text in his editor. Assuming Alice's "bold" operation reaches the *Controller* first, Bob will receive Alice's operation instead of having his own delete operation acknowledged. Bob will then transform the incoming "bold" operation so that he can apply it locally. After this transformation of Alice's remote operation, Bob will transform his local delete operation to account for this change and resend his local delete with a new timestamp as a second attempt to get it accepted by the *Controller*. Since Alice will not have performed any new operations, Bob's operation will be accepted. The updated delete operation will be sent to Alice, who can apply it to her DOM directly via the *ApplyingRemoteOps* state. The sequence of events is therefore similar to the pattern of test "2" (Figure 7.3), which has been updated as Figure 7.3 below. A third user, Carol, is also participating in the session but is simply receiving operations sent by the *Controller* and applying them directly.

Listing 7.15 shows the code that enables the *Events Generator* to distribute the required *FullOp* instances for this test. Alice performs the *giveData* operation from the second example of Section 5.2.6, while Bob performs a DOM-based *deleteData* operation whose position in the text node is after the split introduced by Alice. The `initDoc` value defined on line 1 contains the DOM previously shown in Listing 5.18, while `aliceOpChars` on line 2 corresponds to the *FullOp* consisting of Listing 5.20 and Listing 5.21 combined. The "bold" operation of Figure 7.9 is

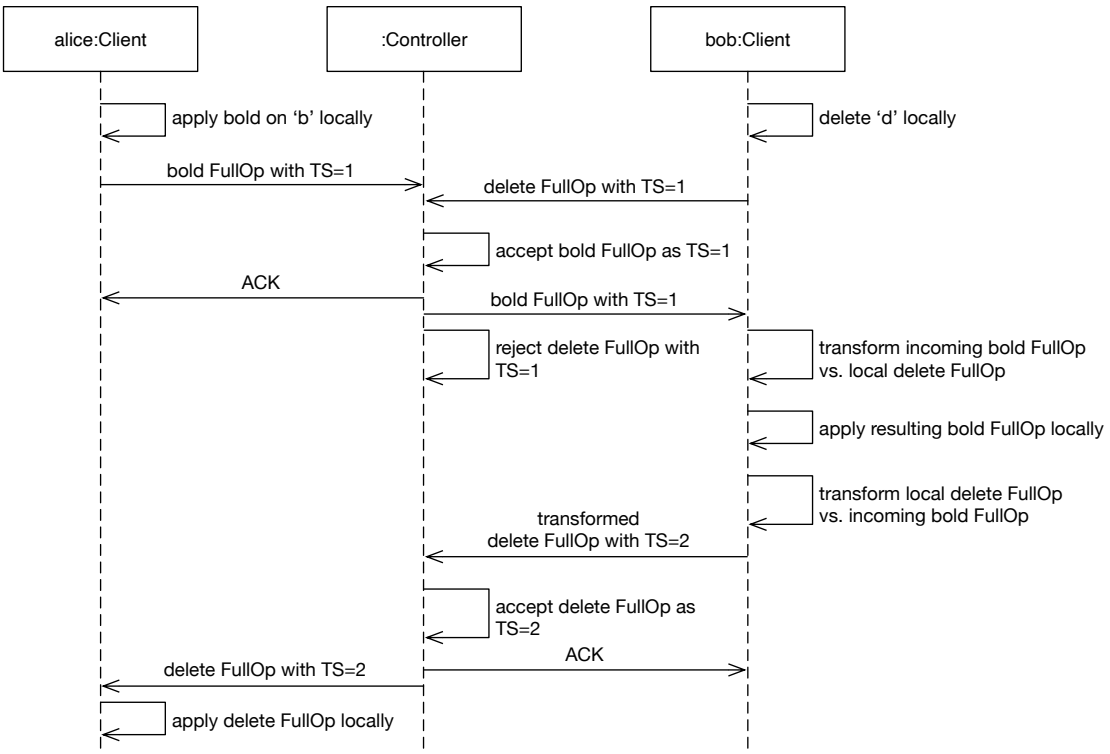



FIGURE 7.9: Message sequence chart for a DOM node splitting scenario (test "7").

therefore encoded using a *FullOp* consisting of *insertNode* and *giveData* operations defined from line 2 to line 20. The *toChar()* static method of the *FullOp* class is used to assemble a text version of the *FullOp* by calling the *toChar()* methods of objects like *Op* and *Payload* introduced in Section 5.1.1 and defined in MATLAB as described in Section 6.5.1.1 and Appendix A. Bob's delete operation is achieved on lines 21 to 25 by using the DOM-based version of *deleteData* that includes the *loc* attribute (Section 5.2.5), shown here as `OpLoc`.

```
1  initDoc = 'OTConst.initialDocEx3Chars'; % <body><p>abcd</p></body>
2  aliceOpChars = FullOp(Op(uint8(ClientNameEnum.Alice),...
3                           char(OpCodeEnum.insertNode),...
4                           OpCon(3),...
5                           OpLoc(1, 1)),...
6                      Op(uint8(ClientNameEnum.Alice),...
7                           char(OpCodeEnum.insertNode),...
8                           OpCon(4),...
9                           OpLoc(1, 2)),...
10                     Op(uint8(ClientNameEnum.Alice),...
11                          char(OpCodeEnum.giveData),...
12                          OpCon(true, 1, 5, true, 2, 4),...
13                          OpLoc(2)),...
14                     Payload(true, 1, "strong",...
15                          PayloadAux()),...
16                     Payload(true, 3, "cd",...
17                          PayloadAux()),...
18                     Payload(false, 3, "b",...
19                          PayloadAux(3, 0))...
20                     ).toChar();
21 bobOpChars = FullOp(Op(uint8(ClientNameEnum.Bob),...
22                          char(OpCodeEnum.deleteData),...
23                          OpCon("d", 3),...
24                          OpLoc(2)),...
25                     ).toChar();
26 ops = [ops, aliceOpChars];
27 ops = [ops, bobOpChars];
28 expectedResult = '<body><p>a<strong>b</strong>c</p></body>';
```

Listing 7.15: MATLAB Language code executed by Events Generator to initiate a DOM node splitting scenario (test "7").

Listing B.6 of Appendix B shows the full execution log of test "7". The first line shows the result of the *toChar()* static method when applied to the *FullOp* of Alice defined in Listing 7.15 (a single-line version of Listing 5.20 and Listing 5.21 combined). Lines 2-4 confirm that all three *Client* instances have initialized their `realMap` with the desired DOM based on the character array arriving on the *Initial Content* port of the *Client* and originating from `initDoc` above.

Lines 8 to 20 show how Alice loops through the various parts of the *FullOp* received from the *Events Generator* to apply them using the state machine previously shown in Figure 6.27. The `prevMap` and `realMap` are logged on separate lines, while the `transMap` is also denoted as `localTM` or `remoteTM` to indicate if it is a

*localTransMap* or a *remoteTransMap* as per Section 6.5.1.3. The *FullOp* is then sent to the *Controller* and accepted on line 23 before being broadcast. Carol can apply the *FullOp* directly to reach the same state as Alice with the bold "b" (line 55). However, Bob had received and applied the *FullOp* from the *Events Generator* containing the *deleteData* operation (line 30). Bob therefore enters the *ApplyingRemoteOpsWithoutACK* state (line 44) upon receiving Alice's *FullOp*.

The snippet in Listing 7.16 corresponds to lines 76 to 79 of the full log of Listing B.6). In order for Bob to transform each operation from Alice's *FullOp* (*insertNode*, *insertNode*, *giveData*) against his local *deleteData* operation, Bob executes the state machine of Figure 6.28. Bob therefore reaches states *TransformingInsertNodeVsDeleteData* and *TransformingGiveDataVsDeleteData* previously seen in Figure 6.30 to determine that the resulting *FullOp* remains unchanged and can be applied directly, as seen on line 77. Line 78 shows that the *Diff* state determined the local *deleteData* operation required adjusting of both the node which the operation affects (`loc` was changed from a `2` to a `4` to reach the last text node) as well as the character position within that new text node (`opcon` was changed from a `3` to a `1` for the letter "d" within the new text node containing simply "cd"). As the full example in Listing B.6 shows, this *FullOp* is then able to be applied by Alice and Carol successfully by accounting for the new *strong* node and reaching the correct new text node, resulting with a converged and intention-preserving DOM of `<body><p>a<strong>b</strong>c</p></body>` for all three users. Figure 6.32 shows this final result in the display blocks along the right side (as well as the inputs to Alice and Bob of Listing 7.15 along the top), as observed from the three-user control loop executing in Simulink.

```
1 ...
2     That is, Bob transforms remoteOps {"userid":1,"opcode":"insertNode","opcon
      ":3,"loc":[1,1]}{"userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]}{"
      userid":1,"opcode":"giveData","opcon":[[true,1,5],[true,2,4]],"loc":2}
      against localOps {"userid":2,"opcode":"deleteData","opcon":["d",3],"loc":2}
       to get {"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]}{"userid
      ":1,"opcode":"insertNode","opcon":4,"loc":[1,2]}{"userid":1,"opcode":"
      giveData","opcon":[[true,1,5],[true,2,4]],"loc":2} and applies it, Bob.
      realMap is now [body, p, "a", strong, "c", "b"]
3 Bob performed the diff() and obtained [{"userid":2,"opcode":"deleteData","opcon
      ":["d",1],"loc":4},[]]
4 Bob sent CtoS_Msg with [{"userid":2,"opcode":"deleteData","opcon":["d",1],"loc
      ":4},[],{"TS":136}] via SendingOpsToController
```

LISTING 7.16: Snippet of console output during FSM execution of a DOM node splitting scenario (test "7").

Figure 7.10 shows a large screenshot containing several MATLAB windows at the point in the execution of test "7" where the *Controller* rejected Bob's operation. The window at the top-center of the figure shows the path taken through the state machine by the *Controller* to accomplish this rejection and return to the *Listening* state. The path taken through a diagram is highlighted in blue as the animation is executed, thereby assisting with the development of such models by making deadlocks and infinite loops easy to spot. The display blocks along the right side of the Simulink window in the top-left corner of the figure can be seen to contain the latest document content of all *Client* instances (Alice has a longer string since it already includes the *strong* node). Line 49 in the log of Listing B.6 corresponds to the last line shown in the top-right window showing the live console log. The bottom-left window shows Alice in the *Synced* state awaiting new operations, while Bob is in the *ApplyingRemoteOpsWithoutACK* state processing Alice's *FullOp*. Carol's *Client* is shown to still be in the *ApplyingRemoteOps* state to finish applying Alice's *FullOp*. The *Events Generator* is not shown but is awaiting the `MAX_TICK` condition to be met in order to leave the *SendingOp* state and perform the final consistency check.

Monitoring the dynamic nature of the distributed system in the concurrent fashion shown in Figure 7.10 was found to be invaluable for ensuring the models behaved correctly as their level of complexity continued to increase. The visual simulation of the FSM models was beneficial for colleagues and other developers wishing to understand the OT concepts that were implemented, as OT algorithms can

sometimes be challenging to grasp from reading source code alone. These factors, along with the core "serverless" and FSM-based design principles of the system, are therefore key to successful web implementations of such algorithms.

The seven integration tests presented in this chapter have revealed several recognizable patterns that reappear in the execution of similar tests due to the FSM-based nature of the CLOT integration algorithm. With the main patterns established to execute correctly, the DOM-based OT system was tested with more than 50 other custom sequences of remote and local edit operations to further increase the level of confidence in the algorithms. The comprehensive tests were created based on common edge cases and challenging scenarios typical of test suites from existing OT literature [133][134], as well as new DOM-based scenarios derived from real-world interactions with web applications such as WYSIWYG editors. The randomization feature, which was shown to generate test "6" of Section 7.3, allowed generating tests consisting of up to five random operations for up to three users and the end result remained consistent across all users after more than 12 consecutive hours of runtime (tests typically take less than one second to complete due to the C-based output of MATLAB). The robustness of the FSM-based design therefore allowed for the simulation of new OT operations not previously explored, such as splitting DOM-based text nodes, as was shown with the *giveData* operation.

The successful simulation of DOM-based operations and transformations by using FSM models within a real-time architecture has enabled a Web-Based Collaborative Platform to be implemented in JavaScript, as will be shown in Chapter 8.

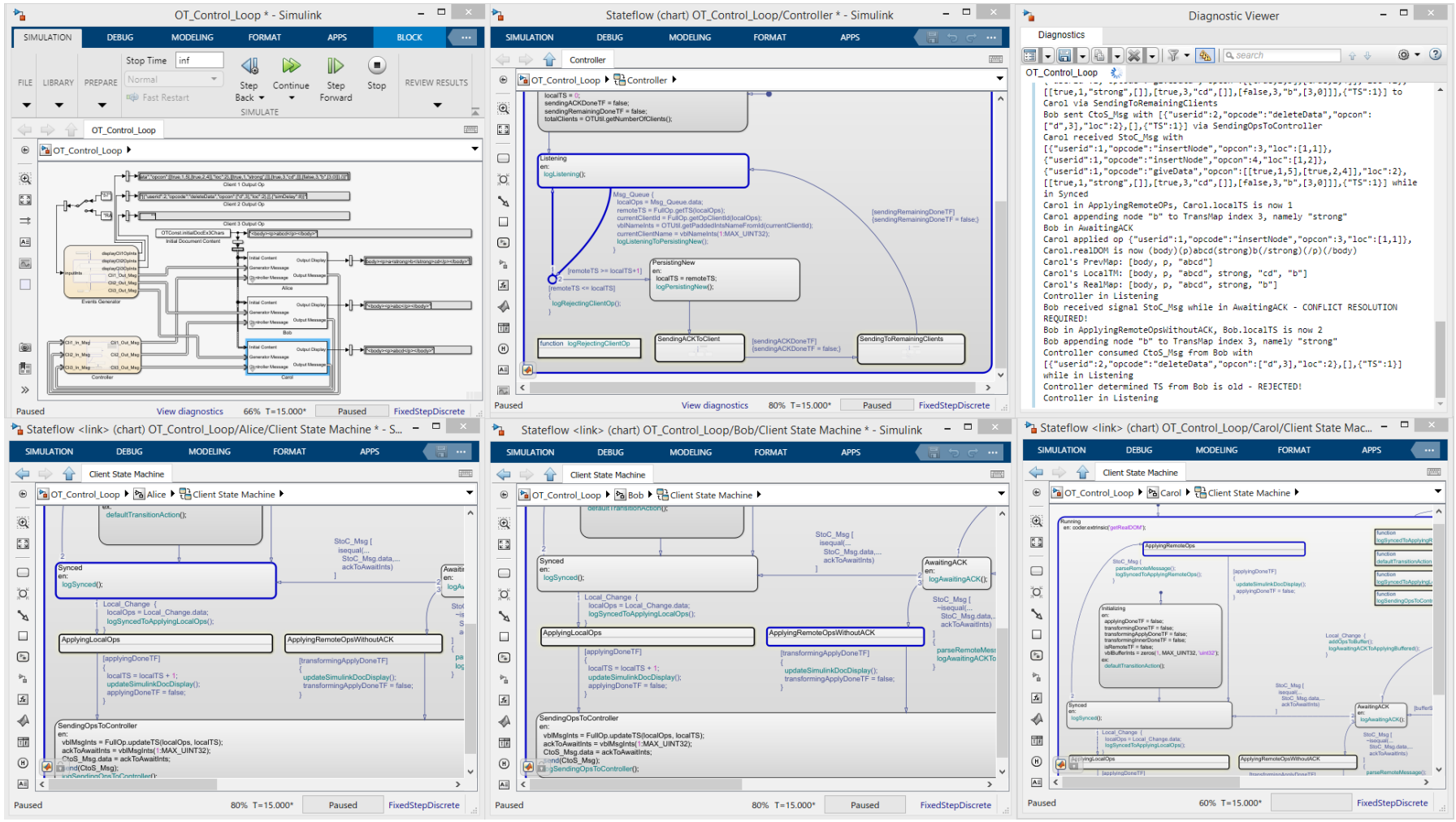



FIGURE 7.10: MATLAB windows for observing top-level Simulink model, Controller (Stateflow), console log output, Alice's Client (Stateflow), Bob's Client (Stateflow), and Carol's Client (Stateflow) during execution of a DOM node splitting scenario (test "7").

# Chapter 8

# Web-Based Collaborative Platform

Chapter 7 showed that the CLOT integration algorithm of Chapter 4 can be used together with the DOM-based algorithms of Chapter 5 to enable multiple participants of a collaborative session to maintain synchronized DOMs. As team-based collaboration increasingly takes place on the web, the ability to keep the local browser DOM synchronized among multiple users is important for enabling co-editing experiences on top of existing web applications such as WYSIWYG rich-text editors. To ensure that network-based interactions remain reliable, the architecture and algorithms of this thesis were designed to support a new "server-less" Operational Transformation approach that can make use of the scalability features and key-value storage services offered by Backend-as-a-Service (BaaS) providers such as Firebase [11], as will be discussed in this chapter.

Although not all DOM-based operations of Section 5.1 were explored in Chapter 6 and Chapter 7, the thoroughly-simulated models described therein were sufficient to determine the reliability and usefulness of the CLOT integration algorithm for enabling the real-time DOM synchronization capabilities and other architecture requirements identified in Section 3.3. The remainder of the implementation therefore continued alongside other researchers at the NCCT Lab with the development of the Web-Based Collaborative Platform. The MATLAB-based models and visual simulation tools made the OT concepts more accessible to all involved, while the structured FSM-based CLOT algorithm allowed keeping the JavaScript code

organized and complexity manageable as new operations and transformations were introduced.

In this chapter, the Web-Based Collaborative Platform is first devised by adapting the components of the control loop architecture and Finite State Machine models into a reusable design for the web. The chapter then shows how the Web-Based Collaborative Platform was evaluated through the development of a rich-text document co-editing application and a collaborative 3D virtual environment. The performance of the rich-text editor is measured by connecting multiple users to the same document and performing typical rich-text document editing actions to observe latency timings as the number of users is increased.

## 8.1 Platform Design and Implementation

While the control loop architecture of Figure 3.3 and FSM-based design remain at the heart of the Web-Based Collaborative Platform, an alternative view of the platform with high-level blocks more adapted to web-based technologies is summarized by Figure 8.1. This way, the rich-text co-editing application and multi-user 3D virtual environment described in this chapter can make adjustments to components of the architecture based on their specific requirements.

The *DOM Markup Language* component of the Web-Based Collaborative Platform of Figure 8.1 must render the DOM content within the user's web browser. A *DOM Modification Detection/Application* component monitors the designated root DOM node in which the content that is to be made collaborative resides. This component corresponds to the *Feedback DOM Observer*, *Comparison Logic* and *Client Content Processor* components of the architecture in Figure 3.3. Any changes detected to nodes therein must be submitted to the *DOM Synchronization Server* by using a WebSocket connection. The *DOM Synchronization Server*, previously the *Central Controller* in Figure 3.3, then persists the changes to a *NoSQL Database*. The *DOM Modification Detection/Application* component also receives changes from other remote users and applies the changes directly to the DOM of the local user's web page. All interactions with the NoSQL database must be done using a "livequery" mechanism that ensures all clients are subscribed to relevant data locations such that they will receive notifications related to changes in any of that data.

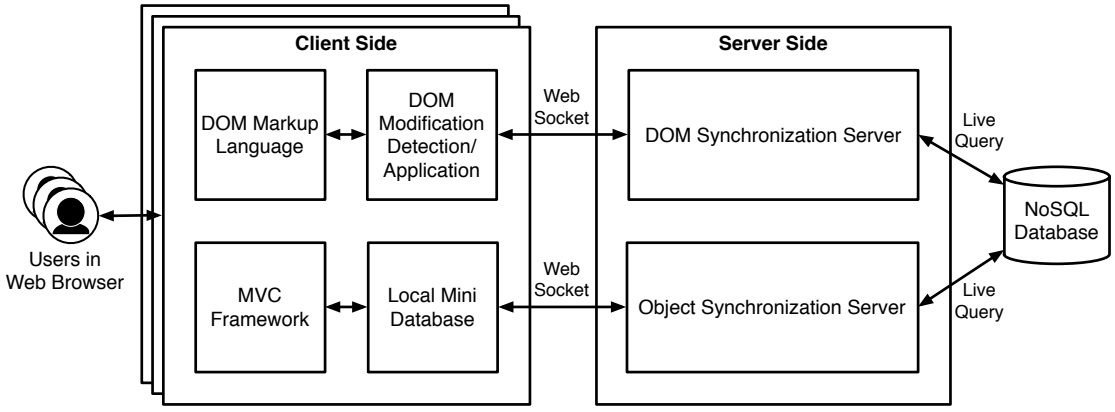



FIGURE 8.1: Main blocks of the Web-Based Collaborative Platform.

Additional collaborative functionality such as a multi-user chat can be accomplished through shared key-value pairs, which can be used to store messages and online users. The client-side code must specify which JavaScript variables need to be synchronized with other users in this fashion. The chat-related state therefore can be synchronized using a client-side *Local Mini Database* that is synchronized to the *NoSQL Database* via an *Object Synchronization Server*.

## 8.2 Rich-Text Editor

As mentioned in Section 1.1, the OT-based DOM synchronization capabilities explored in this thesis were motivated by the desire to make existing web-based rich-text editors, such as those within popular wikis and content management systems, collaborative with co-editing functionality similar to products such as Google Docs (*transparent adaptation*). The Web-Based Collaborative Platform must therefore synchronize the DOM on which rich-text editors operate without requiring modifications to the code of the rich-text editors themselves. This section describes the design and implementation of a collaborative rich-text document editor created by combining the Web-Based Collaborative Platform with a popular WYSIWYG editor component. Previous publications of the NCCT team related to the rich-text editor functionality include [38], [39], and [40].

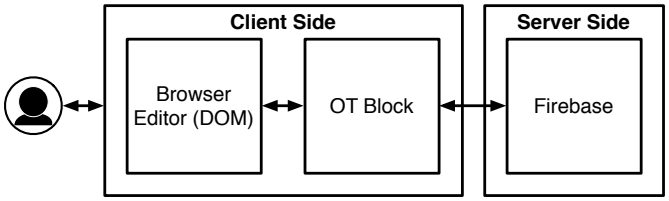



FIGURE 8.2: Basic collaborative text editor architecture using Firebase.

### 8.2.1 Design and Implementation

The web-based rich-text word processor was implemented by following the basic architecture shown in Figure 8.2. The TinyMCE editor [25] was used as the source of the HTML DOM content to be kept synchronized. The *Controller* was implemented using the BaaS provider *Firebase* [11]. Firebase handled the *Timestamp Logic* and *PubSub Broker* functionality of the system, as well as provided data persistence for key-value pairs. While the *PubSub Broker* and persistence functionality is part of the core BaaS offering of Firebase, the *Timestamp Logic* required the use of the *transaction()* function of the Firebase API [107] to ensure that only values that do not already exist (namely, new timestamp values) can be written to Firebase (a technique also used by Firepad [109]). The *OT Block* of Figure 8.2 handles the change detection and transformation logic, and also applies changes to the DOM (as per the *Feedback DOM Observer*, *Comparison Logic* and *Client Content Processor* blocks in the architecture of Figure 3.3). For this basic editor, no additional application data needed to be persisted (the *Local Mini Database* component from Figure 8.1 was omitted).

### 8.2.2 Results

Figure 8.3 shows two users which have completed the same *giveData* scenario that was previously described in Section 7.4, where Alice (middle window) and Bob (top window) start with the DOM of Listing 5.18 (a paragraph node within a body node has a text node containing "abcd"). Alice sets the "b" to bold while Bob deletes the letter "d". Alice, which still has the "b" character selected in the figure, also has her console open and scrolled back to the time when the *FullOp* containing the *giveData* operation was generated. The operation generation process of the *Comparison Logic* component is visible at the top of the console output and presents lengthy sentences as descriptions of the changes. Summarizing changes with this level of detail was important for the successful implementation and debugging of

the Web-Based Collaborative Platform. The *prevMap*, *transMap* and *realMap* values introduced in Section 5.1.2 are also shown near the bottom of the log as `Prev`, `Temp` and `Real`. In the top window, Bob is shown to have received and applied this operation (close inspection of the figure will reveal his letter "b" ended up bold). The successful OT-based splitting of a text node by operating directly on a real-world DOM (TinyMCE) while keeping all web clients synchronized therefore demonstrates new functionality not previously achieved by the solutions identified in Section 2.3 (in fact, the example above first appeared in Section 2.3.3 as a limitation of ShareDB [96] and Webstrates [97] after an analysis of their source code).

The corresponding Firebase database contents can be seen in Figure 8.4. The last operation is expanded, showing an operation with an *opcon* of {"*d*" : 1} and *loc* of 4. The operation is shown to have been persisted correctly and the list of operations (forming the entire document history) can therefore be obtained and replayed by late-joining users. In addition to persisting a history of changes, the latest version of the document representing the synchronized portion of the DOM can also be persisted so that new users do not need to play back all changes to get to the most recent version of the data. Two users are also currently shown to be online based on the "users" key, which Firebase updates automatically based on the number of active WebSocket connections for this document.

The system was tested with over 80 users in the same collaborative session using the Windows-based Chrome web browser running on a 3.4GHz Intel Core i7-4770 with 32GB of RAM. 80 users is more than the limit of 50 users per session imposed by Google Docs [135]. The Chrome DevTools were used to observe the round-trip time of outgoing and incoming WebSocket interactions with Firebase as the number of concurrent users was increased with new browser tabs. Figure 8.5 plots the latencies observed from the time the message containing a single-letter *insertData* typing operation was sent until the acknowledgement was received. The latency values range from 39ms with 10 users to 46ms with 80 users. The Firebase performance was therefore found to remain relatively stable as more users were added to the session, and while the delay did increase slightly with additional users, some of the increase could be due to additional CPU load. Even with 80 users, typing inside the editor remained responsive, while editing operations of all users continued to be synchronized.

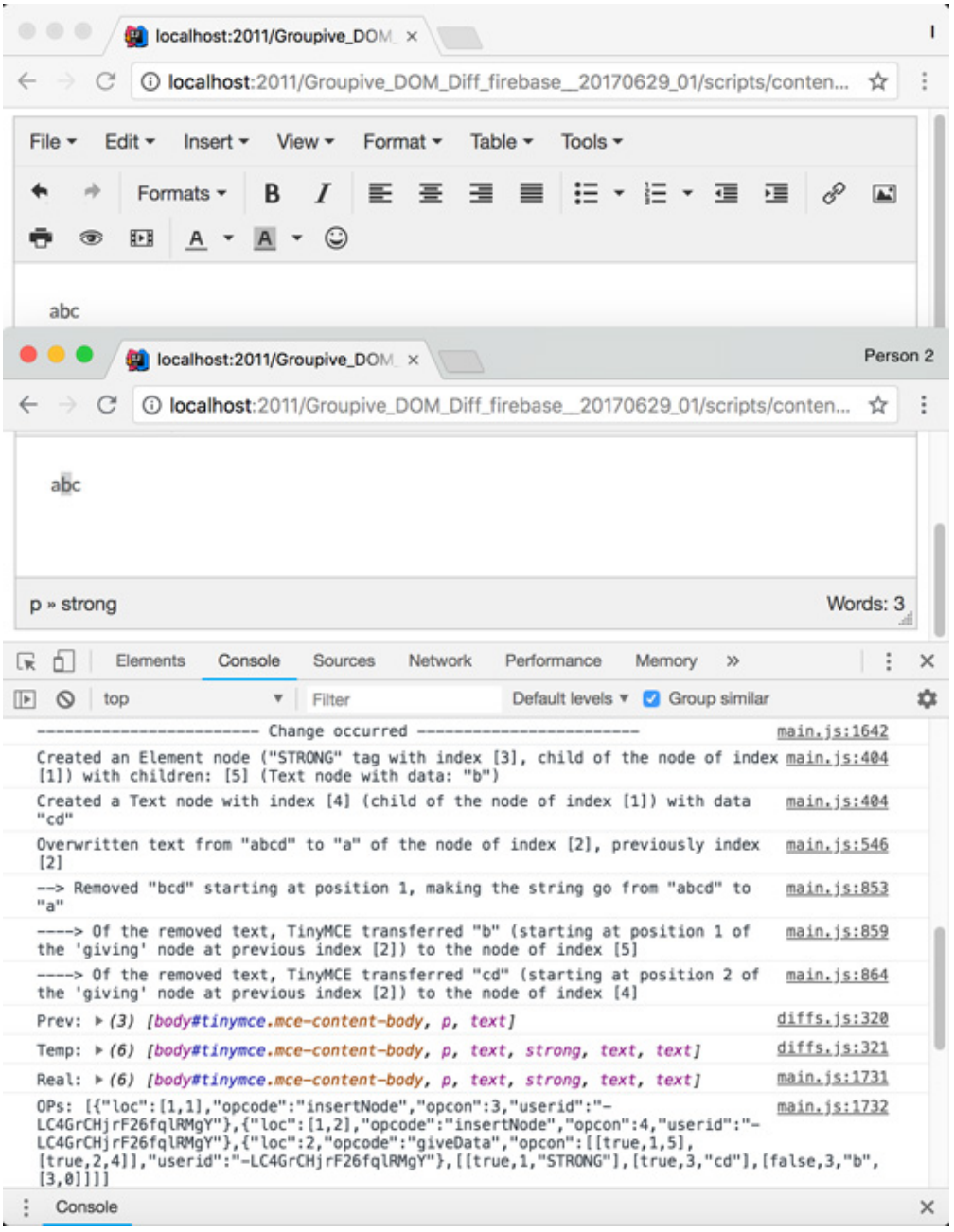



FIGURE 8.3: Synchronization of node splitting "bold" operation using DOM of TinyMCE text editor.

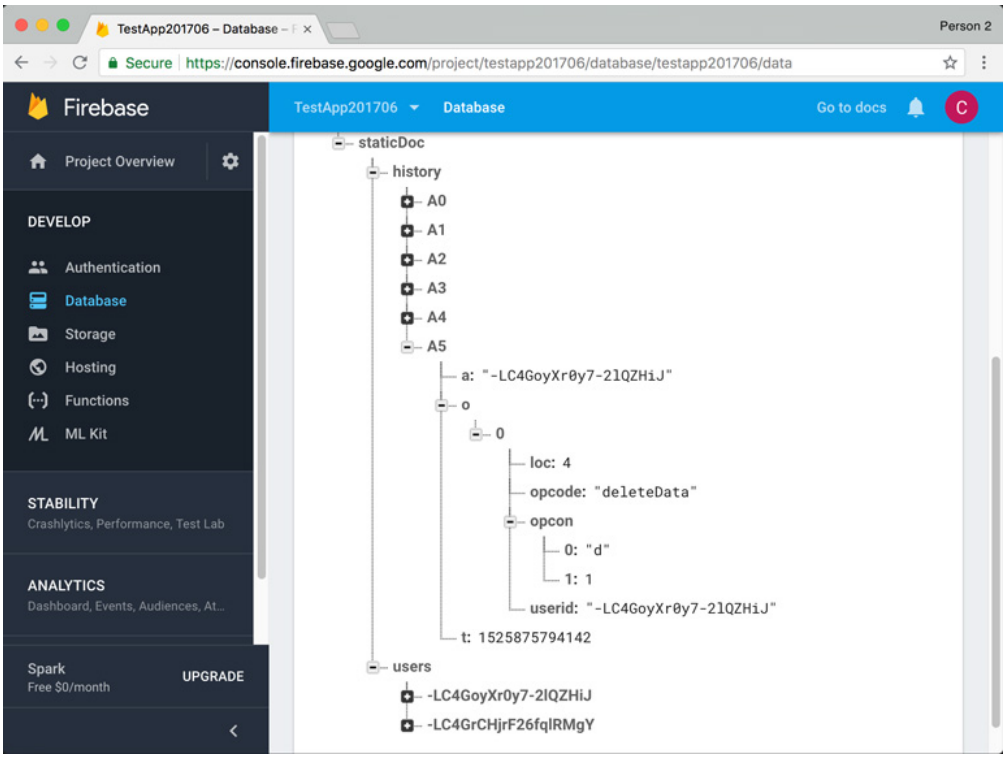



FIGURE 8.4: Firebase database contents showing the last operation of a two-user session.

Firebase has been shown to scale to over a million concurrent users [103][104], therefore making it an unlikely bottleneck if used properly and motivating the "serverless" approach taken in this thesis. However, the timestamp mechanism of the CLOT integration algorithm relies on a retry-based approach to accept operations in an atomic fashion to a central history. With many concurrent users of varying latencies participating in a session, there is therefore the potential of some (higher latency) clients being "starved" as they continuously buffer their local operations and attempt to re-transform and re-send the unacknowledged operation. While further experimentation is required to determine if this is a problem for real-world use, the developers of the Firepad editor (which also uses the Firebase service as per Section 2.3.6) have generally found that the "bursty" nature of normal human typing contains breaks that allow "starved" clients to catch up [136]. The *diff()* approach introduced in Section 5.1.2.3 also has the effect of combining multiple changes into a single *FullOp* so that a single successfully-accepted message is enough for a "starved" client to communicate all of its changes (since its previous successful message) at once. Realistically, a very large number of users typing at the same time in the same document is unlikely to happen frequently in the real world due to Groupware-related challenges such as coordination of the

parties involved [2].

In order to gain a performance advantage over other approaches listed in Chapter 2, the design of the operations of Section 5.1.1 ensures that the number of operations that need to be communicated and processed is reduced as much as possible. Instead of requiring $n$ operations of type *insert* or *delete* to be processed for a string of length $n$, operations such as *insertNode* (see "the quick brown fox" example of Section 5.2.1) and *giveData* (see the "abcd" example of Section 5.2.6) were shown to support strings of characters, while the *moveNode* operation handles the underlying *insert* and *delete* manipulations with a single operation instead of two separate ones (which is also important for intention preservation, as per Section 2.2.1.4). Section 5.2.1 highlighted how the *realMap*, *prevMap* and *transMap* approach, when combined with careful design of operation attributes, enables forming and referencing data structures known to exist on all *Client* instances in order to identify a node's location. The `loc` approach therefore allows for less verbose operations (leading to improved network performance) than previous DOM approaches, since all known previous approaches identified nodes by their full path rather than just two integers [67][120][97]. The FSM-based design of Figure 5.3 (Section 5.3, later modeled in Section 6.5.1.3 and simulated in Section 7.4) introduced how DOM-based optimizations are made possible once such node locations are determined, where checking the parent of a node can be used to

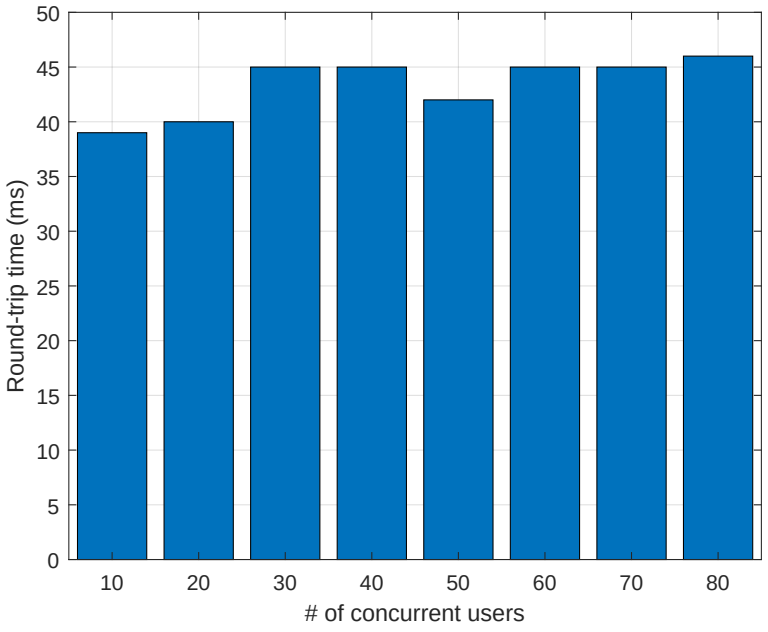



FIGURE 8.5: Bar graph of observed round-trip time measurements from operation sending until acknowledgement using Firebase.

bypass entire sections of transformation logic. Significant amounts of computation can therefore be avoided by designing operations to efficiently pinpoint the sections of a document that they will affect, thereby preventing them from interfering with each other. By reducing the number of required transformations with the FSM approach, and therefore reducing algorithm time complexity in general, delays are reduced throughout the system and the system becomes more responsive.

Figure 8.6 shows a four-user collaboration session in a more refined version of the previously-mentioned rich-text editing experiments. The real-time web application has been named "Groupive" by the NCCT team for its ability to make *group*work more collaborat*ive*. Online users are denoted by the colored images in the top-right corner of the interface. The document along the left side of the screen remains synchronized, even with complex rich-text editing operations such as creating bulleted lists, manipulating tables (inserting/removing rows/columns), and inserting images. Chat messages are displayed on the right side of the screen. Both the document and past chat messages are restored from the corresponding databases in the case that the user refreshes the entire web page. Rather than using the services of a provider such as Firebase, the *DOM Synchronization Server* and *Object Synchronization Server* components were developed as microservices [137] using the Node.js-based Socket.IO WebSocket library [138] to communicate with the browser. In this way, the "serverless" requirements identified in Section 3.3 are met while retaining full control over all data, thereby supporting "air-gapped" internal deployments. Scalability can be achieved by starting both Node.js servers as cluster processes and using Redis for inter-server communication [139], although this remains to be explored, along with various user studies and performance measurements (no other DOM-based approaches currently feature a "serverless" architecture).

## 8.3 Virtual Reality Application

The Web-Based Collaborative Platform was applied in the creation of a collaborative 3D environment, as previously explored in [41] by the NCCT team, and summarized in this section.

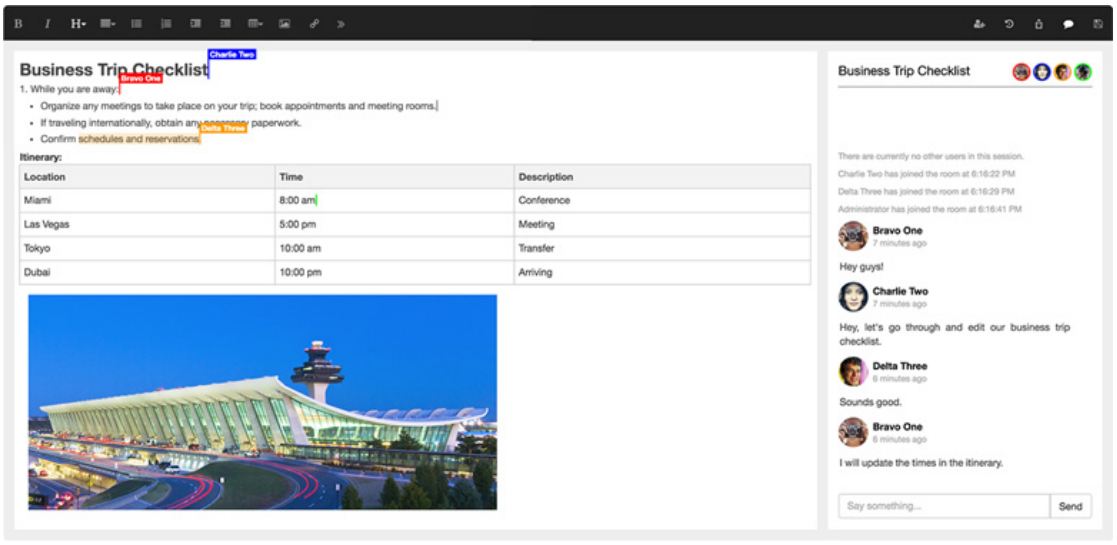



FIGURE 8.6: A four-user collaborative document editing session with a table and other rich-text elements in "Groupive".

### 8.3.1 Virtual Reality on the Web

Ever since science fiction introduced the concept of the "metaverse", researchers in academia and industry have been inspired to assemble the hardware and software technologies required to implement such a persistent virtual world inhabited by real people [140]. Driven partially by the mobile industry, recent technological advancements have resulted with affordable display, motion sensing and 3D rendering technologies that have now made it possible for regular consumers to have comfortable and immersive virtual reality (VR) experiences by using commercially-available head-mounted displays (HMDs). By covering the user's peripheral vision with a high frame rate display and using sensor data to instantly update the rendered field-of-view, VR HMDs are able to effectively create a sense of immersion and presence in the virtual space. While much work remains to be done for truly natural and intuitive human computer interaction within virtual worlds (such as for hand, finger and full-body tracking), the pieces are finally starting to come together to enable the creation of quality VR applications for entertainment, education, remote work, training, and much more.

A key attribute of the metaverse is the social interaction and collaboration between users in the same persistent virtual space, and this requires researchers in the field of distributed computing to tackle problems such as scalable cloud-based or peer-to-peer architectures that provide reliable and low-latency real-time data synchronization. An expanding metaverse requires multiple developers and artists

to simultaneously create content that can be seen by other users and is persisted for future visits. The technical challenges of creating such a multi-user system can be overwhelming and therefore hinder progress in this direction.

VR is expected to become a significant market in the near future, and the recent surge in interest has not gone unnoticed by web browser makers. With the rapid release cycles of today's modern web browsers, versions of popular browsers already exist [141][142] that can interface with virtual reality devices based on a standard called WebXR [143]. This standard allows HTML, CSS, JavaScript and WebGL to be used to create content that is viewable using HMDs. VR experiences can therefore be created that benefit from the openness of the web and features such as cross-platform compatibility (including on mobile devices), instant publishing, cloud hosting, as well as communication protocols such as WebSocket and WebRTC. In addition, web-based VR experiences can be shared with a simple URL and can contain hyperlinks to other VR experiences. This is in stark contrast to VR applications developed using Unity [144] or Unreal Engine [145], which may require large downloads, as well as extensive installation and setup procedures.

Web technologies can make it easier for developers to have access to the networking technology required to enable the creation of virtual environments by allowing developers to apply well-established web development practices and tools when creating multiplayer content. In addition, the web provides access to large amounts of freely available 3D graphics, image data and components that can be reused in the creation process. The system proposed in this section builds on the Web-Based Collaborative Platform and its DOM synchronization functionality to enable the creation of collaborative 3D environments using simple HTML-like markup (declarative HTML).

### 8.3.2 Design and Implementation

Figure 8.7 shows how the Web-Based Collaborative Platform can be applied in this domain. The A-Frame project [146] was selected as the open source component for generating the 3D DOM content that can be viewed on VR HMDs. Released in December 2015, A-Frame offers a user-friendly approach for creating 3D environments through DOM modifications. The implementation of a VR scene in A-Frame containing a cube is shown in Listing 8.1. Figure 8.8 shows the result when this file is viewed inside a web browser. The simple use of a *a-cube* node

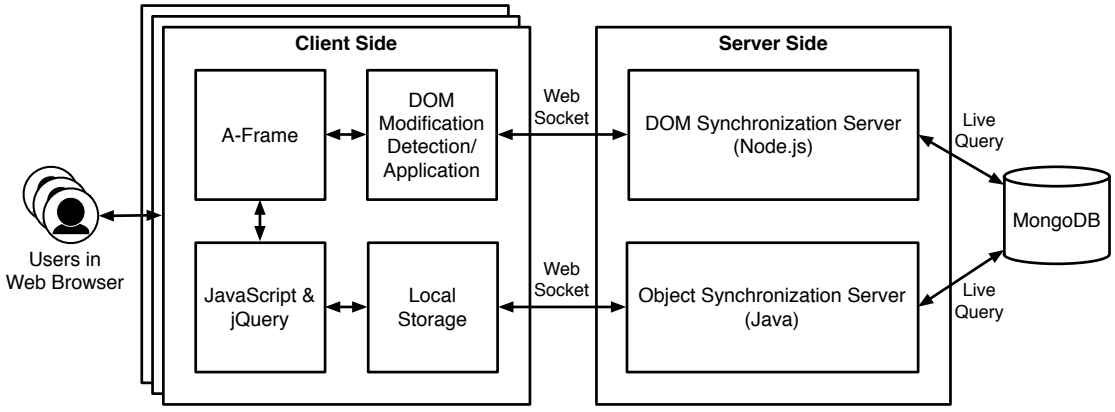



FIGURE 8.7: Implemented architecture for synchronized 3D virtual spaces.

inside the `a-scene` node of the HTML *body* node causes an interactive scene with a 3D cube to appear inside of the user's browser. The icon in the bottom right of the screen can then be used to output the scene to a connected VR HMD.

```
<!doctype html>
<html>
  <head>
    <title>VR Cube Example</title>
    <script src="aframe.min.js"></script>
  </head>
  <body>
    <a-scene>
      <a-cube></a-cube>
    </a-scene>
  </body>
</html>
```

LISTING 8.1: A-Frame cube markup example.

The implemented *DOM Modification Detection/Application* component works by setting a HTML5 *MutationObserver* object as a "change observer" to listen for any changes to the relevant portion of the DOM (within the `a-scene` element node) and sending the corresponding changes to the server-side. Based on the requirements of the "serverless" approach outlined in Section 3.3, a Node.js-based *DOM Synchronization Server* was created that makes use of a MongoDB database. Communication with this server was performed using JSON messages via the WebSocket protocol. Similarly, a Java server was set up as a WebSocket server to manage the synchronization of other JavaScript variables such as chat messages and user presence, thereby forming the *Object Synchronization Server*. JavaScript

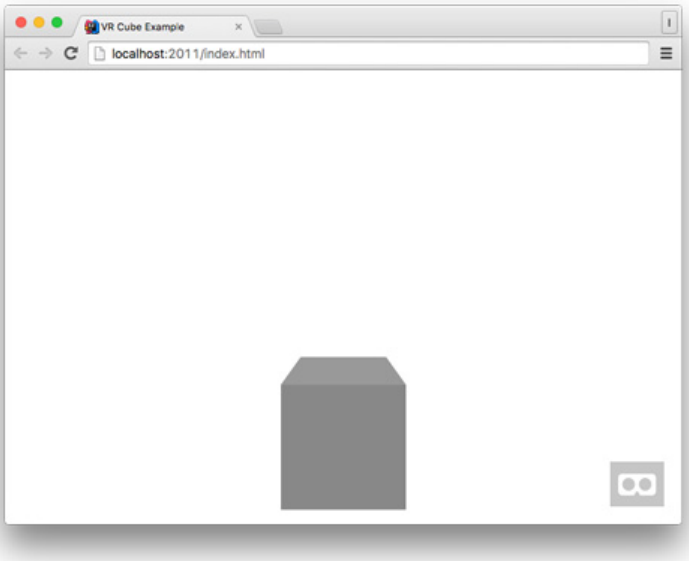



FIGURE 8.8: Result of running A-Frame cube markup example inside of a web browser.

and jQuery were used to communicate with a *Local Mini Database* that was implemented using the Local Storage functionality of modern web browsers.

A-Frame contains a `camera` element node and a `canvas` element node which are user-specific (based on the user's selected view and resolution) and must therefore not be synchronized. All remaining elements were grouped using A-Frame's *a-entity* node, which was given a unique ID for binding as the root node. The *DOM Modification Detection/Application* component was then set up to act on this unique ID as the root node. Any DOM changes coming from the server (such as from other users) are applied directly to this DOM based on the details contained in the delta object.

To make it easier to insert objects into the environment, a text dialog was added that users can bring up by pressing the enter key. The dialog was created by applying a HTML `canvas` element node as a texture inside the 3D environment but is only visible for the initiating user. The user can then type the name of an object from a predefined library to have the object spawn inside of the collaborative environment. By typing "box", for example, a new `a-cube` element node (previously shown in Listing 8.1) is inserted into the synchronized `a-entity` section of the DOM by using JavaScript. This change to the DOM creates a "change observer" event and the *diff()* function (Section 5.1.2.3) is used to form a description of the element node (including properties such as size and color) and the position at

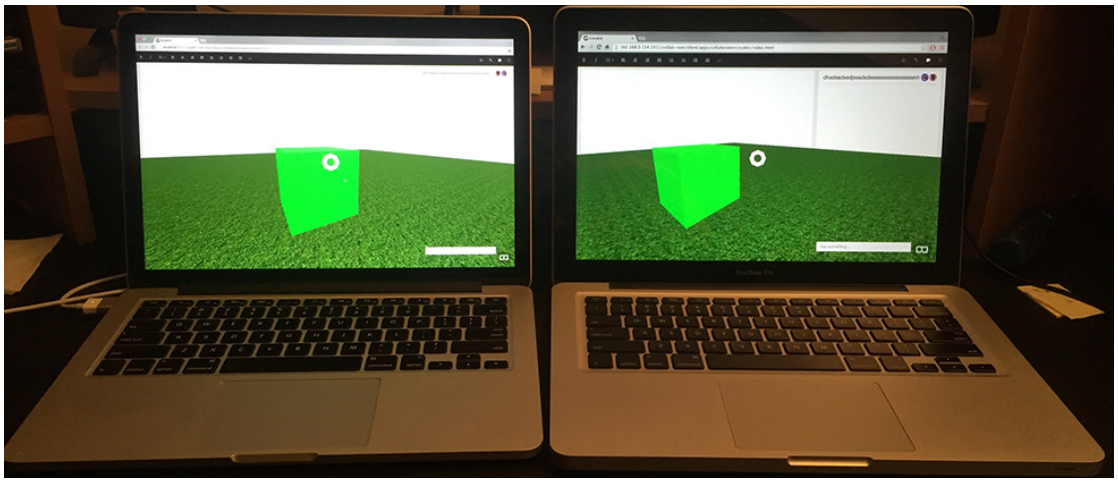


FIGURE 8.9: Two users in the same 3D virtual space seeing the same object from different angles.

which it was inserted in the DOM hierarchy. This information is packaged as an *insertNode* operation (described in Section 5.2.1) and transferred to the *DOM Synchronization Server*, which accepts it (if it doesn't conflict with other operations, as per the CLOT integration algorithm of Chapter 4) and persists it in a specific location in MongoDB. Other users connected to the same session (the same virtual environment) will have subscribed to changes to that location in MongoDB and will therefore be notified of the new *insertNode* operation that was added. Once the web browsers of the other users receive the operations, they apply them to their local DOM (or transform them first in the case of a conflict). Through these incremental updates, all users in the session will result with identical DOMs, and therefore they all participate in a shared virtual environment that they all can change.

### 8.3.3 Results

The implemented system can be seen in Figure 8.9, where two users are looking at two different sides of the same cube. JavaScript was added to allow the cube color to change when a user clicks the cube, and since the `color` attribute of the `a-cube` element node gets updated by the JavaScript, an *updateAttribute* operation (Section 5.2.10) is synchronized to all other users in the session so that all users see the same color.

Figure 8.10 shows a white dialog box on the right side of the screen that appeared after the user pressed enter. By typing "tree" and pressing enter again, the user


FIGURE 8.10: A tree object was spawned by typing "tree" into the text box.

submits the command to spawn a tree into the environment, such as the one seen on the left side of the image. Objects can therefore easily be imported and positioned within the multi-user environment. The background seen in this image was created by using a `a-sky` element node that points to the URL of a equirectangular image.

Certain A-Frame components may have additional variables that need to be synchronized, and this can be accomplished by using the *Object Synchronization Server*. For example, a theater-like environment is shown in Figure 8.11 where users can watch videos together. Figure 8.12 shows another user in the same environment. These examples makes use of the `video` element node of A-Frame. Techniques previously presented in [147] were used to synchronize the time of the video among all users within the session. As new components are created for the A-Frame library, they will therefore be instantly compatible with the synchronized environment as well.

It is important to note that browser-based VR technology is still in its infancy. For example, few 3D file formats are supported for importing objects into the 3D environment, and even supported formats will often exhibit missing textures or polygons. Even with relatively few test objects in the environment, browser-based 3D graphics libraries are generally more demanding of CPU and memory performance than native desktop applications. A-Frame developers have promised support for graphics shaders, ray casting calculations, physics, collisions, and many other features in the future.

FIGURE 8.11: Synchronized 3D theater environment with video playing for all users. The frame that the Oculus Rift DK2 HMD receives is shown.


FIGURE 8.12: A second user in the same theater environment appears as the helmet avatar on the left side.

## 8.4 Summary

Table 8.1 summarizes how the most prominent approaches reviewed in Section 2.3 fare with respect to the requirements identified in Section 3.3 that lie at the heart of the Web-Based Collaborative Platform. It can be seen that no existing approach fully meets the requirements at the "Excellent" level, as this thesis achieves with the CLOT (FSM OT) approach. In addition, no known existing research has presented FSM-based models of an OT system. This thesis not only modeled OT algorithms using FSMs, but also analyzed results from executable simulations of the proposed models and architecture. The models and architecture were then used to develop the Web-Based Collaborative Platform which was used in the implementation of the two functional collaborative web applications presented in this chapter. By further summarizing the main research contributions of this thesis, Chapter 9 will draw conclusions and provide an outlook for future work.

TABLE 8.1: Assessment of existing approaches and CLOT (FSM OT) approach with respect to identified requirements.

| | **FSM-Based Design** | **DOM Support** | **Serverless** |
|---|---|---|---|
| **Wave** [87] | – | – | – |
| **Webstrates** [97] | – | + | – |
| **GCI** [30] | – | ∘ | – |
| **Pulsar** [102] | – | ∘ | – |
| **Firepad** [109] | + | – | ++ |
| **Ignat et al.** [120] | – | + | – |
| **CLOT (FSM OT)** | ++ | ++ | ++ |

Support Levels: – None or Very Poor ∘ Poor + Good ++ Excellent

# Chapter 9

# Conclusion

The optimistic consistency control method known as Operational Transformation (OT) has been studied by researchers for over three decades, with centralized versions lying at the heart of most real-time web co-editing tools in industry. Concurrent document editing is now an essential team collaboration technology for government departments, universities, banks and other organizations, almost all of which can benefit from the real-time coordination of various activities such as live meeting minutes, co-editing a software module, or co-authoring a document with a quickly-approaching deadline. Products such as Google Docs [28] have shown how real-time team collaboration tools can now be accessed as cloud services from a regular web browser, but many organizations also seek on-premises and hybrid cloud deployments for additional control over their data.

In this thesis, a Finite State Machine approach was used to capture the dynamic client-server interactions of OT systems by modeling them as real-time systems. A new architecture was introduced which can describe and solve the dynamics of co-editing processes controlled by the use of a *Central Controller*. A *Feedback DOM Observer* component was used together with a *Comparison Logic* block to identify and encapsulate changes in the DOM as operations that can be transformed in a way that preserves user intentions beyond existing approaches. A Rule-Based *Decision Maker* component was shown to enable multiple end-user clients to transform a set of operations and remain synchronized through the use of a Control Loop-based OT integration algorithm (CLOT) designed to achieve

convergence while addressing the requirements of modern "serverless" web applications. Operations received from the feedback control mechanism were applied to a local DOM as part of the *Client Content Processor* component.

Based the successfully-simulated control loop architecture and FSM-based models, a Web-Based Collaborative Platform was developed to support centralized DOM-based Operational Transformation algorithms ideal for the modern web. A collaborative rich-text editor and a multi-user 3D virtual environment were built on top of the platform to illustrate its effectiveness.

## 9.1 Summary of Contributions

A recent paper of Sun et al. [76] expressed the following view about the present state of OT research:

> "The basic ideas and external effects of OT are simple to illustrate, but inner-workings of OT are not simple to understand by non-experts or application developers. There is still large room for improvement in making OT solutions more accessible to practitioners, and most importantly in applying OT to real world collaborative applications"

The executable Finite State Machine approach, overarching control loop architecture, new algorithms, and example applications presented in this thesis aimed to advance the OT state-of-the-art in exactly this fashion. The research contributions of this thesis can be summarized as follows:

1. A novel architecture was proposed based on the real-time behavior when synchronizing changes among multiple versions of replicated content by modeling such distributed systems as a real-time feedback control loop. The centralized OT optimistic consistency maintenance technique was found to be suitable for this architecture and was extended to enable DOM-based synchronization. This type of feedback architecture has not been associated with OT prior to this work.

2. Finite State Machine (FSM) theory was used to capture the dynamic client-server interactions of OT systems by modeling them as real-time systems.

To show that the causality, convergence, and intention preservation properties of OT systems are satisfied, the real-time behavior was simulated in MATLAB with co-editors performing a number of realistic situations consisting of sequences of remote and local editing operations defined as part of a DOM-based test suite. Previous academic approaches did not explore FSM-based modeling and visual simulation of OT algorithms.

3. A new Control Loop-based OT integration algorithm (CLOT) is presented that makes use of unique timestamps to omit the requirement for server-side transformations, making it ideal for modern scalable "serverless" deployments. To the author's knowledge, no other such approach has been published this far.

4. A list of ten operations was determined that enables capturing and replaying DOM-based changes, including the splitting of a node which happens when an operation such as "bold" is used in a rich-text editor. While further experimentation is required to evaluate every possible transformation, the resulting operations list is longer, more focused on intention preservation, and more applicable to real-world DOM changes than any other previous work.

5. A Web-Based Collaborative Platform was successfully implemented using JavaScript to provide DOM-based synchronization capabilities to web-based applications that can benefit from multi-user support. As was shown with the early implementations of a rich-text editor and a 3D virtual environment, the DOM synchronization capabilities and versatility of the platform are unique.

## 9.2 Future Research

The proposed architecture and Web-Based Collaborative Platform aim to increase the availability of real-time collaborative web applications by making them easier to understand and develop. The work presented in this thesis is therefore worth pursuing further in the direction of a World Wide Web that increasingly encourages online content creation and user participation – two themes at the heart of the modern web. This section identifies several avenues for potential improvements and enhancements to the work presented in this thesis.

### 9.2.1 Algorithm Enhancements

The algorithms presented in this work focused on Operational Transformations without undo [60][148] or operation compression [149] functionality. However, the FSM techniques described in this thesis can be applied to assist with the practical implementation of such algorithms by allowing developers to focus on smaller pieces (e.g. a single transformation) encapsulated as individual states. Applying the control loop and FSM-based distributed systems principles of this thesis to Commutative Replicated Data Types (CmRDTs) (Section 2.2.2) is also worth further investigation and may help address some of the numerous remaining research challenges of this emerging technology [76].

The security and access control implications of the algorithms presented in this thesis are also left to be investigated as future work. For example, browser debugging tools allow adding `script` tags to the synchronized portion of the DOM, potentially resulting with the transfer and execution of the injected scripts by all receiving participants of a collaborative session. Collaborative sessions with untrusted parties should therefore be avoided until the new and interesting security challenges introduced by the DOM synchronization approach have been carefully evaluated.

### 9.2.2 New Collaborative Applications

While this thesis already presented several use cases where DOM synchronization can enable powerful new methods of group-based interaction, there exist several other opportunities for extension to the work presented. By packaging the client-side code of the the Web-Based Collaborative Platform into a separate JavaScript file, the robust FSM-based DOM synchronization functionality can be included by existing editing applications with relative ease. Figure 9.1 shows three other content environments that have already been tested and found to successfully remain synchronized, and are therefore good candidates for future work. The Wikimedia Foundation has been exploring the idea of a collaborative editor for Wikipedia authors with little progress [18], but the platform designed in this thesis appears to synchronize Wikipedia content correctly in early testing. Already identified as the most popular blogging platform in Chapter 1, WordPress [21] can also benefit from a plugin or similar component developed based on the technology presented

in this thesis. Finally, PageCloud [150] is an example of a complex visual website design tool that can be transformed to allow multiple users to edit the same web page at the same time. This has not been explored in any previous academic work but is now feasible. While results for these three web applications already appear very promising, many other web-based applications can benefit from the multi-user functionality enabled by the Web-Based Collaborative Platform.

### 9.2.3 Artificial Intelligence

Several opportunities exist for incorporating principles from the field of Artificial Intelligence into the work presented in this thesis. For example, the *Decision Maker* from the architecture of Section 3.4 can be modeled as an "expert system", where the user's changes (*Facts*) are combined with the operation-related "if-then" rules from a *Knowledge Base* and with a *Database of Transformations* as input to the *Inference Engine*, the output of which is used by the *Resolution Manager* to deduce the resulting operation to apply to the *User Interface*. This is illustrated in Figure 9.2.

The publisher-subscriber mechanism presented in this thesis also opens the door for agents and robots that apply deep learning concepts to assist collaborative sessions. For example, bots can be developed that themselves subscribe to documents and modify the DOM by applying services such as real-time grammar checking or translation services. Similar concepts without DOM synchronization have already been explored in other research [2][88][102][39]. Enterprise messaging products such as Slack provide integration features that have demonstrated the importance of such agents for real-time teamwork [53].

### 9.2.4 Peer-to-Peer & Blockchain

Decentralized collaboration has recently gained significant attention as censorship and privacy issues are becoming more important in our society. In addition, decentralized approaches offer increased failure tolerance and can provide users and organizations with more control over where and how a document is stored. While the quality of internet access across the globe continues to improve and cloud services such as (Google-owned) Firebase have been shown to scale efficiently through the use of multiple geographically-distributed datacenters [103], the growing thirst

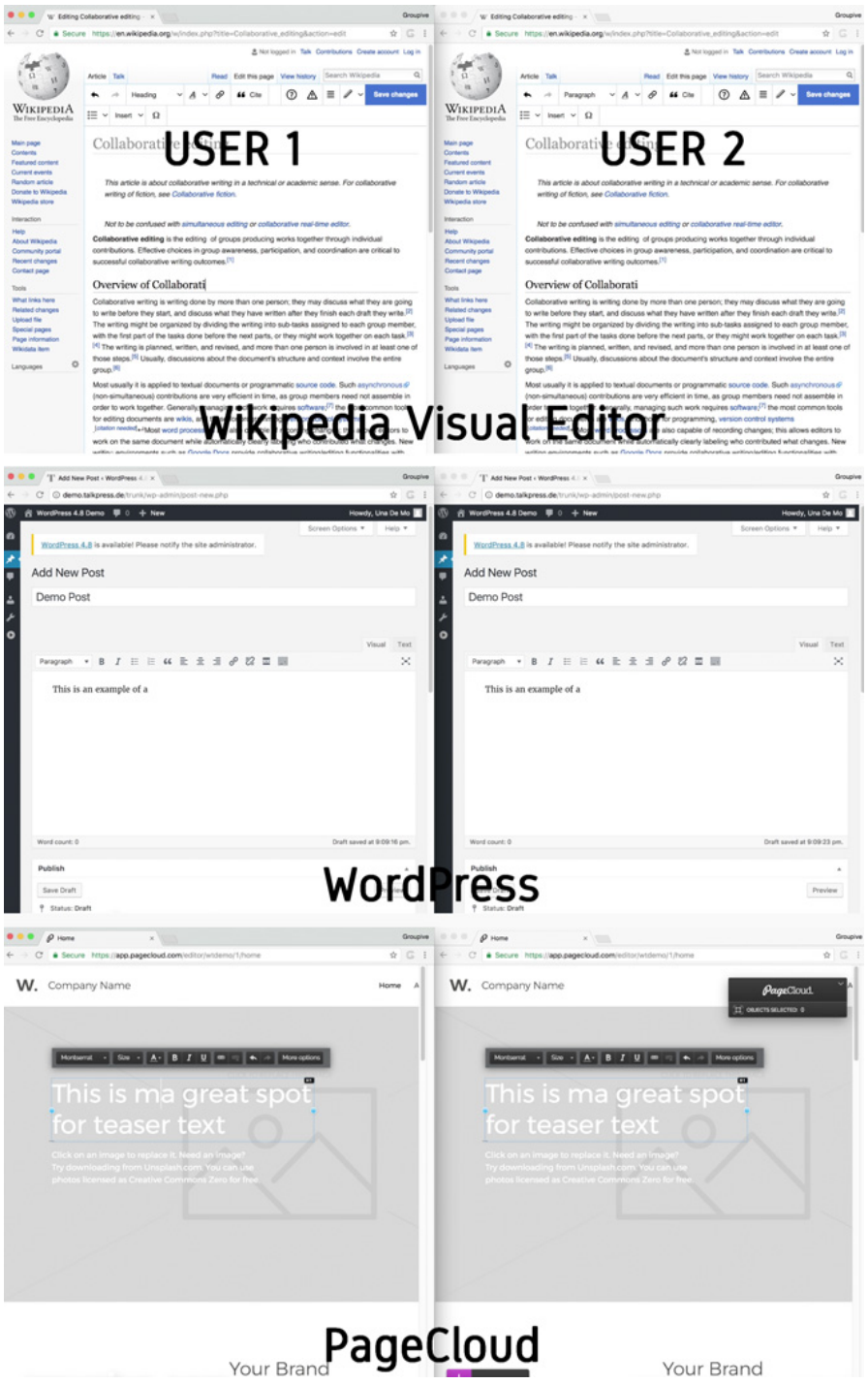



FIGURE 9.1: Early work on synchronizing Wikipedia Visual Editor, WordPress and PageCloud.

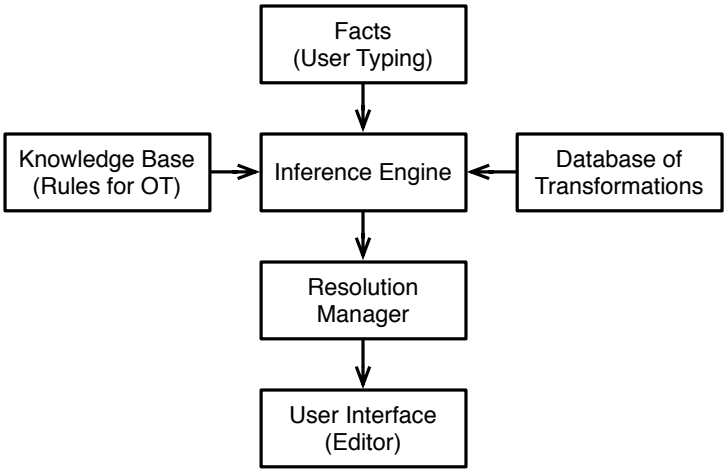



FIGURE 9.2: Modeling the Decision Maker block as an Expert System.

for Big Data and real-time communication has motivated researchers to also evaluate decentralized and hybrid approaches. For example, Chapter 1 mentioned collaborative use cases such as multiple users adding notes to the same document during a team meeting. If two laptop users are sitting beside each other in a meeting room in Australia and the nearest datacenter for the centralized collaborative service is in North America, it may not be efficient to have every keystroke of every user travel such large distances and incur the related round-trip latency penalties. Similarly, if the cloud service (or related backbone segment) is temporarily offline or blocked, it may be beneficial for a business if the two laptops are able to communicate directly, thereby allowing the users to continue their real-time collaborative work.

With roots in distributed databases and secure peer-to-peer consensus mechanisms, blockchain technology is emerging as a platform on top of which novel decentralized applications ("dapps") can be created [151]. Since the application logic, as well as the application data, resides in the distributed network, the web application and its use is not controlled by one central authority. The Operational Transformation approach of this thesis is a good fit for integration with blockchain technology since new operations that make up a rich-text document are built on top of previous operations, where previous operations (that make up the document history) should remain as a permanent record of that document. Use of blockchain technology would also provide cryptographic proof that a certain change was made by a certain author at a certain time. New operations can therefore be submitted to a blockchain network as transactions forming new blocks, and operations contained within previous blocks can be retrieved to assemble existing documents.

While the Bitcoin network would not be suitable for this purpose (transactions can take several minutes to be accepted into a block and transaction fees are currently prohibitive), several other blockchain networks exist that would make such a proof-of-concept implementation feasible [152].

As an alternative to a distributed blockchain network, the InterPlanetary File System (IPFS) protocol and network [153] uses blockchain principles (such as Merkle directed acyclic graphs for hashes) to provide access to a "content-addressable" distributed file system with support for browser-accessible peer-to-peer web applications. A peer-to-peer pubsub mechanism has also been enabled [154], which maintains multi-user sessions by using WebRTC, a protocol now supported by all modern web browsers [13]. The pubsub mechanism would allow operations to remain synchronized between all users of a session, as required by the *PubSub Broker* component identified in the feedback-based architecture of Chapter 3.

While Section 2.2.1 mentioned the long history of OT and decentralized approaches, so far very little research exists that combines modern OT principles with blockchain technology (especially at the DOM-based editing level explored in this thesis), and this subject is therefore worth pursuing further. Some aspects of the existing system design (such as the DOM-based operations presented in Section 5.1) can be reused, but a blockchain approach will require revisiting the architecture and Finite State Machine models to ensure that they are adapted into a solution suitable for a decentralized system.

# Appendix A

# Distributed System Simulation API

Listing A.1 contains a description of the `.m` file functions and classes employed by the FSM models of Chapter 6 and implemented for this thesis. The listing includes the signatures of each function and method, thereby detailing the Application Programming Interface (API) of the MATLAB Language layer of the system. Where possible, the official commenting style guide [155] is followed so that the commands `help` and `doc` can be invoked from the MATLAB console on each entity name. In this way, the development of new FSM models for distributed systems is facilitated, as many entities (for example, the static methods of `OTUtil.m`) are reusable across a wide range of systems and domains.

Files are listed in alphabetical order. Files containing unit tests (such as `OTUtilTest.m`) are not listed but are mentioned in the header comment of the corresponding file (in this case, the class `OTUtil.m`). Over 100 unit tests were implemented. The class diagram of Figure 6.25 may also be useful to reference alongside Listing A.1.

```
applyDeleteData.m

% applyDeleteData   Apply 'deleteData' operations to the global state of the given client.
%   This function is meant to be called as extrinsic. Applying the operation will modify
%   the client's global 'localTransMap' to include the change in localOps, thereby
%   modifying 'realDOM' nodes indirectly. This function does not return anything.
%   @param {uint32} cliId                  client ID ('1' for Alice etc.)
%   @param {uint32[]} inputOpsToApplyInts   FullOp including Ops as uint32 array
%   @param {uint32} inputOpIndex            integer of Op index within FullOp to apply
```

```
%   @param {boolean} isRemoteTMTF           boolean indicating if local or remote TM

================================================================================================

applyGiveData.m

% applyGiveData   Apply 'giveData' operations to the global state of the given client.
%   This function is meant to be called as extrinsic. Applying the operation will modify
%   the client's global 'localTransMap' to include the change in localOps, thereby
%   modifying 'realDOM' nodes indirectly. This function does not return anything.
%   @param {uint32} cliId                   client ID ('1' for Alice etc.)
%   @param {uint32[]} inputOpsToApplyInts   FullOp including Ops as uint32 array
%   @param {uint32} inputOpIndex            integer of Op index within FullOp to apply
%   @param {boolean} isRemoteTMTF           boolean indicating if local or remote TM

================================================================================================

applyInsertData.m

% applyInsertData   Apply 'insertData' operations to the global state of the given client.
%   This function is meant to be called as extrinsic. Applying the operation will modify
%   the client's global 'localTransMap' to include the change in localOps, thereby
%   modifying 'realDOM' nodes indirectly. This function does not return anything.
%   @param {uint32} cliId                   client ID ('1' for Alice etc.)
%   @param {uint32[]} inputOpsToApplyInts   FullOp including Ops as uint32 array
%   @param {uint32} inputOpIndex            integer of Op index within FullOp to apply
%   @param {boolean} isRemoteTMTF           boolean indicating if local or remote TM

================================================================================================

applyInsertNode.m

% applyInsertNode   Apply 'insertNode' operations to the global state of the given client.
%   This function is meant to be called as extrinsic. Applying the operation will modify
%   the client's global 'localTransMap' to include the change in localOps, thereby
%   modifying 'realDOM' nodes indirectly. This function does not return anything.
%   @param {uint32} cliId                   client ID ('1' for Alice etc.)
%   @param {uint32[]} inputOpsToApplyInts   FullOp including Ops as uint32 array
%   @param {uint32} inputOpIndex            integer of Op index within FullOp to apply
%   @param {boolean} isRemoteTMTF           boolean indicating if local or remote TM

================================================================================================

clientApplyingOps.m

% clientApplyingOps     Apply a FullOp to the client's extrinsic state.
%   This function is meant to be called as extrinsic and applies a given FullOp received
%   as 'localOpsIntArray' to the local extrinsic state of the specified client.
%   Tests via: disp(table(runtests('clientApplyingOpsTest')));
%   @param {uint32} cliId                   client ID ('1' for Alice etc.)
%   @param {uint32[]} localOpsIntArray      FullOp including Ops as uint32 array
%   @return {uint8} outputOpCodeInt         integer representing the opcode of the op

================================================================================================

```

```
clientApplyingRemoteOpsWithoutACK.m

% clientApplyingRemoteOpsWithoutACK     Apply remote ops via transforming them.
%   This function is meant to be called as extrinsic from the ApplyingRemoteOpsWithoutACK
%   state of OT_Client_Lib and applies operations received as 'remoteOpsIntArray' to the
%   local extrinsic state of the specified client after transforming them. The
%   un-acknowledged local operation is then also transformed for re-sending.
%   Tests via: disp(table(runtests('clientApplyingRemoteOpsWithoutACKTest')));
%   @param {uint32} cliId                   client ID ('1' for Alice etc.)
%   @param {uint32[]} remoteOpsIntArray     FullOp including Ops as uint32 array

================================================================================

clientDefaultTransition.m

% clientDefaultTransition   Initialize the extrinsic states of all clients.
%   This function is meant to be called as extrinsic from the default transition of
%   OT_Client_Lib and initializes realDOM, realMap and prevMap variables for all clients.
%   Nothing is returned.
%   Tests via: disp(table(runtests('clientDefaultTransitionTest')));
%   @param {uint32} cliId                       client ID ('1' for Alice etc.)
%   @param {uint32[]} inputDocIntArray          initial document content as uint32 array

================================================================================

ClientNameEnum.m

% ClientNameEnum     Enumeration class for display-friendly client names.

================================================================================

diffDOM.m

% diffDOM   Obtain a FullOp of differences in the client's extrinsic state of the DOM.
%   This function is meant to be called as extrinsic and uses the global vDOM, prevMap,
%   and realMap variables for a given client ID to generate a FullOp containing ops that
%   describe the observed differences.
%   Tests via: disp(table(runtests('diffDOMTest')));
%   @param {uint32} cliId                   client ID ('1' for Alice etc.)
%   @param {uint32[]} inputLocalOpsInts     FullOp including Ops as uint32 array
%   @return {uint32[]} outputLocalOpsInts   FullOp including Ops as uint32 array

================================================================================

DOMNodeTypeEnum.m

% DOMNodeTypeEnum   Enumeration class for DOM node types.

================================================================================

FullOp.m

    % FullOp    This class creates 'FullOp' objects.
    %   The 'FullOp' format is defined as having the structure [+{},[],{}], where '+'
    %   denotes 'one or more' ops, followed by a payload array, and concluded with a
```

```
120     %   extras structure.
121     %
122     % Tests via: disp(table(runtests('FullOpTest')));
123
124     properties
125         ops             %{Op[]}         array of Op objects within the FullOp
126         payload         %{Payload[]}    array of Payload object within the FullOp
127         extrasTS        %{num}          TS value from extras section
128     end
129
130     methods
131             % FullOp    Construct an instance of this class.
132             %   @param {char[] or uint32[] or Op} inputOne
133             %   @param {Op or Payload} inputTwo
134             %   @param {Op or Payload or num} inputThree
135             %   @param {Payload or num} inputFour
136             %   @param {Payload or num} inputFour
137             %   @return {FullOp} this
138
139             % toChar    Get a character array description of the FullOp object.
140             %   @return {char[]} outputCharArray        character array for disp() output
141
142             % getOp     Retrieve a char array describing an Op at an index in a FullOp.
143             %   @param {uint32[]} inputFullOpIntArray   FullOp as uint32 array
144             %   @param {uint32} inputOpIndex            uint32 of op index to retrieve
145             %   @return {uint32[]} outputOpInts         Op section of FullOp as char array
146
147             % updateOp  Update the 'op' parameter of a FullOp.
148             %   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
149             %   @param {uint32} inputIndexInt           uint32 of requested index
150             %   @param {uint32[]} inputNewOp            uint32 of Op to set
151             %   @return {uint32[]} outputOpInts         FullOp with updated op
152
153             % getFullOpFromOp   Retrieve a char array describing an Op as a FullOp.
154             %   @param {uint32[]} inputOpIntArray       Op as uint32 array
155             %   @return {uint32[]} outputFullOpInts     FullOp as uint32 array
156
157             % getOpsSize    Retrieve the number of ops within a FullOp.
158             %   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
159             %   @return {uint32} outputSizeInt          integer value of number of ops
160
161             % getOpClientId   Retrieve a uint32 client ID from a JSON uint32 array.
162             %   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
163             %   @return {uint32} clientId               client ID ('1' for Alice etc.)
164             %   @usage C = getClientId(A) returns a uint32 value C containing the 'userid'
165             %   value within the op section of the JSON in A.
166
167             % getOpCode     Retrieve the Op code/type for a given Op.
168             %   @param {uint32[]} inputOpIntArray       FullOp as uint32 array
169             %   @param {uint32} inputOpIndex            integer representing the op index
170             %   @return {uint8} outputOpCodeInt         integer representing opcode of op
171
172             % updateOpCode  Update the 'opcode' parameter of a FullOp.
173             %   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
174             %   @param {uint8} inputCodeToSet           uint8 of OpCodeEnum value to set
```

```
175             %   @return {uint32[]} outputOpInts         FullOp with updated 'opcode'
176
177             % getOpCon  Retrieve a given index in the 'opcon' parameter of a FullOp.
178             %   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
179             %   @param {uint32} inputConIndex           zero-indexed 'opcon' param to get
180             %   @return {uint32} outputOpConInt         value of requested index as uint32
181
182             % updateOpCon   Update a given index in the 'opcon' parameter of a FullOp.
183             %   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
184             %   @param {uint32} inputConIndex           zero-indexed 'opcon' param to get
185             %   @param {uint32} inputValueToSet         uint32 with 'opcon' val to set
186             %   @return {uint32[]} inputValueToSet      FullOp with updated 'opcon'
187
188             % getOpLoc  Retrieve an index within the 'loc' parameter of a FullOp.
189             %   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
190             %   @param {uint32} inputLocIndex           zero-indexed 'loc' param index
191             %   @return {uint32} outputOpLocInt         integer value of requested index
192
193             % updateOpLoc   Update the 'loc' parameter of a FullOp.
194             %   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
195             %   @param {uint32} inputCodeToSet          uint32 of new loc value to set
196             %   @return {uint32[]} outputOpInts         FullOp with updated 'oploc'
197
198             % getTS     Retrieve a uint32 TS value within a given JSON uint32 array.
199             %   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
200             %   @return {uint32} outputTSInt            obtained TS value
201             %   @usage C = getTS(A) returns a uint32 value C containing the TS
202             %   value in A.
203
204             % updateTS  Within the given uint32 array, set/update the given TS value.
205             %   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
206             %   @param {uint32} inputLocalTS            desired new TS value to set
207             %   @return {uint32[]} outputIntArray       FullOp with updates TS value
208             %   @usage C = updateTS(A,B) returns a uint32 array C containing the
209             %   modified JSON in A to include the TS value in B.
210     end
211 ================================================================================
212
213 genMapFromDOM.m
214
215 % genMapFromDOM     Obtain a OTMap object from the DOM node input parameter.
216 %   For example, called by 'clientDefaultTransition.m' and 'clientApplyingOps.m'.
217 %   Tests via: disp(table(runtests('genMapFromDOMTest')));
218 %   @param {ElementNSImpl} inputDOM         node object for which to generate the OTMap
219 %   @return {OTMap} otmapToReturn           OTMap object for the node and its children
220
221 % recursiveMapGen     Recursively include the 'inputNode' DOM into an 'inputMap' OTMap.
222 %   This function assumes the root element node is handled by 'genMapFromDOM()' and
223 %   appends the rest of the nodes.
224 %   @param {ElementNSImpl} inputNode        node object for which to generate the OTMap
225 %   @param {OTMap} inputMap                 existing OTMap object to append nodes onto
226 %   @return {OTMap} outputMap               OTMap object for the node and its children
227
228 ================================================================================
229
```

```
genTransMap.m

% genTransMap   Generate a transitional OTMap object from an existing OTMap and JSON.
%   For example, called by 'clientApplyingOps.m'.
%   TODO: Technically just need to receive the last array position of the JSON since
%   that's where all inner payload arrays are. This would require passing an array of
%   'Payload' objects.
%   Tests via: disp(table(runtests('genTransMapTest')));
%   @param {uint32} cliId                   client ID ('1' for Alice etc.)
%   @param {OTMap} inputMap                 existing OTMap object to append nodes onto
%   @param {uint32[]} inputOpsIntArray      FullOp as uint32 array
%   @return {OTMap} resultingMap            transitional OTMap object

================================================================================

getApplySummary.m

% getApplySummary   Display a summary after an operation was applied.
%   This function is meant to be called as extrinsic.
%   @param {uint32} cliId                   client ID ('1' for Alice etc.)
%   @param {Op} inputOpObj                  Op object to access
%   @param {boolean} includeRemoteTF        boolean if to show the remote TM value

================================================================================

getRealDOM.m

% getRealDOM   Retrieve the realDOM for a given client ID.
%   This function is meant to be called as extrinsic.
%   @param {uint32} cliId                   client ID ('1' for Alice etc.)

================================================================================

getSyncedSummary.m

% getSyncedSummary  Display the client's extrinsic state while in the Synced state.
%   This function is meant to be called as extrinsic from the Synced state of
%   OT_Client_Lib so that the disp() statement has access to the global 'realDOM' variable
%   in order to generate a useful output. Nothing is returned.
%   @param {uint32} cliId                    client ID ('1' for Alice etc.)

================================================================================

getXformApplySummary.m

% getXformApplySummary  Display a summary after a transformation result was applied.
%   This function is meant to be called as extrinsic.
%   @param {uint32} cliId                    client ID ('1' for Alice etc.)

================================================================================

MultiOp.m

    % MultiOp   Static methods to manipulate concatenated FullOps as characters.
    %
```

```
285     % Tests via: disp(table(runtests('MultiOpTest')));
286
287     methods (Static)
288             % addOp    Append an op to an existing MultiOp.
289             %   This function is likely called from 'OT_Client_Lib' to get a 'MultiOp'.
290             %   @param {uint32[]} inputMultiOpIntArray  input MultiOp uint32 to append to
291             %   @param {uint32[]} inputNewOpIntArray    input FullOp uint32 to append
292             %   @return {uint32[]} outputIntArray       MultiOp as a uint32 array
293
294             % getSize   Retrieve the number of FullOps within a given MultiOp.
295             %   This function is likely called from 'OT_Client_Lib'.
296             %   @param {uint32[]} inputIntArray     input MultiOp uint32 with FullOps
297             %   @return {uint32} outputInt          number of FullOps found
298
299             % getWithoutFirst    Remove first FullOp from MultiOp, return MultiOp.
300             %   This function is likely called from 'OT_Client_Lib' after got ACK for
301             %   previous local operation and need to form new op with oldest buffer
302             %   contents in 'CreatingLocalOpFromBuffer' state. We remove oldest item from
303             %   the buffer since will be sending it as localOps via the next state
304             %   ('SendingOpsToController').
305             %   @param {uint32[]} inputIntArray     input MultiOp uint32 with FullOps
306             %   @return {uint32} outputIntArray     MultiOp with first FullOp removed
307
308             % updateOpCode     Update OpCode of the FullOp at given MultiOp index.
309             %   This function is meant to be called as extrinsic via
310             %   transformation-related logic in the 'ApplyingRemoteOpsWithBuffer' state of
311             %   'OT_Client_Lib'.
312             %   @param {uint32[]} inputOpsIntArray      MultiOp including FullOps as
313             %                                           uint32 array
314             %   @param {uint32} inputIndexInt           uint32 of requested MultiOp index
315             %                                           (using one-based indexing)
316             %   @param {uint8} inputCodeToSet           uint8 of OpCodeEnum value to set
317             %   @return {uint32[]} outputOpInts         MultiOp with updated 'opcon'
318             %                                           parameter
319
320             % updateFullOpAtIndex    Update FullOp of the MultiOp at a given index.
321             %   This function is meant to be called as extrinsic via
322             %   transformation-related logic in the 'ApplyingRemoteOpsWithBuffer' state of
323             %   'OT_Client_Lib'.
324             %   @param {uint32[]} inputOpsIntArray      MultiOp including FullOps as
325             %                                           uint32 array
326             %   @param {uint32} inputIndexInt           uint32 of requested MultiOp index
327             %                                           (using one-based indexing)
328             %   @param {uint8} inputOpToSet             uint32 array of FullOp to set
329             %   @return {uint32[]} outputOpInts         MultiOp with updated 'opcon'
330             %                                           parameter
331
332             % getFullOp  Get uint32 array of a single FullOp section.
333             %   This static method is meant to be called as from the 'ObtainingOp' state
334             %   of the 'Events Generator' chart in 'OT_Control_Loop' and returns a
335             %   'FullOp' subset from the received 'MultiOp'.
336             %   @param {uint32[]} inputIntArray     input MultiOp uint32 with FullOps
337             %   @param {uint32} inputIndexInt       uint32 of requested MultiOp index
338             %                                       (using one-based indexing)
339             %   @return {uint32[]} outputIntArray   one FullOp as a uint32 array
```

```
    end

================================================================================================

Op.m
    % Op    This class creates 'Op' objects via their JSON descriptions.
    %
    % Tests via: disp(table(runtests('OpTest')));

    properties
        userid  %{num}      client ID ('1' for Alice etc. as per ClientNameEnum)
        opcode  %{char[]}   operation type ('insertData' etc. as per OpCodeEnum)
        opcon   %{OpCon}    operation content (character to insert etc.)
        loc     %{OpLoc}    typically reference to index within transitional map
    end

    methods
            % Op     Construct an instance of this class.
            %   @param {num} useridin           see Op.userid
            %   @param {char[]} opcodein        see Op.opcode
            %   @param {OpCon} opconin          see Op.opconin
            %   @param {OpLoc} locin            see Op.locin
            %   @return {Op} this               see Op

            % toChar    Get a character array description of the Op object.
            %   @return {char[]} outputCharArray     character array for disp() output
    end

================================================================================================

OpCon.m
    % OpCon     This class creates 'OpCon' objects via their JSON descriptions.
    %   OpCon, short for 'Operation Content', typically includes operation data such as
    %   the letter to be inserted and the position at which to do the insertion.
    %
    % Tests via: disp(table(runtests('FullOpTest/Test1')));

    properties
        cons    %{string[]}  array of strings, each containing a piece of op content
    end

    methods
            % OpCon     Construct an instance of this class.
            % Patterns: OneInt, OneStr, OneStrOneInt, SixStr, ThreeStr, TwoStr, Zero
            %   @param {char[]} one         first piece of content character array
            %   @param {char[]} two         second piece of content character array
            %   @param {char[]} three       third piece of content character array
            %   @param {char[]} four        fourth piece of content character array
            %   @param {char[]} five        fifth piece of content character array
            %   @param {char[]} six         sixth piece of content character array
            %   @return {OpCon} this        see OpCon
            %   @usage insertData of letter 'b' at position 1: conObj = OpCon('b',1);

            % get   Retrieve a number from the OPCon object. Zero-indexed.
            %   @param {num} index                      integer of index to retrieve
```

```
395             %   @returns {char[]} outputCharArray       char array value at index @usage
396             %   get(A) retrieves the char array of the string at index A from the OPCon
397             %   array.
398 
399             % toChar    Get a character array description of the OpCon object.
400             %   @return {char[]} outputCharArray     character array for disp() output
401     end
402 
403 ================================================================================
404 
405 OpLoc.m
406 
407     % OpLoc     This class creates 'OpLoc' objects via their JSON descriptions.
408     %   OpLoc, short for Operation Location, typically includes a reference to a source or
409     %   destination index in the transitional map.
410     %
411     % Tests via: disp(table(runtests('FullOpTest/Test1')));
412 
413     properties
414         locs    %{num[]}    array of integers, each index containing a location parameter
415     end
416 
417     methods
418             % OpLoc     Construct an instance of this class.
419             % Patterns: One, Two, Zero
420             %   @param {num} one            first location parameter integer
421             %   @param {num} two            second location parameter integer
422             %   @return {OpLoc} this        see OpLoc
423             %   @usage insertData at index 2 of transMap: locObj = OpLoc(2);
424 
425             % get   Retrieve a number from the OPLoc object. Zero-indexed.
426             %   @param {num} index           integer of index to retrieve
427             %   @returns {num} outputInt     integer value at index
428             %   @usage get(A) retrieves the integer at index A from the OPLoc array.
429 
430             % toChar  Get a character array description of the OpLoc object.
431             %   @return {char[]} outputCharArray     character array for disp() output
432             % TODO: one function that share with OpCon.toChar that knows if to put quotes
433     end
434 
435 ================================================================================
436 
437 OTConst.m
438 
439     % OTConst   Useful constants for tests, hardcoded setups etc. across multiple files.
440 
441 ================================================================================
442 
443 OTGenerator.m
444 
445     % OTGenerator   Fixed and random operation generation.
446 
447             % getRandomOps  Get a uint32 array containing an operation in JSON format.
448             %   @return {uint32[]} outputOpsInts            MulitOp uint32 array
449             %   @return {uint32} outputOpsTotInt            uint32 with number of FullOps
```

```
450             %                                           contained within the MultiOp
451             %   @usage A = getRandomOps() returns a uint32 array A containing a MultiOp
452             %   that describes one or more operations.
453             
454             % generateOpEx  Generate a random operation for the given clientID.
455             %   This function is meant to be called as extrinsic.
456             %   @param {num} cliId                      client ID ('1' for Alice etc.)
457             %   @param {boolean} newRoundTF             boolean if need adjustment reset
458             %   @return {char[]} outputOpsAsChars       character array of output ops
459             %   @usage The following command will display an operation for user Alice:
460             %       clear functions; disp(OTGenerator.generateOpEx(1, true));
461             %   In this example, the "clear functions;" command is used to reset
462             %   persistent variables.
463             
464             % getStaticOps  Get a uint32 array of ops for a uint32 index into 'testList'.
465             %   @param {uint32} inputInt                    uint32 of the test number
466             %   @return {uint32[]} outputOpsInts            MulitOp uint32 array
467             %   @return {uint32} outputOpsTotInt            uint32 with number of FullOps
468             %                                               contained within the MultiOp
469             %   @return {uint32[]} outputExpectedDocInts    uint32 array used to check
470             %                                               for output correctness
471             %   @usage C = OTGenerator.getStaticOps(A) returns a uint32 array C containing
472             %   the MultiOp that describes the operations for the test at index 'A' in
473             %   'OTConst.testList'.
474             
475             % isConsistent  Check if the Simulink client block outputs are the same.
476             %   This function is meant to be called from the self-transition
477             %   in the 'Events Generator' chart in 'OT_Control_Loop'. It returns true or
478             %   false depending on if the client output states match.
479             %   No tests exist since this code requires a live running simulation.
480             %   @param {uint32[]} inputExpectedInts     expected document contents
481             
482             % isRandomMode  Check if the 'OT_Control_Loop' switch is set to 'RANDOM'.
483             %   This function is meant to be called from the self-transition in the
484             %   'Client State Machine' and 'Controller' charts. It returns true or false
485             %   depending on if the switch is set to the 'RANDOM' position at runtime.
486             %   No tests exists since this code requires a live running simulation.
487             %   @return {boolean} outputTF          boolean if the switch is on 'RANDOM'
488             
489             % isSwitchInputRandom   Check if the input uint32 array is the word 'RANDOM'.
490             %   @param {uint32} inputIntArray       uint32 array to check
491             %   @return {boolean} outputTF          boolean if the input is 'RANDOM'
492             
493 ================================================================================
494 
495 OTMap.m
496 
497     % OTMap     This class is used to create 'Map' objects (arrays of nodes).
498     %   A local cell array is used to store the provided nodes.
499     %   TODO: 'OTMap.clone()' for use via 'clientDefaultTransition.m'
500     %
501     % Tests via: disp(table(runtests('genMapFromDOMTest')));
502     
503     properties (Access = private)
504         nodeList    %{ElementNSImpl[]}  node object array with nodes in map order
```

```
505     end
506
507     methods
508             % OTMap     Construct an instance of this class.
509             %   @return {OTMap} this        see OTMap
510
511             % add   Add a node to the OTMap object.
512             %   @param {ElementNSImpl} inputNode    node object to add to the local array
513             %   @usage add(A) adds the node A to the end of the nodeList array and returns
514             %   nothing.
515
516             % get   Retrieve a node from the OTMap object. Uses zero-based indexing.
517             %   @param {num} index                      integer of index to retrieve
518             %   @returns {ElementNSImpl} outputNode     node object at index
519             %   @usage get(A) retrieves the node at index A from the OTMap.
520
521             % size  Retrieve a node from the OTMap object.
522             %   @returns {num} outputSize     integer size/length of local nodeList array
523             %   @usage get(A) retrieves the node at index A from the OTMap.
524
525             % indexOf   Retrieve the index of a given node from the OTMap object.
526             %   @param {ElementNSImpl} inputNode    node object to add to the local array
527             %   @returns {num} outputIndex          integer of index where node found
528             %   @usage C = get(A) returns the index C of node A if it is found in the
529             %   OTMap, and returns 0 if it is not found.
530
531             % getNumberOfNodesOfType    Find the number of nodes of a given type.
532             %   @param {num} inputNodeType      node type number ('1' for element nodes,
533             %                                   '3' for text nodes etc.)
534             %   @returns {num} outputNumber     integer of number of such nodes found
535             %   @usage C = get(A) returns the number of nodes C of a given type A found in
536             %   the OTMap.
537
538             % getNthNodeTypePos     Retrieve the index at a given occurence of given type.
539             %   @param {num} inputNodeType      node type number ('1' for element nodes,
540             %                                   '3' for text nodes etc.)
541             %   @param {num} nodeNum            Nth occurence interested in obtaining
542             %                                   the index of ('1' for first)
543             %   @returns {num} outputIndex      integer of index of requested node
544             %   @usage Use 'getNthNodeTypePos(3,1)' to obtain the index of the 0th text
545             %   node (type 3).
546
547             % toChar    Get a character array description of the OTMap object.
548             %   @return {char[]} outputCharArray     character array for disp() output
549     end
550
551 ================================================================================
552
553 OTUtil.m
554
555     % OTUtil    Assorted useful static utility methods.
556     %   String, casting and other useful methods grouped into one place. A 'MAX_UINT32'
557     %   variable is expected in the workspace. 'MXA' is used as shorthand for 'MXArray'.
558     %
559     % Tests via: disp(table(runtests('OTUtilTest')));
```

```
560
561     methods (Static)
562             % getBoolFromMXA    Get a boolean/logical value from a '1x1' mxArray variable.
563             %   @param {mxArray} inputMXA           input boolean in mxArray form
564             %   @returns {boolean} convertedBool    output boolean/logical value
565             %   @usage C = getBoolFromMXA(A) returns a boolean value C by casting A
566             %   into a 1x1 boolean memory location.
567
568             % getIntFromFixedMXA    Get a uint32 value from a '1x1' mxArray variable.
569             %   @param {mxArray} inputMXA           input mxArray that can store as uint32
570             %   @returns {uint32} convertedInt      output uint32 value
571             %   @usage C = getIntFromFixedMXA(A) returns a uint32 value C by casting A
572             %   into a 1x1 uint32 memory location.
573
574             % getIntFromVarMXA  Get a uint32 from a variable-sized mxArray (size '1x:?').
575             %   @param {mxArray} inputMXA           input mxArray that can store as uint32
576             %   @returns {uint32} convertedInt      output uint32 value
577             %   @usage C = getIntFromVarMXA(A) returns a uint32 value C by casting A into
578             %   a 1x1 uint32 memory location.
579
580             % getIntFromNumericChars    Get a uint32 from multiple uint32 array digits.
581             %   This function is useful for receiving a number represented as multiple
582             %   characters, such as from a 'String Constant' block in Simulink.
583             %   @param {uint32[]} inputIntArray     input uint32 array with number
584             %   @returns {uint32} convertedInt      output uint32 value
585             %   @usage C = getIntFromNumericChars(A) returns a uint32 value C by
586             %   converting the digits represented by the array A into a number.
587
588             % getUnpaddedIntFromIntArray    Get uint32 from padded uint32 array.
589             %   @param {uint32[]} inputIntArray      input uint32 array to be shortened
590             %   @returns {uint32} outputInt          output uint32 value from array head
591             %   @usage C = getUnpaddedIntFromIntArray(A) returns the first uint32 value C
592             %   by trimming the zeros from the end of the uint32 input array A.
593
594             % getUnpaddedIntsFromIntArray    Get uint32 array from padded uint32 array.
595             %   @param {uint32[]} inputIntArray      input uint32 array to be shortened
596             %   @returns {uint32[]} outputIntArray   output uint32 array value from head
597             %   @usage C = getUnpaddedIntsFromIntArray(A) returns a uint32 array value C
598             %   by trimming the zeros from the end of the uint32 input array A.
599
600             % getPaddedCharArrayFromMXA     Get a padded character array (fixed length).
601             %   @param {mxArray} inputMXA               input mxArray that contains chars
602             %   @returns {char[]} convertedCharArray    output padded character array
603             %   @usage C = getPaddedCharArrayFromMXA(A) returns an array of characters by
604             %   stretching the input mxArray into MAX_UINT32 characters.
605
606             % getCharArrayFromMXA   Get a character array (unpadded) from a mxArray.
607             %   @param {mxArray} inputMXA               input mxArray that contains chars
608             %   @returns {char[]} convertedCharArray    output unpadded character array
609             %   @usage C = getCharArrayFromMXA(A) returns an array of characters based on
610             %   the contents of mxArray A.
611
612             % getCharsFromInts  Get a character array (unpadded) from a uint32 array.
613             %   @param {uint32[]} inputIntArray         input uint32 array usable as chars
614             %   @returns {char[]} convertedCharArray    output unpadded character array
```

```
615           %   @usage C = getCharsFromInts(A) returns an array of characters based on
616           %   the contents of A.
617           %
618           % padCharArray  Get a character array of a fixed length back.
619           %   This is useful for obtaining a fixed length character array after casting
620           %   to uint32.
621           %   @param {char[]} usefulCharArray         input char array
622           %   @returns {char[]} paddedCharArray       output padded character array
623           %   @usage C = padCharArray(A) streches out the character array A to be of
624           %   MAX_UINT32 in size and returns it as character array C.
625           %
626           % inputIntArray  Get a uint32 array of fixed length back from an input uint32.
627           %   This is useful for obtaining a fixed length uint32 array for tests.
628           %   @param {uint32[]} inputIntArray          input uint8 or uint32 array
629           %   @returns {uint32[]} paddedIntArray       output uint32 array of fixed
630           %                                            length
631           %   @usage C = inputIntArray(A) streches out the uint32 array A to be of
632           %   MAX_UINT32 in size and returns it as uint32 array C.
633           %
634           % getPaddedIntsFromCharArray  Get a character array of a fixed length back.
635           %   This is useful for obtaining a fixed length after casting to uint32.
636           %   @param {char[]} inputCharArray          input char array
637           %   @returns {uint32[]} paddedIntArray      output padded integer array
638           %
639           % ensureSystemLoaded    Verify that OTConst.systemName is the active system.
640           %   The system defined via 'OTConst.systemName' is loaded if it is not open
641           %   already to ensure that system properties can be queried. Returns nothing.
642           %
643           % getNumberOfClients    Get the number of clients in the system.
644           %   This static method must be marked as extrinsic before calling. Requires
645           %   model to be open (automatically opened otherwise).
646           %   @returns {num} numberOfClis         number of clients
647           %   @usage C = getNumberOfClients() returns the value C with the number of
648           %   clients.
649           %
650           % getCharsNameFromId    Get a name like 'Alice' from an integer like '1'.
651           %   Useful at extrinsic code level.
652           %   @param {num} inputIdInt                          input integer to convert
653           %   @return {char[]} outputClientNameCharArray      output unpadded char array
654           %   @usage C = getCharsNameFromId(A) returns a character array C containing a
655           %   name corresponding to the number in integer input A. ClientNameEnum is
656           %   used to make the correlation.
657           %
658           % getPaddedIntsNameFromId   Get padded uint32 array 'Alice' from int '1' etc.
659           %   Useful at Stateflow chart level.
660           %   @param {uint32} inputIdInt                       input uint32 to convert
661           %   @return {uint32[]} outputClientNameIntArray      output padded uintarray
662           %   C = getPaddedIntsNameFromId(A) returns a uint32 array C containing
663           %   the name corresponding to the number in uint32 integer input A.
664           %   ClientNameEnum is used to make the correlation.
665           %
666           % delSpaces     Remove space characters from a given character array input.
667           %   @param {char[]} inputCharArray      input char array containing spaces
668           %   @return {char[]} outputCharArray    output unpadded char array with space
669           %                                       chars removed
```

```
670             %   @usage The following example will return the character array 'abc':
671             %       getCharsFieldFromJSON(' a b  c  ');
672             %
673             % runSimulation     Start console-based simulation using latest 'stop time'.
674             %   Returns nothing.
675             %
676             % getSimulationStopTime     Retrieve the latest system 'stop time'.
677             %   @return {uint32} outputStopTimeInt      the simulation stop time integer
678             %
679             % setInitialContent     Set the value of the 'Initial Document Content' block.
680             %   All clients will grab the latest block value when initializing. Nothing is
681             %   returned by this method.
682             %   @param {char[]} inputDocChars           char array of document content
683             %
684             % getNumberOfFullOpsInMultiOp   Get the number of FullOps in a given MultiOp.
685             %   @param {char[]} inputMultiOpChars       MultiOp char array
686             %   @return {uint32[]} outputOpsInts        uint32 containing the discovered
687             %                                           number of FullOps
688             %
689             % rndUpToIncl   Get a random number from 1 up to and including the max input.
690             %   @param {num} inputMaxNum                input upper number
691             %   @return {uint32} outputRndInt           resulting random number
692             %   @usage The following example will return a dice roll (1 to 6 inclusive):
693             %       OTUtil.rndUpToIncl(6);
694             %
695             % appendInts    Append a source uint32 array into a target uint32 array.
696             %   @param {uint32[]} inputSourceInts     input uint32 array to append
697             %   @param {uint32[]} inputTargetInts     input uint32 array to append into
698             %   @return {uint32[]} outputInts         output padded uint32 array
699             %
700             % appendBufferInts  Assemble buffer-specific log message section.
701             %   This static method is meant to be called by the
702             %   'ApplyingRemoteOpsWithBuffer' state of 'OT_Client_Lib'.
703             %   @param {uint32[]} inputSourceInts     input uint32 array to append
704             %   @param {uint32[]} inputTargetInts     input uint32 array to append into
705             %   @return {uint32[]} outputInts         output padded uint32 array
706             %
707             % cleanForDisp  Get a console-safe string of the input (fix XML brackets).
708             %   @param {char[]} inputChars              input chars to process
709             %   @returns {char[]} outputCharArray       output with '<' and '>' symbols
710             %                                           replaced
711             %   @usage C = getNodeAsCharsForDisp(A) assembles the input node A into a
712             %   one-row string from the document level and returns it as character array
713             %   C.
714         end
715
716     ================================================================================
717
718     OTXML.m
719
720         % OTXML     XML-related functions.
721         %
722         % Tests via: disp(table(runtests('genMapFromDOMTest')));
723
724         methods (Static)
```

```
            % getNodeAsChars    Get a whitespace-less char array of the document XML.
            %   This static method is ok to use for sending to Simulink Display blocks,
            %   but see 'OTUtil.cleanForDisp()' if using in 'disp()' statements.
            %   @param {ElementNSImpl} inputXMLNode     input XML node to convert
            %   @returns {char[]} outputCharArray       unpadded XML char array containing
            %                                           full XML document
            %   @usage C = getNodeAsChars(A) assembles the input node A into a one-row
            %   string from the document level and returns it as character array C.

            % getNodeAsPaddedInts   Get a padded uint32 array with the XML of the node.
            %   @param {char[]} inputXMLNode            input XML node as char array
            %   @returns {uint32[]} outputIntArray      padded single-line uint32 array
            %   @usage C = getNodeAsPaddedInts(A) returns the uint32 array C by padding
            %   the character representation of the XML node A.

            % getNodeAsCharsMapFormat   Get a char array in the "map format" for a node.
            %   @param {char[]} inputXMLNode         input XML node as char array
            %   @returns {char[]} outputCharArray    unpadded char array in map format
            %   @usage C = getNodeAsCharsMapFormat(A) for the body node A forming the XML
            %   <body><p>abc</p></body> means that C will be the character array
            %   '[body, p, "abc"]'.
    end

================================================================================

Payload.m

    % Payload   This class creates 'Payload' objects via their JSON descriptions.

    properties
        parentInMap     %{boolean}      true=parent already in map, false=new inclusion
        nodeType        %{num}          node type (1=element etc.)
        nodeValue       %{char[]}       string with tag name or text data for text nodes
        opPayloadAux1   %{PayloadAux}   attribs for ele ([""] if none), parent idx for txt
        opPayloadAux2   %{PayloadAux}   array with parent index at [0] for element nodes
    end

    methods
            % Payload    Construct an instance of this class.
            % Patterns: Zero, Three, Four, Five
            %   @param {boolean} parentInMapInput           see Payload.parentInMap
            %   @param {num} nodeTypeInput                  see Payload.nodeType
            %   @param {char[]} nodeValueInput              see Payload.nodeValue
            %   @param {PayloadAux} opPayloadAux1Input      see Payload.opPayloadAux1
            %   @param {PayloadAux} opPayloadAux2Input      see Payload.opPayloadAux2
            %   @return {Payload} this                      see Payload
            %   @usage payloadObj = Payload(true,1,"STRONG",payloadAuxObj);

            % toChar  Get a character array description of the Payload object.
            %   @return {char[]} outputCharArray         character array for disp() output
    end

================================================================================

PayloadAux.m
```

```
780
781     % PayloadAux  This class creates 'PayloadAux' objects via their JSON descriptions.
782
783     properties
784         auxVals     %{string[]}   array of strings, each string containing an aux value
785     end
786
787     methods
788             % PayloadAux    Construct an instance of this class.
789             % Patterns: Zero, OneStr, TwoInt, TwoStr
790             %   @param {char[]} one        first aux parameter character array
791             %   @param {char[]} two        second aux parameter character array
792             %   @return {PayloadAux} this  see PayloadAux
793             %   @usage payloadAuxObj = PayloadAux();
794
795             % toChar  Get a character array description of the PayloadAux object.
796             %   @return {char[]} outputCharArray         character array for disp() output
797     end
798
799 ================================================================================
800
801 xformGiveDataVsDeleteData.m
802
803 % xformGiveDataVsDeleteData   Transform a target giveData vs. reference deleteData.
804 %   This function is meant to be called as extrinsic.
805 %   @param {uint32} cliId                    client ID ('1' for Alice etc.)
806 %   @param {uint32[]} inputTargetOpInts        FullOp including Ops as uint32 array
807 %   @param {uint32[]} inputRefOpInts        FullOp including Ops as uint32 array
808 %   @return {uint32[]} outputXformedOpInts   FullOp including Ops as uint32 array
809
810 ================================================================================
811
812 xformGiveDataVsDeleteData.m
813
814 % xformInsertNodeVsDeleteData   Transform a target insertNode vs. reference deleteData.
815 %   This function is meant to be called as extrinsic.
816 %   @param {uint32} cliId                    client ID ('1' for Alice etc.)
817 %   @param {uint32[]} inputTargetOpInts        FullOp including Ops as uint32 array
818 %   @param {uint32[]} inputRefOpInts        FullOp including Ops as uint32 array
819 %   @return {uint32[]} outputXformedOpInts   FullOp including Ops as uint32 array
```

Listing A.1: Descriptions of .m file functions and classes used in MATLAB models of Chapter 6.

# Appendix B

# Simulation Execution Logs

The Simulink console logs obtained after executing a simulation based on input from the *Events Generator* offer valuable insight into the functionality of the real-time architecture introduced in Chapter 3, the validity of the FSM-based CLOT algorithm developed in Chapter 4, and the DOM-based operations presented in Chapter 5. This appendix contains the full logs of the scenarios that were previously described using smaller snippets in Chapter 7.

## B.1 Basic OT FSM

```
1  SENDING [{"userid":1,"opcode":"i"},{}] TO: Alice
2  Alice in Synced with: abc
3  Bob in Synced with: abc
4  Controller in Listening
5  Alice typed and received Local_Change with [{"userid":1,"opcode":"i"},{}] while in Synced
6  Alice in ApplyingLocalOps, Alice.localTS is now 2
7  SENDING [{"userid":2,"opcode":"i"},{}] TO: Bob
8  Bob typed and received Local_Change with [{"userid":2,"opcode":"i"},{}] while in Synced
9  Bob in ApplyingLocalOps, Bob.localTS is now 2
10 Alice sent CtoS_Msg with [{"userid":1,"opcode":"i"},{"TS":3}] via SendingOpsToController
11 Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"i"},{"TS":3}] while in
       Listening
12 Controller determined TS from Alice is new and reached PersistingNew, Controller.localTS is now 3
13 Alice in AwaitingACK
14 Bob sent CtoS_Msg with [{"userid":2,"opcode":"i"},{"TS":3}] via SendingOpsToController
15 Controller sent StoC_ACK to Alice via SendingACKToClient
16 Alice in Synced with: abc
17 Bob in AwaitingACK
18 Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"i"},{"TS":3}] to Bob via
       SendingToRemainingClients
19 Bob received signal StoC_Msg while in AwaitingACK (CONFLICT RESOLUTION REQUIRED!)
```

```
20 Bob in ApplyingRemoteOpsWithoutACK, Bob.localTS is now 4
21 Controller in Listening
22 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"i"},{"TS":3}] while in Listening
23 Controller determined TS from Bob is old (REJECTED!)
24 Controller in Listening
25 Bob transforms remoteOps [{"userid":1,"opcode":"i"},{"TS":3}] against localOps [{"userid":2,"
       opcode":"i"},{}] to get [{"userid":1,"opcode":"i"},{"TS":3}] and applies it, Bob's document
       is now abc
26 Bob performed the transformation with reversed parameter sequence and obtained [{"userid":2,"
       opcode":"i"},{}]
27 Bob sent CtoS_Msg with [{"userid":2,"opcode":"i"},{"TS":4}] via SendingOpsToController
28 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"i"},{"TS":4}] while in Listening
29 Controller determined TS from Bob is new and reached PersistingNew, Controller.localTS is now 4
30 Bob in AwaitingACK
31 Controller sent StoC_ACK to Bob via SendingACKToClient
32 Bob in Synced with: abc
33 Controller sent signal StoC_Msg with data [{"userid":2,"opcode":"i"},{"TS":4}] to Alice via
       SendingToRemainingClients
34 Alice received StoC_Msg with [{"userid":2,"opcode":"i"},{"TS":4}] while in Synced
35 Alice in ApplyingRemoteOps, Alice.localTS is now 4
36 Controller in Listening
37 Alice in Synced with: abc
38 CONSISTENT AND CORRECT!
```

Listing B.1: Full console output during FSM execution of a basic transformation scenario between two identity operations (test "2").

## B.2 Basic OT FSM with Buffer

```
1  SENDING [{"userid":1,"opcode":"i"},{}] TO: Alice
2  Alice in Synced with: abc
3  Bob in Synced with: abc
4  Controller in Listening
5  Alice typed and received Local_Change with [{"userid":1,"opcode":"i"},{}] while in Synced
6  Alice in ApplyingLocalOps, Alice.localTS is now 4
7  SENDING [{"userid":1,"opcode":"i"},{}] TO: Alice
8  Alice sent CtoS_Msg with [{"userid":1,"opcode":"i"},{"TS":5}] via SendingOpsToController
9  Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"i"},{"TS":5}] while in
       Listening
10 Controller determined TS from Alice is new and reached PersistingNew, Controller.localTS is now 5
11 Alice in AwaitingACK
12 Controller sent StoC_ACK to Alice via SendingACKToClient
13 Alice typed and received signal Local_Change with buffer op [{"userid":1,"opcode":"i"},{}] while
       in AwaitingACK
14 Alice in ApplyingBufferedLocalOps with buffered op [{"userid":1,"opcode":"i"},{}]
15 Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"i"},{"TS":5}] to Bob via
       SendingToRemainingClients
16 Bob received StoC_Msg with [{"userid":1,"opcode":"i"},{"TS":5}] while in Synced
17 Bob in ApplyingRemoteOps, Bob.localTS is now 5
18 Controller in Listening
19 Alice in AwaitingWithBuffer with buffer size 1
```

```
Alice received signal StoC_ACK while in AwaitingWithBuffer
Alice in CreatingLocalOpFromBuffer, Alice.localTS is now 6
Bob in Synced with: abc
Alice sent CtoS_Msg with [{"userid":1,"opcode":"i"},{"TS":6}] via SendingOpsToController
Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"i"},{"TS":6}] while in
    Listening
Controller determined TS from Alice is new and reached PersistingNew, Controller.localTS is now 6
Alice in AwaitingACK
Controller sent StoC_ACK to Alice via SendingACKToClient
Alice in Synced with: abc
Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"i"},{"TS":6}] to Bob via
    SendingToRemainingClients
Bob received StoC_Msg with [{"userid":1,"opcode":"i"},{"TS":6}] while in Synced
Bob in ApplyingRemoteOps, Bob.localTS is now 6
Controller in Listening
Bob in Synced with: abc
CONSISTENT AND CORRECT!
```

Listing B.2: Full console output during FSM execution of a basic buffer scenario with two identity operations (test "3").

```
SENDING [{"userid":1,"opcode":"i"},{}] TO: Alice
Alice in Synced with: abc
Bob in Synced with: abc
Controller in Listening
Alice typed and received Local_Change with [{"userid":1,"opcode":"i"},{}] while in Synced
Alice in ApplyingLocalOps, Alice.localTS is now 6
SENDING [{"userid":2,"opcode":"i"},{}] TO: Bob
Bob typed and received Local_Change with [{"userid":2,"opcode":"i"},{}] while in Synced
Bob in ApplyingLocalOps, Bob.localTS is now 6
Alice sent CtoS_Msg with [{"userid":1,"opcode":"i"},{"TS":7}] via SendingOpsToController
Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"i"},{"TS":7}] while in
    Listening
Controller determined TS from Alice is new and reached PersistingNew, Controller.localTS is now 7
SENDING [{"userid":2,"opcode":"i"},{}] TO: Bob
Alice in AwaitingACK
Bob sent CtoS_Msg with [{"userid":2,"opcode":"i"},{"TS":7}] via SendingOpsToController
Controller sent StoC_ACK to Alice via SendingACKToClient
Alice in Synced with: abc
Bob in AwaitingACK
Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"i"},{"TS":7}] to Bob via
    SendingToRemainingClients
Bob typed and received signal Local_Change with buffer op [{"userid":2,"opcode":"i"},{}] while in
    AwaitingACK
Bob in ApplyingBufferedLocalOps with buffered op [{"userid":2,"opcode":"i"},{}]
Controller in Listening
Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"i"},{"TS":7}] while in Listening
Controller determined TS from Bob is old (REJECTED!)
Controller in Listening
Bob in AwaitingWithBuffer with buffer size 1
Bob received signal StoC_Msg while in AwaitingWithBuffer (CONFLICT RESOLUTION WITH BUFFER!)
Bob in ApplyingRemoteOpsWithBuffer, remoteOps is [{"userid":1,"opcode":"i"},{"TS":7}], Bob.localTS
     is now 8
```

```
29    That is, Bob transforms remoteOps [{"userid":1,"opcode":"i"},{"TS":7}] against localOps [{"
      userid":2,"opcode":"i"},{}] to get [{"userid":1,"opcode":"i"},{"TS":7}] and applied it, Bob's
       document is now abc
30  Bob performed the transformation with reversed parameter sequence and obtained [{"userid":2,"
      opcode":"i"},{}] as the updated ACK that awaiting
31  Bob sent CtoS_Msg with [{"userid":2,"opcode":"i"},{"TS":8}] via SendingOpsToController
32  Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"i"},{"TS":8}] while in Listening
33  Controller determined TS from Bob is new and reached PersistingNew, Controller.localTS is now 8
34  Bob in AwaitingWithBuffer with buffer size 1
35  Controller sent StoC_ACK to Bob via SendingACKToClient
36  Bob received signal StoC_ACK while in AwaitingWithBuffer
37  Bob in CreatingLocalOpFromBuffer, Bob.localTS is now 9
38  Controller sent signal StoC_Msg with data [{"userid":2,"opcode":"i"},{"TS":8}] to Alice via
      SendingToRemainingClients
39  Alice received StoC_Msg with [{"userid":2,"opcode":"i"},{"TS":8}] while in Synced
40  Alice in ApplyingRemoteOps, Alice.localTS is now 8
41  Bob sent CtoS_Msg with [{"userid":2,"opcode":"i"},{"TS":9}] via SendingOpsToController
42  Controller in Listening
43  Bob in AwaitingACK
44  Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"i"},{"TS":9}] while in Listening
45  Controller determined TS from Bob is new and reached PersistingNew, Controller.localTS is now 9
46  Alice in Synced with: abc
47  Controller sent StoC_ACK to Bob via SendingACKToClient
48  Bob in Synced with: abc
49  Controller sent signal StoC_Msg with data [{"userid":2,"opcode":"i"},{"TS":9}] to Alice via
      SendingToRemainingClients
50  Alice received StoC_Msg with [{"userid":2,"opcode":"i"},{"TS":9}] while in Synced
51  Alice in ApplyingRemoteOps, Alice.localTS is now 9
52  Controller in Listening
53  Alice in Synced with: abc
54  CONSISTENT AND CORRECT!
```

Listing B.3: Full console output during FSM execution of a basic buffer scenario with three identity operations (test "4").

## B.3 Buffered OT FSM with Insert and Delete

```
1   SENDING [{"userid":1,"opcode":"insertData","opcon":["a",1]},{}] TO: Alice
2   Alice in Synced with: xyz
3   Bob in Synced with: xyz
4   Controller in Listening
5   Alice typed and received Local_Change with [{"userid":1,"opcode":"insertData","opcon":["a",1]},{}]
       while in Synced
6   Alice in ApplyingLocalOps, Alice.localTS is now 16
7   SENDING [{"userid":2,"opcode":"insertData","opcon":["b",2]},{}] TO: Bob
8   Bob typed and received Local_Change with [{"userid":2,"opcode":"insertData","opcon":["b",2]},{}]
      while in Synced
9   Bob in ApplyingLocalOps, Bob.localTS is now 16
10  Alice applied op [{"userid":1,"opcode":"insertData","opcon":["a",1]},{}], Alice's document is now
      xayz
```

```
11 Alice sent CtoS_Msg with [{"userid":1,"opcode":"insertData","opcon":["a",1]},{"TS":17}] via
       SendingOpsToController
12 Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"insertData","opcon":["a
       ",1]},{"TS":17}] while in Listening
13 Controller determined TS from Alice is new and reached PersistingNew, Controller.localTS is now 17
14 Alice in AwaitingACK
15 Bob applied op [{"userid":2,"opcode":"insertData","opcon":["b",2]},{}], Bob's document is now xybz
16 Bob sent CtoS_Msg with [{"userid":2,"opcode":"insertData","opcon":["b",2]},{"TS":17}] via
       SendingOpsToController
17 Controller sent StoC_ACK to Alice via SendingACKToClient
18 Alice in Synced with: xayz
19 Bob in AwaitingACK
20 Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"insertData","opcon":["a",1]},{"TS
       ":17}] to Bob via SendingToRemainingClients
21 Bob received signal StoC_Msg while in AwaitingACK (CONFLICT RESOLUTION REQUIRED!)
22 Bob in ApplyingRemoteOpsWithoutACK, Bob.localTS is now 18
23 Controller in Listening
24 Bob XFORM: Target op is {"userid":1,"opcode":"insertData","opcon":["a",1]}, reference is {"userid
       ":2,"opcode":"insertData","opcon":["b",2]}
25 Bob XFORM: insertData vs. insertData with no parameter adjustment
26 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"insertData","opcon":["b",2]},{"
       TS":17}] while in Listening
27 Controller determined TS from Bob is old (REJECTED!)
28 Controller in Listening
29 Bob transforms remoteOps [{"userid":1,"opcode":"insertData","opcon":["a",1]},{"TS":17}] against
       localOps [{"userid":2,"opcode":"insertData","opcon":["b",2]},{}] to get [{"userid":1,"opcode
       ":"insertData","opcon":["a",1]},{"TS":17}] and applies it, Bob's document is now xaybz
30 Bob XFORM: Target op is {"userid":2,"opcode":"insertData","opcon":["b",2]}, reference is {"userid
       ":1,"opcode":"insertData","opcon":["a",1]}
31 Bob XFORM: insertData vs. insertData where increase position of target insertion
32 Bob performed the transformation with reversed parameter sequence and obtained [{"userid":2,"
       opcode":"insertData","opcon":["b",3]},{}]
33 Bob sent CtoS_Msg with [{"userid":2,"opcode":"insertData","opcon":["b",3]},{"TS":18}] via
       SendingOpsToController
34 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"insertData","opcon":["b",3]},{"
       TS":18}] while in Listening
35 Controller determined TS from Bob is new and reached PersistingNew, Controller.localTS is now 18
36 Bob in AwaitingACK
37 Controller sent StoC_ACK to Bob via SendingACKToClient
38 Bob in Synced with: xaybz
39 Controller sent signal StoC_Msg with data [{"userid":2,"opcode":"insertData","opcon":["b",3]},{"TS
       ":18}] to Alice via SendingToRemainingClients
40 Alice received StoC_Msg with [{"userid":2,"opcode":"insertData","opcon":["b",3]},{"TS":18}] while
       in Synced
41 Alice in ApplyingRemoteOps, Alice.localTS is now 18
42 Controller in Listening
43 Alice applied op [{"userid":2,"opcode":"insertData","opcon":["b",3]},{"TS":18}], Alice's document
       is now xaybz
44 Alice in Synced with: xaybz
45 CONSISTENT AND CORRECT!
```

Listing B.4: Full console output during FSM execution of a transformation scenario between character-based insertData operations (test "5").

```
1  SENDING [{"userid":2,"opcode":"deleteData","opcon":["_",33]},{}] TO: Bob
2  Alice in Synced with: tymlykekpymtkiksqsisnuamdmwmqagffnng
3  Bob in Synced with: tymlykekpymtkiksqsisnuamdmwmqagffnng
4  Controller in Listening
5  Bob typed and received Local_Change with [{"userid":2,"opcode":"deleteData","opcon":["_",33]},{}]
       while in Synced
6  Bob in ApplyingLocalOps, Bob.localTS is now 39
7  SENDING [{"userid":1,"opcode":"insertData","opcon":["g",24]},{}] TO: Alice
8  Alice typed and received Local_Change with [{"userid":1,"opcode":"insertData","opcon":["g
       ",24]},{}] while in Synced
9  Alice in ApplyingLocalOps, Alice.localTS is now 39
10 Bob applied op [{"userid":2,"opcode":"deleteData","opcon":["_",33]},{}], Bob's document is now
       tymlykekpymtkiksqsisnuamdmwmqagffng
11 Bob sent CtoS_Msg with [{"userid":2,"opcode":"deleteData","opcon":["_",33]},{"TS":40}] via
       SendingOpsToController
12 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"deleteData","opcon":["_",33]},{"
       TS":40}] while in Listening
13 Controller determined TS from Bob is new and reached PersistingNew, Controller.localTS is now 40
14 SENDING [{"userid":1,"opcode":"deleteData","opcon":["_",15]},{}] TO: Alice
15 Alice applied op [{"userid":1,"opcode":"insertData","opcon":["g",24]},{}], Alice's document is now
        tymlykekpymtkiksqsisnuamgdmwmqagffnng
16 Alice sent CtoS_Msg with [{"userid":1,"opcode":"insertData","opcon":["g",24]},{"TS":40}] via
       SendingOpsToController
17 Bob in AwaitingACK
18 Controller sent StoC_ACK to Bob via SendingACKToClient
19 Alice in AwaitingACK
20 Bob in Synced with: tymlykekpymtkiksqsisnuamdmwmqagffng
21 Controller sent signal StoC_Msg with data [{"userid":2,"opcode":"deleteData","opcon":["_",33]},{"
       TS":40}] to Alice via SendingToRemainingClients
22 SENDING [{"userid":1,"opcode":"deleteData","opcon":["_",13]},{}] TO: Alice
23 Alice typed and received signal Local_Change with buffer op [{"userid":1,"opcode":"deleteData","
       opcon":["_",15]},{}] while in AwaitingACK
24 Alice in ApplyingBufferedLocalOps with buffered op [{"userid":1,"opcode":"deleteData","opcon":["_
       ",15]},{}]
25 Controller in Listening
26 Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"insertData","opcon":["g
       ",24]},{"TS":40}] while in Listening
27 Controller determined TS from Alice is old (REJECTED!)
28 Controller in Listening
29 Alice in AwaitingWithBuffer with buffer size 1
30 Alice typed and received signal LocalChange with new buffer op [{"userid":1,"opcode":"deleteData
       ","opcon":["_",13]},{}] while in AwaitingWithBuffer
31 Alice in ApplyingBufferedLocalOps with buffered op [{"userid":1,"opcode":"deleteData","opcon":["_
       ",13]},{}]
32 Alice in AwaitingWithBuffer with buffer size 2
33 Alice received signal StoC_Msg while in AwaitingWithBuffer (CONFLICT RESOLUTION WITH BUFFER!)
34 Alice in ApplyingRemoteOpsWithBuffer, remoteOps is [{"userid":2,"opcode":"deleteData","opcon":["_
       ",33]},{"TS":40}], Alice.localTS is now 41
35 Alice XFORM: Target op is {"userid":2,"opcode":"deleteData","opcon":["_",33]}, reference is {"
       userid":1,"opcode":"insertData","opcon":["g",24]}
36 Alice XFORM: deleteData vs. insertData where increase position of incoming deletion
37 Alice XFORM: Target op is {"userid":2,"opcode":"deleteData","opcon":["_",34]}, reference is {"
       userid":1,"opcode":"deleteData","opcon":["_",15]}
38 Alice XFORM: deleteData vs. deleteData where decrease position of incoming deletion
```

```
39 Alice XFORM: Target op is {"userid":2,"opcode":"deleteData","opcon":["_",33]}, reference is {"
       userid":1,"opcode":"deleteData","opcon":["_",13]}
40 Alice XFORM: deleteData vs. deleteData where decrease position of incoming deletion
41     That is, Alice transforms remoteOps [{"userid":2,"opcode":"deleteData","opcon":["_",33]},{"TS
       ":40}] against localOps [{"userid":1,"opcode":"insertData","opcon":["g",24]},{}] and buffer
       [{"userid":1,"opcode":"deleteData","opcon":["_",15]},{}] and buffer [{"userid":1,"opcode":"
       deleteData","opcon":["_",13]},{}] to get [{"userid":2,"opcode":"deleteData","opcon":["_
       ",32]},{"TS":40}] and applied it, Alice's document is now tymlykekpymtkkqsisnuamgdmwmqagffng
42 Alice XFORM: Target op is {"userid":1,"opcode":"insertData","opcon":["g",24]}, reference is {"
       userid":2,"opcode":"deleteData","opcon":["_",33]}
43 Alice XFORM: insertData vs. deleteData with no parameter adjustment (incoming Ins is before local
       Del)
44 Alice performed the transformation with reversed parameter sequence and obtained [{"userid":1,"
       opcode":"insertData","opcon":["g",24]},{}] as the updated ACK that awaiting
45 Alice XFORM: Target op is {"userid":1,"opcode":"deleteData","opcon":["_",15]}, reference is {"
       userid":2,"opcode":"deleteData","opcon":["_",34]}
46 Alice XFORM: deleteData vs. deleteData with no parameter adjustment
47     That is, Alice buffer[1] is now [{"userid":1,"opcode":"deleteData","opcon":["_",15]},{}]
48 Alice XFORM: Target op is {"userid":1,"opcode":"deleteData","opcon":["_",13]}, reference is {"
       userid":2,"opcode":"deleteData","opcon":["_",33]}
49 Alice XFORM: deleteData vs. deleteData with no parameter adjustment
50     That is, Alice buffer[2] is now [{"userid":1,"opcode":"deleteData","opcon":["_",13]},{}]
51 Alice sent CtoS_Msg with [{"userid":1,"opcode":"insertData","opcon":["g",24]},{"TS":41}] via
       SendingOpsToController
52 Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"insertData","opcon":["g
       ",24]},{"TS":41}] while in Listening
53 Controller determined TS from Alice is new and reached PersistingNew, Controller.localTS is now 41
54 Alice in AwaitingWithBuffer with buffer size 2
55 Controller sent StoC_ACK to Alice via SendingACKToClient
56 Alice received signal StoC_ACK while in AwaitingWithBuffer
57 Alice in CreatingLocalOpFromBuffer, Alice.localTS is now 42
58 Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"insertData","opcon":["g",24]},{"
       TS":41}] to Bob via SendingToRemainingClients
59 Alice sent CtoS_Msg with [{"userid":1,"opcode":"deleteData","opcon":["_",15]},{"TS":42}] via
       SendingOpsToController
60 Bob received StoC_Msg with [{"userid":1,"opcode":"insertData","opcon":["g",24]},{"TS":41}] while
       in Synced
61 Bob in ApplyingRemoteOps, Bob.localTS is now 41
62 Controller in Listening
63 Alice in AwaitingWithBuffer with buffer size 1
64 Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"deleteData","opcon":["_
       ",15]},{"TS":42}] while in Listening
65 Controller determined TS from Alice is new and reached PersistingNew, Controller.localTS is now 42
66 Bob applied op [{"userid":1,"opcode":"insertData","opcon":["g",24]},{"TS":41}], Bob's document is
       now tymlykekpymtkiksqsisnuamgdmwmqagffng
67 Bob in Synced with: tymlykekpymtkiksqsisnuamgdmwmqagffng
68 Controller sent StoC_ACK to Alice via SendingACKToClient
69 Alice received signal StoC_ACK while in AwaitingWithBuffer
70 Alice in CreatingLocalOpFromBuffer, Alice.localTS is now 43
71 Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"deleteData","opcon":["_",15]},{"
       TS":42}] to Bob via SendingToRemainingClients
72 Alice sent CtoS_Msg with [{"userid":1,"opcode":"deleteData","opcon":["_",13]},{"TS":43}] via
       SendingOpsToController
73 Bob received StoC_Msg with [{"userid":1,"opcode":"deleteData","opcon":["_",15]},{"TS":42}] while
       in Synced
```

```
74 Bob in ApplyingRemoteOps, Bob.localTS is now 42
75 Controller in Listening
76 Alice in AwaitingACK
77 Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"deleteData","opcon":["_
       ",13]},{"TS":43}] while in Listening
78 Controller determined TS from Alice is new and reached PersistingNew, Controller.localTS is now 43
79 Bob applied op [{"userid":1,"opcode":"deleteData","opcon":["_",15]},{"TS":42}], Bob's document is
       now tymlykekpymtkikqsisnuamgdmwmqagffng
80 Bob in Synced with: tymlykekpymtkikqsisnuamgdmwmqagffng
81 Controller sent StoC_ACK to Alice via SendingACKToClient
82 Alice in Synced with: tymlykekpymtkkqsisnuamgdmwmqagffng
83 Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"deleteData","opcon":["_",13]},{"
       TS":43}] to Bob via SendingToRemainingClients
84 Bob received StoC_Msg with [{"userid":1,"opcode":"deleteData","opcon":["_",13]},{"TS":43}] while
       in Synced
85 Bob in ApplyingRemoteOps, Bob.localTS is now 43
86 Controller in Listening
87 Bob applied op [{"userid":1,"opcode":"deleteData","opcon":["_",13]},{"TS":43}], Bob's document is
       now tymlykekpymtkkqsisnuamgdmwmqagffng
88 Bob in Synced with: tymlykekpymtkkqsisnuamgdmwmqagffng
89 CONSISTENT AND CORRECT!
```

Listing B.5: Full console output during FSM execution of a randomized buffer scenario with character-based insertData and deleteData operations (test "6").

## B.4 HTML DOM-Based OT FSM with Three Users

```
1 SENDING [{"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]},{"userid":1,"opcode":"insertNode
       ","opcon":4,"loc":[1,2]},{"userid":1,"opcode":"giveData","opcon":[[true,1,5],[true,2,4]],"loc
       ":2},[[true,1,"strong",[]],[true,3,"cd",[]],[false,3,"b",[3,0]]],{}] TO: Alice
2 Alice in Synced with realMap: [body, p, "abcd"]
3 Bob in Synced with realMap: [body, p, "abcd"]
4 Carol in Synced with realMap: [body, p, "abcd"]
5 Controller in Listening
6 Alice typed and received Local_Change with [{"userid":1,"opcode":"insertNode","opcon":3,"loc
       ":[1,1]},{"userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]},{"userid":1,"opcode":"
       giveData","opcon":[[true,1,5],[true,2,4]],"loc":2},[[true,1,"strong",[]],[true,3,"cd",[]],[
       false,3,"b",[3,0]]],{}] while in Synced
7 Alice in ApplyingLocalOps, Alice.localTS is now 134
8 Alice appending node "b" to transMap index 3, namely "strong"
9 Alice applied op {"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]}, Alice.realDOM is now <
       body><p>abcd<strong>b</strong></p></body>
10 Alice's prevMap: [body, p, "abcd"]
11 Alice's localTM: [body, p, "abcd", strong, "cd", "b"]
12 Alice's realMap: [body, p, "abcd", strong, "b"]
13 Alice applied op {"userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]}, Alice.realDOM is now <
       body><p>abcd<strong>b</strong>cd</p></body>
14 Alice's prevMap: [body, p, "abcd"]
15 Alice's localTM: [body, p, "abcd", strong, "cd", "b"]
```

```
16 Alice's realMap: [body, p, "abcd", strong, "cd", "b"]
17 Alice applied op {"userid":1,"opcode":"giveData","opcon":[[true,1,5],[true,2,4]],"loc":2}, Alice.
       realDOM is now <body><p>a<strong>b</strong>cd</p></body>
18 Alice's prevMap: [body, p, "a"]
19 Alice's localTM: [body, p, "a", strong, "cd", "b"]
20 Alice's realMap: [body, p, "a", strong, "cd", "b"]
21 Alice sent CtoS_Msg with [{"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]},{"userid":1,"
       opcode":"insertNode","opcon":4,"loc":[1,2]},{"userid":1,"opcode":"giveData","opcon":[[true
       ,1,5],[true,2,4]],"loc":2},[[true,1,"strong",[]],[true,3,"cd",[]],[false,3,"b",[3,0]]],{"TS
       ":135}] via SendingOpsToController
22 Controller consumed CtoS_Msg from Alice with [{"userid":1,"opcode":"insertNode","opcon":3,"loc
       ":[1,1]},{"userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]},{"userid":1,"opcode":"
       giveData","opcon":[[true,1,5],[true,2,4]],"loc":2},[[true,1,"strong",[]],[true,3,"cd",[]],[
       false,3,"b",[3,0]]],{"TS":135}] while in Listening
23 Controller determined TS from Alice is new and reached PersistingNew, Controller.localTS is now
       135
24 SENDING [{"userid":2,"opcode":"deleteData","opcon":["d",3],"loc":2},[]] TO: Bob
25 Alice in AwaitingACK
26 Bob typed and received Local_Change with [{"userid":2,"opcode":"deleteData","opcon":["d",3],"loc
       ":2},[]] while in Synced
27 Bob in ApplyingLocalOps, Bob.localTS is now 134
28 Controller sent StoC_ACK to Alice via SendingACKToClient
29 Alice in Synced with realMap: [body, p, "a", strong, "cd", "b"]
30 Bob applied op {"userid":2,"opcode":"deleteData","opcon":["d",3],"loc":2}, Bob.realDOM is now <
       body><p>abc</p></body>
31 Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"insertNode","opcon":3,"loc
       ":[1,1]},{"userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]},{"userid":1,"opcode":"
       giveData","opcon":[[true,1,5],[true,2,4]],"loc":2},[[true,1,"strong",[]],[true,3,"cd",[]],[
       false,3,"b",[3,0]]],{"TS":135}] to Bob via SendingToRemainingClients
32 Controller sent signal StoC_Msg with data [{"userid":1,"opcode":"insertNode","opcon":3,"loc
       ":[1,1]},{"userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]},{"userid":1,"opcode":"
       giveData","opcon":[[true,1,5],[true,2,4]],"loc":2},[[true,1,"strong",[]],[true,3,"cd",[]],[
       false,3,"b",[3,0]]],{"TS":135}] to Carol via SendingToRemainingClients
33 Bob sent CtoS_Msg with [{"userid":2,"opcode":"deleteData","opcon":["d",3],"loc":2},[],{"TS":135}]
       via SendingOpsToController
34 Carol received StoC_Msg with [{"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]},{"userid
       ":1,"opcode":"insertNode","opcon":4,"loc":[1,2]},{"userid":1,"opcode":"giveData","opcon":[[
       true,1,5],[true,2,4]],"loc":2},[[true,1,"strong",[]],[true,3,"cd",[]],[false,3,"b",[3,0]]],{"
       TS":135}] while in Synced
35 Carol in ApplyingRemoteOps, Carol.localTS is now 135
36 Carol appending node "b" to transMap index 3, namely "strong"
37 Bob in AwaitingACK
38 Carol applied op {"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]}, Carol.realDOM is now <
       body><p>abcd<strong>b</strong></p></body>
39 Carol's prevMap: [body, p, "abcd"]
40 Carol's localTM: [body, p, "abcd", strong, "cd", "b"]
41 Carol's realMap: [body, p, "abcd", strong, "b"]
42 Controller in Listening
43 Bob received signal StoC_Msg while in AwaitingACK - CONFLICT RESOLUTION REQUIRED!
44 Bob in ApplyingRemoteOpsWithoutACK, Bob.localTS is now 136
45 Bob appending node "b" to transMap index 3, namely "strong"
46 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"deleteData","opcon":["d",3],"loc
       ":2},[],{"TS":135}] while in Listening
47 Controller determined TS from Bob is old - REJECTED!
48 Controller in Listening
```

```
49 Carol applied op {"userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]}, Carol.realDOM is now <
       body><p>abcd<strong>b</strong>cd</p></body>
50 Carol's prevMap: [body, p, "abcd"]
51 Carol's localTM: [body, p, "abcd", strong, "cd", "b"]
52 Carol's realMap: [body, p, "abcd", strong, "cd", "b"]
53 Bob XFORM: Target op is {"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]}, reference is {"
       userid":2,"opcode":"deleteData","opcon":["d",3],"loc":2}
54 Bob XFORM: Resulting op is: {"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]}
55 Carol applied op {"userid":1,"opcode":"giveData","opcon":[[true,1,5],[true,2,4]],"loc":2}, Carol.
       realDOM is now <body><p>a<strong>b</strong>cd</p></body>
56 Carol's prevMap: [body, p, "a"]
57 Carol's localTM: [body, p, "a", strong, "cd", "b"]
58 Carol's realMap: [body, p, "a", strong, "cd", "b"]
59 Bob XFORM: Target op is {"userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]}, reference is {"
       userid":2,"opcode":"deleteData","opcon":["d",3],"loc":2}
60 Bob XFORM: Resulting op is: {"userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]}
61 Carol in Synced with realMap: [body, p, "a", strong, "cd", "b"]
62 Bob XFORM: Target op is {"userid":1,"opcode":"giveData","opcon":[[true,1,5],[true,2,4]],"loc":2},
       reference is {"userid":2,"opcode":"deleteData","opcon":["d",3],"loc":2}
63 Bob XFORM: Resulting op is: {"userid":1,"opcode":"giveData","opcon":[[true,1,5],[true,2,4]],"loc
       ":2}
64 Bob appending node "b" to transMap index 3, namely "strong"
65 Bob applied op {"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]}, Bob.realDOM is now <body
       ><p>abc<strong>b</strong></p></body>
66 Bob's prevMap: [body, p, "abc"]
67 Bob's remoteTM: [body, p, "abc", strong, "c", "b"]
68 Bob's realMap: [body, p, "abc", strong, "b"]
69 Bob applied op {"userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]}, Bob.realDOM is now <body
       ><p>abc<strong>b</strong>c</p></body>
70 Bob's prevMap: [body, p, "abc"]
71 Bob's remoteTM: [body, p, "abc", strong, "c", "b"]
72 Bob's realMap: [body, p, "abc", strong, "c", "b"]
73 Bob applied op {"userid":1,"opcode":"giveData","opcon":[[true,1,5],[true,2,4]],"loc":2}, Bob.
       realDOM is now <body><p>a<strong>b</strong>c</p></body>
74 Bob's prevMap: [body, p, "a"]
75 Bob's remoteTM: [body, p, "a", strong, "c", "b"]
76 Bob's realMap: [body, p, "a", strong, "c", "b"]
77     That is, Bob transforms remoteOps {"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]}{"
       userid":1,"opcode":"insertNode","opcon":4,"loc":[1,2]}{"userid":1,"opcode":"giveData","opcon
       ":[[true,1,5],[true,2,4]],"loc":2} against localOps {"userid":2,"opcode":"deleteData","opcon
       ":["d",3],"loc":2} to get {"userid":1,"opcode":"insertNode","opcon":3,"loc":[1,1]}{"userid
       ":1,"opcode":"insertNode","opcon":4,"loc":[1,2]}{"userid":1,"opcode":"giveData","opcon":[[
       true,1,5],[true,2,4]],"loc":2} and applies it, Bob.realMap is now [body, p, "a", strong, "c",
        "b"]
78 Bob performed the diff() and obtained [{"userid":2,"opcode":"deleteData","opcon":["d",1],"loc
       ":4},[]]
79 Bob sent CtoS_Msg with [{"userid":2,"opcode":"deleteData","opcon":["d",1],"loc":4},[],{"TS":136}]
       via SendingOpsToController
80 Controller consumed CtoS_Msg from Bob with [{"userid":2,"opcode":"deleteData","opcon":["d",1],"loc
       ":4},[],{"TS":136}] while in Listening
81 Controller determined TS from Bob is new and reached PersistingNew, Controller.localTS is now 136
82 Bob in AwaitingACK
83 Controller sent StoC_ACK to Bob via SendingACKToClient
84 Bob in Synced with realMap: [body, p, "a", strong, "c", "b"]
```

```
85 Controller sent signal StoC_Msg with data [{"userid":2,"opcode":"deleteData","opcon":["d",1],"loc
       ":4},[],{"TS":136}] to Alice via SendingToRemainingClients
86 Alice received StoC_Msg with [{"userid":2,"opcode":"deleteData","opcon":["d",1],"loc":4},[],{"TS
       ":136}] while in Synced
87 Alice in ApplyingRemoteOps, Alice.localTS is now 136
88 Controller sent signal StoC_Msg with data [{"userid":2,"opcode":"deleteData","opcon":["d",1],"loc
       ":4},[],{"TS":136}] to Carol via SendingToRemainingClients
89 Alice applied op {"userid":2,"opcode":"deleteData","opcon":["d",1],"loc":4}, Alice.realDOM is now
       <body><p>a<strong>b</strong>c</p></body>
90 Carol received StoC_Msg with [{"userid":2,"opcode":"deleteData","opcon":["d",1],"loc":4},[],{"TS
       ":136}] while in Synced
91 Carol in ApplyingRemoteOps, Carol.localTS is now 136
92 Carol applied op {"userid":2,"opcode":"deleteData","opcon":["d",1],"loc":4}, Carol.realDOM is now
       <body><p>a<strong>b</strong>c</p></body>
93 Controller in Listening
94 Alice in Synced with realMap: [body, p, "a", strong, "c", "b"]
95 Carol in Synced with realMap: [body, p, "a", strong, "c", "b"]
96 CONSISTENT AND CORRECT!
```

Listing B.6: Full console output during FSM execution of a DOM node splitting scenario (test "7").

# Bibliography


[1] M. Brenan. (2020, April) U.S. Workers Discovering Affinity for Remote Work. Gallup Inc. [Accessed: March 2021]. [Online]. Available: https://news.gallup.com/poll/306695/workers-discovering-affinity-remote-work.aspx

[2] C. A. Ellis, S. J. Gibbs, and G. Rein, "Groupware: Some Issues and Experiences," *Communications of the ACM*, vol. 34, no. 1, pp. 39–58, 1991.

[3] C. A. Ellis and S. J. Gibbs, "Concurrency Control in Groupware Systems," in *ACM SIGMOD Record*, vol. 18, no. 2. ACM, 1989, pp. 399–407.

[4] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 5, no. 1, pp. 63–108, 1998.

[5] C. Sun and C. Ellis, "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements," in *Proc. of the 1998 ACM Conference on Computer Supported Cooperative Work*. ACM, 1998, pp. 59–68.

[6] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping, "High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System," in *Proceedings of the 8th annual ACM symposium on User interface and software technology*. ACM, 1995, pp. 111–120.

[7] N. Vidot, J. Ferrié, and M. Suleiman, "Copies Convergence in a Distributed Real-Time Collaborative Environment," in *Proc. of the 2000 ACM Conf. on Computer Supported Cooperative Work*. ACM, 2000, pp. 171–180.

[8] H. Shen and C. Sun, "Flexible Notification for Collaborative Systems," in *Proc. of the 2002 ACM Conf. on Computer Supported Cooperative Work*. ACM, 2002, pp. 77–86.

[9] R. Li, D. Li, and C. Sun, "A Time Interval Based Consistency Control Algorithm for Interactive Groupware Applications," in *ICPADS 2004: Proc. of the 10th Int. Conf. on Parallel and Distributed Systems*. IEEE Computer Society, 2004, pp. 429–436.

[10] Roberts, Mike. (2016, August) Serverless Architectures. [Accessed: March 2021]. [Online]. Available: https://martinfowler.com/articles/serverless.html

[11] Firebase. Google Inc. [Accessed: March 2021]. [Online]. Available: https://firebase.google.com/

[12] I. Koren, A. Guth, and R. Klamma, "Shared Editing on the Web: A Classification of Developer Support Libraries," in *Collaboratecom 2013: 9th Int. Conf. on Collaborative Computing: Networking, Applications and Worksharing*. IEEE Computer Society, 2013, pp. 468–477.

[13] A. B. Johnston and D. C. Burnett, *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web, Third Edition*. Digital Codex LLC, 2014.

[14] Google Docs - Online Documents, Spreadsheets, Presentations. Google Inc. [Accessed: March 2021]. [Online]. Available: http://docs.google.com

[15] Google Workspace (Formerly G Suite): Business Collaboration Tools. Google Inc. [Accessed: March 2021]. [Online]. Available: https://workspace.google.com/

[16] Solomon, Scott. (2016, April) Google and Microsoft's Battle for the Enterprise: How Users Are Interacting With SaaS Applications - BetterCloud Monitor. BetterCloud Monitor. [Accessed: March 2021]. [Online]. Available: https://www.bettercloud.com/monitor/google-apps-and-office-365-differences/

[17] Google Trends - Search Results for "google docs". Google Inc. [Accessed: March 2021]. [Online]. Available: https://trends.google.ca/trends/explore?date=2006-01-01%202020-10-01&q=google%20docs

[18] MediaWiki VisualEditor. Wikimedia Foundation. [Accessed: March 2021]. [Online]. Available: https://www.mediawiki.org/wiki/VisualEditor

[19] XWiki SAS Home. XWiki SAS. [Accessed: March 2021]. [Online]. Available: http://www.xwiki.com

[20] Collaboration Software & Social Collaboration Tools. Jive Software Inc. [Accessed: March 2021]. [Online]. Available: https://www.jivesoftware.com/

[21] WordPress.com: Create a free website or blog. WordPress Foundation. [Accessed: March 2021]. [Online]. Available: https://www.wordpress.com/

[22] A. Spadafora. (2021, February) WordPress Now Powers 40% of the World's Websites. TechRadar. [Accessed: March 2021]. [Online]. Available: https://www.techradar.com/news/wordpress-now-powers-40-of-the-worlds-websites

[23] Schäferhoff, Nick. (2016, October) 13 Surprising WordPress Usage Statistics (Updated 2016!). Torque - WordPress News, WP Community Experts. [Accessed: March 2021]. [Online]. Available: https://torquemag.io/2016/10/13-surprising-wordpress-statistics-updated-2016/

[24] Joomla! The CMS Trusted By Millions for their Websites. The Joomla Project Teams. [Accessed: March 2021]. [Online]. Available: https://www.joomla.org/

[25] TinyMCE - The Most Advanced WYSIWYG HTML Editor. Tiny Technologies Inc. [Accessed: September 2018]. [Online]. Available: https://www.tiny.cloud/

[26] Buytaert, Dries. Drupal - Open Source CMS. [Accessed: March 2021]. [Online]. Available: https://www.drupal.org/

[27] CKEditor.com - The best web text editor for everyone. CKSource sp. z.o.o. sp.k. [Accessed: March 2021]. [Online]. Available: http://ckeditor.com/

[28] Google Drive Blog: September 2010. Google Inc. [Accessed: March 2021]. [Online]. Available: http://googledrive.blogspot.ca/2010_09_01_archive.html

[29] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai, "Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 13, no. 4, pp. 531–582, 2006.

[30] M. Heinrich, F. Lehmann, T. Springer, and M. Gaedke, "Exploiting Single-User Web Applications for Shared Editing: A Generic Transformation Approach," in *Proc. of the 21st international conference on World Wide Web*. ACM, 2012, pp. 1057–1066.

[31] M. Heinrich, F. Lehmann, F. J. Grüneberger, T. Springer, and M. Gaedke, "Analyzing the Suitability of Web Applications for a Single-User to Multi-User Transformation," in *Proc. of the 22nd Int. Conf. on World Wide Web*. ACM, 2013, pp. 249–252.

[32] M. Heinrich, "Enriching Web Applications Efficiently with Real-Time Collaboration Capabilities," PhD Thesis, Universitätsverlag Chemnitz, 2014.

[33] A. van Kesteren, A. Gregor, Ms2ger, A. Russell, and R. Berjon, "W3C DOM4," W3C, Tech. Rep., November 2015, [Accessed: March 2021]. [Online]. Available: https://www.w3.org/TR/2015/REC-dom-20151119/

[34] (2017) World Wide Web Consortium (W3C). W3C. [Accessed: March 2021]. [Online]. Available: http://www.w3.org/

[35] B. Solomon, D. Ionescu, C. Gadea, and M. Litoiu, *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. IGI Global, November 2012, ch. Geographically Distributed Cloud-Based Collaborative Application.

[36] B. Ionescu, D. Ionescu, C. Gadea, B. Solomon, and M. Trifan, "An Architecture and Methods for Big Data Analysis," in *SOFA 2014: Proc. of the 6th Int. Workshop Soft Computing Applications*, vol. 1. Springer International Publishing, 2016, pp. 491–514.

[37] B. Ionescu, C. Gadea, B. Solomon, D. Ionescu, V. Stoicu-Tivadar, and M. Trifan, "A Cloud Based Real-Time Collaborative Platform for eHealth," *Studies in Health Technology and Informatics*, vol. 210, pp. 919–923, 2015.

[38] B. Ionescu, C. Gadea, B. Solomon, M. Trifan, D. Ionescu, and V. Stoicu-Tivadar, "A Chat-Centric Collaborative Environment for Web-Based Real-Time Collaboration," in *SACI 2015: IEEE 10th Jubilee Int. Symp. on Applied Computational Intelligence and Informatics*. IEEE Computer Society, 2015, pp. 105–110.

[39] C. Gadea, M. Trifan, D. Ionescu, and B. Ionescu, "A Reference Architecture for Real-Time Microservice API Consumption," in *Proc. of the 3rd Workshop on CrossCloud Infrastructures & Platforms*. ACM, 2016, p. 2.

[40] C. Gadea, M. Trifan, D. Ionescu, M. Cordea, and B. Ionescu, "A Microservices Architecture for Collaborative Document Editing Enhanced with Face Recognition," in *SACI 2016: Proc. of 11th Int. Symp. on Applied Computational Intelligence and Informatics*. IEEE Computer Society, 2016, pp. 441–446.

[41] C. Gadea, D. Hong, D. Ionescu, and B. Ionescu, "An Architecture for Web-based Collaborative 3D Virtual Spaces using DOM Synchronization," in *CIVEMSA 2016: Proc. of Int. Conf. on Computational Intelligence and Virtual Environments for Measurement Systems and Applications*. IEEE Computer Society, 2016, pp. 1–6.

[42] C. Gadea, B. Ionescu, and D. Ionescu, "Modeling and Simulation of an Operational Transformation Algorithm using Finite State Machines," in *SACI 2018: Proc. of 12th Int. Symp. on Applied Computational Intelligence and Informatics*. IEEE Computer Society, 2018, pp. 119–124.

[43] ——, "A Hybrid FSM Rule-Based Approach for the Real-Time Control of Web-Based Collaborative Platforms," in *INES 2018: Proc. of 22nd Int. Conf. on Intelligent Engineering Systems*. IEEE Computer Society, 2018, pp. 27–32.

[44] ——, "New Algorithms and Methods for Collaborative Co-Editing Using HTML DOM Synchronization," in *CIC 2018: Proc. of 4th IEEE Int. Conf. on Collaboration and Internet Computing*. IEEE Computer Society, 2018, pp. 217–226.

[45] ——, "A Control Loop-based Algorithm for Operational Transformation," in *SACI 2020: Proc. of 14th Int. Symp. on Applied Computational Intelligence and Informatics*. IEEE Computer Society, 2020, pp. 247–254.

[46] D. Sun and C. Sun, "Context-Based Operational Transformation in Distributed Collaborative Editing Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 10, pp. 1454–1470, 2009.

[47] D. Li and R. Li, "An Admissibility-Based Operational Transformation Framework for Collaborative Editing Systems," *Computer Supported Cooperative Work (CSCW)*, vol. 19, no. 1, pp. 1–43, 2010.

[48] A. Imine, P. Molli, G. Oster, and M. Rusinowitch, "Proving Correctness of Transformation Functions in Real-Time Groupware," in *ECSCW 2003: Proc. of the 8th European Conf. on Computer Supported Cooperative Work*. Springer Netherlands, September 2003, pp. 277–293.

[49] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design Science in Information Systems Research," *MIS quarterly*, pp. 75–105, 2004.

[50] J. Grudin, "Computer-Supported Cooperative Work: History and Focus," *Computer*, vol. 27, no. 5, pp. 19–26, 1994.

[51] N. Vidot, "Convergence des copies dans les environnements collaboratifs répartis," PhD Thesis, Montpellier 2 University, 2002.

[52] R. Johansen, *GroupWare: Computer Support for Business Teams*. New York, NY, USA: The Free Press, 1988.

[53] Slack: Where work happens. Slack Technologies Inc. [Accessed: March 2021]. [Online]. Available: https://www.slack.com

[54] I. Greif, R. Seliger, and W. E. Weihl, "Atomic Data Abstractions in a Distributed Collaborative Editing System," in *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '86. New York, NY, USA: ACM, 1986, pp. 160–172.

[55] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[56] C. Sun, Y. Yang, Y. Zhang, and D. Chen, "A Consistency Model and Supporting Schemes for Real-Time Cooperative Editing Systems," *Australian Computer Science Communications*, vol. 18, pp. 582–591, 1996.

[57] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design (5th Edition)*. Pearson, 2011.

[58] F. Mattern, "Virtual Time and Global States of Distributed Systems," in *Parallel and Distributed Algorithms*. North-Holland, 1989, pp. 215–226.

[59] C. Fidge, "Logical Time in Distributed Computing Systems," *Computer*, vol. 24, no. 8, pp. 28–33, 1991.

[60] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser, "An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors," in *Proc. of the 1996 ACM Conf. on Computer Supported Cooperative Work*. ACM, 1996, pp. 288–297.

[61] Sun, Chengzheng. OTFAQ: Operational Transformation Frequently Asked Questions and Answers. [Accessed: March 2021]. [Online]. Available: http://web.archive.org/web/20200623064915/https://www3.ntu.edu.sg/home/czsun/projects/otfaq/

[62] A. Imine, M. Rusinowitch, G. Oster, and P. Molli, "Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence," *Theoretical Computer Science*, vol. 351, no. 2, pp. 167–183, 2006.

[63] C. Sun, Y. Xu, and A. Ng, "Exhaustive Search and Resolution of Puzzles in OT Systems Supporting String-Wise Operations," in *Proc. of the 2017 ACM Conf. on Computer Supported Cooperative Work and Social Computing*. ACM, 2017, pp. 2504–2517.

[64] S. Kumawat and A. Khunteta, "A Survey on Operational Transformation Algorithms: Challenges, Issues and Achievements," *Int. Journal of Computer Applications*, vol. 3, no. 12, pp. 30–38, July 2010.

[65] L. André, "Préservation des intentions et maintien de la cohérence des données répliquées en temps réel," PhD Thesis, Université de Lorraine (Nancy), 2016.

[66] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[67] A. H. Davis, C. Sun, and J. Lu, "Generalizing Operational Transformation to the Standard General Markup Language," in *Proc. of the 2002 ACM Conf. on Computer Supported Cooperative Work*. ACM, 2002, pp. 58–67.

[68] A. A. Zafer, "Netedit: A Collaborative Editor," MASc Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, April 2001.

[69] W. R. Stevens, *TCP/IP Illustrated, Vol. 1: The Protocols*. Addison-Wesley Professional, 1994.

[70] D. P. Reed and R. K. Kanodia, "Synchronization with Eventcounts and Sequencers," *Communications of the ACM*, vol. 22, no. 2, pp. 115–123, 1979.

[71] G. Oster, P. Urso, P. Molli, and A. Imine, "Data Consistency for P2P Collaborative Editing," in *Proc. of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*. ACM, 2006, pp. 259–268.

[72] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2011, pp. 386–400.

[73] M. Shapiro and N. Preguiça, "Designing a Commutative Replicated Data Type," *arXiv preprint arXiv:0710.1784*, 2007.

[74] Developing with Riak KV Data Types. [Accessed: March 2021]. [Online]. Available: https://docs.riak.com/riak/kv/2.2.3/developing/data-types/

[75] C. Sun, D. Sun, A. Ng, W. Cai, and B. Cho, "Real Differences between OT and CRDT under a General Transformation Framework for Consistency Maintenance in Co-Editors," *Proc. of the 2020 ACM on Human-Computer Interaction*, vol. 4, no. GROUP, pp. 1–26, 2020.

[76] C. Sun, D. Sun, W. Cai *et al.*, "Real Differences Between OT and CRDT for Co-Editors," *arXiv preprint arXiv:1810.02137*, 2018.

[77] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A Commutative Replicated Data Type for Cooperative Editing," in *ICDCS'09: 29th IEEE Int. Conf. on Distributed Computing Systems*. IEEE Computer Society, 2009, pp. 395–403.

[78] R. Dagher, C. Gadea, B. Ionescu, D. Ionescu, and R. Tropper, "A SIP Based P2P Architecture for Social Networking Multimedia," in *DS-RT 2008: 12th IEEE/ACM Int. Symp. on Distributed Simulation and Real-Time Applications*. IEEE Computer Society, October 2008, pp. 187–193.

[79] R. Tropper, R. Dagher, C. Gadea, B. Ionescu, and D. Ionescu, "UC-IC: A Cloud Based and Real-Time Collaboration Platform Using Many-to-Many-on-Many Relationships," in *I2TS 2009: Proc. of 8th IEEE Int. Information and Telecommunication and Technologies Conference*. IEEE Computer Society, May 2009, pp. 9–17.

[80] R. Tropper, "New Architecture and Programming Paradigms for Cloud Collaborative RIA," MASc Thesis, School of Information Technology and Engineering, University of Ottawa, December 2009.

[81] C. Gadea, "Collaborative Web-Based Mapping of Real-Time Sensor Data," MASc Thesis, School of Information Technology and Engineering, University of Ottawa, February 2011.

[82] C. Gadea, B. Solomon, B. Ionescu, and D. Ionescu, "A Real-Time Browser-Based Collaboration System for Synchronized Online Multimedia Sharing," in *I2TS 2010: Proc. of 9th IEEE Int. Information and Telecommunication and Technologies Conference*. IEEE Computer Society, May 2010, pp. 39–46.

[83] B. Solomon, D. Ionescu, C. Gadea, S. Veres, and M. Litoiu, "Leaky Bucket Model for Autonomic Control of Distributed, Collaborative Systems," in *SACI 2013: Proc. of IEEE 8th Int. Symp. on Applied Computational Intelligence and Informatics*. IEEE Computer Society, 2013, pp. 483–488.

[84] B. Solomon, D. Ionescu, and C. Gadea, "Self-Organizing System for the Autonomic Management of Collaborative Cloud Applications," in *SOFA 2014: Proc. of the 6th Int. Workshop Soft Computing Applications*, vol. 1. Springer International Publishing, 2016, pp. 469–490.

[85] Siegler, MG. (2009, May) Google Wave Drips With Ambition. A New Communication Platform For A New Web. TechCrunch. [Accessed: March 2021]. [Online]. Available: https://www.techcrunch.com/2009/05/28/google-wave-drips-with-ambition-can-it-fulfill-googles-grand-web-vision/

[86] Wave Incubation Status - Apache Incubator. Apache Software Foundation. [Accessed: March 2021]. [Online]. Available: http://incubator.apache.org/wave/

[87] Wave - Github. [Accessed: March 2021]. [Online]. Available: https://github.com/wave-protocol

[88] A. Ferrate, *Google Wave: Up and Running*. O'Reilly Media, Inc., 2010.

[89] Spiewak, Daniel. (2011, December) Common Collaborative Coding Protocol. [Accessed: March 2021]. [Online]. Available: https://github.com/djspiewak/cccp

[90] Gregorio, Joe and North, Alex. (2009, October) Google Wave Conversation Model. Google Inc. [Accessed: March 2021]. [Online]. Available: https://svn.apache.org/repos/asf/incubator/wave/whitepapers/conversation/convspec.html

[91] Wang, David and Mah, Alex and Lassen, Soren. (2010, July) Google Wave Operational Transformation. Google Inc. [Accessed: March 2021]. [Online]. Available: https://svn.apache.org/repos/asf/incubator/wave/whitepapers/operational-transform/operational-transform.html

[92] DevChat.tv. (2018) JavaScript Jabber Podcast Episode 142: Share.js with Joseph Gentle. [Accessed: March 2021]. [Online]. Available: https://devchat.tv/js-jabber/142-jsj-share-js-with-joseph-gentle

[93] (2010) Google Wave Federation Protocol FAQ. Google Inc. [Accessed: March 2021]. [Online]. Available: http://web.archive.org/web/20180614133041/http://www.waveprotocol.org/faq

[94] F. Yusuf. (2019, December) How to Clean Content from Google Docs to Avoori/WordPress. Avoori. [Accessed: March 2021]. [Online]. Available: https://avoori.com/avoori/how-to-clean-content-from-google-docs-to-avoori-wordpress/

[95] H. Meran. (2013, October) Wikidocs - Real Time Collaborative Editing for HTML. Wikidocs Inc. [Accessed: March 2021]. [Online]. Available: https://www.slideshare.net/draftkraft/wikidocs-real-time-collaborative

[96] Gentle, Joseph and Smith, Nate. sharedb - A Database Frontend for Concurrent Editing Systems. [Accessed: March 2021]. [Online]. Available: https://github.com/share/sharedb

[97] C. N. Klokmose, J. R. Eagan, S. Baader, W. Mackay, and M. Beaudouin-Lafon, "Webstrates: Shareable Dynamic Media," in *Proc. of the 28th Annual ACM Symp. on User Interface Software & Technology*. ACM, 2015, pp. 280–290.

[98] Gentle, Joseph. (2017) JSON0 OT Type - ottypes/json0 - GitHub. [Accessed: March 2021]. [Online]. Available: https://github.com/ottypes/json0

[99] Klokmose, Clemens Nylandsted. Webstrates. [Accessed: March 2021]. [Online]. Available: https://github.com/Webstrates/Webstrates

[100] Rose, Jeremy. (2013) Rich text support - Issue #1 - josephg/ShareJS - GitHub. [Accessed: March 2021]. [Online]. Available: https://github.com/josephg/ShareJS/issues/1#issuecomment-12861678

[101] A. Le Hors, P. Le Hgaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne, "Document Object Model (DOM) Level 3 Core Specification Version 1.0," W3C, Tech. Rep., April 2004, [Accessed: March 2021]. [Online]. Available: https://www.w3.org/TR/DOM-Level-3-Core/

[102] D. Linner, "Instant Synchronization of States in Web Hypertext Applications," PhD Thesis, Electrical Engineering and Computer Science, Technische Universität Berlin, January 2012.

[103] A. Lee. The Firebase Blog: One Million. Google Inc. [Accessed: March 2021]. [Online]. Available: https://firebase.googleblog.com/2014/07/one-million.html

[104] G. Soltis. (2014, Sep) Scale By the Bay 2014: Designing Highly Concurrent, Multi-Protocol, Multi-Tenant Services in Scala. FunctionalTV. [Accessed: March 2021]. [Online]. Available: https://www.youtube.com/watch?v=WoLEREfTabM

[105] I. Fette and A. Melnikov, "RFC6455: The WebSocket Protocol," Internet Engineering Task Force (IETF), Tech. Rep., December 2011, [Accessed: March 2021]. [Online]. Available: https://tools.ietf.org/html/rfc6455

[106] K. Birman and T. Joseph, "Exploiting Virtual Synchrony in Distributed Systems," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, pp. 123–138, 1987.

[107] Firebase Documentation - JavaScript - firebase.database - transaction. Google Inc. [Accessed: March 2021]. [Online]. Available: https://firebase.google.com/docs/reference/js/firebase.database.Reference#transaction

[108] Baumann, Tim. Operational-Transformation / ot.js. [Accessed: March 2021]. [Online]. Available: https://github.com/Operational-Transformation/ot.js/

[109] Firepad. Google Inc. [Accessed: March 2021]. [Online]. Available: https://firepad.io

[110] M. Lehenbauer. (2013, April) Announcing Firepad - Our Open Source Collaborative Text Editor. Google Inc. [Accessed: March 2021]. [Online]. Available: https://firebase.googleblog.com/2013/04/announcing-firepad-our-open-source.html

[111] Modern Development Environment Delivered. Koding Inc. [Accessed: March 2021]. [Online]. Available: https://www.koding.com/

[112] P. B. Landers, "Speakur: leveraging web components for composable applications," MASc Thesis, University of Texas at Austin, May 2015.

[113] S. Krusche and B. Bruegge, "Model-Based Real-Time Synchronization," in *CVSM14: International Workshop on Comparison and Versioning of Software Models*, 2014.

[114] A. Taivalsaari, T. Mikkonen, and K. Systä, "Liquid software manifesto: The era of multiple device ownership and its implications for software architecture," in *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*. IEEE, 2014, pp. 338–343.

[115] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Yjs: a Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types," in *Engineering the Web in the Big Data Era*. Springer, 2015, pp. 675–678.

[116] Jahns, Kevin. yjs/yjs - Peer-to-Peer Shared Types. [Accessed: March 2021]. [Online]. Available: https://github.com/yjs/yjs/

[117] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, "Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types," in *Proc. of the 19th Int. Conf. on Supporting Group Work*. ACM, 2016, pp. 39–49.

[118] Jahns, Kevin. y-xml - XML Type for Yjs. [Accessed: March 2021]. [Online]. Available: https://github.com/y-js/y-xml

[119] L. André, C.-L. Ignat, and G. Oster, "Collaboration Over Wiki Content," in *IWCES'12: The 12th Int. Workshop on Collaborative Editing Systems*, 2012.

[120] C.-L. Ignat, L. André, and G. Oster, "Enhancing Rich Content Wikis with Real-Time Collaboration," *Concurrency and Computation: Practice and Experience*, 2017.

[121] C. Chedeau. (2018) Reacts Diff Algorithm. [Accessed: March 2021]. [Online]. Available: https://calendar.perfplanet.com/2013/diff/

[122] J.-P. Voutilainen, T. Mikkonen, and K. Systä, "Synchronizing Application State Using Virtual DOM Trees," in *International Conference on Web Engineering*. Springer, 2016, pp. 142–154.

[123] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[124] J. Lengstorf and P. Leggetter, *Real Time Web Apps*. Apress, 2013.

[125] B. Shneiderman, "Response Time and Display Rate in Human Performance with Computers," *ACM Computing Surveys (CSUR)*, vol. 16, no. 3, pp. 265–285, 1984.

[126] B. Bermes, *Lean Websites*. SitePoint Pty. Ldt., 2015.

[127] PouchDB, the JavaScript dabatase that Syncs! [Accessed: March 2021]. [Online]. Available: http://pouchdb.com/

[128] Meteor: A better way to build apps. Meteor Development Group. [Accessed: March 2021]. [Online]. Available: https://www.meteor.com/

[129] RethinkDB - The Open-Source Database for the Realtime Web. [Accessed: March 2021]. [Online]. Available: https://www.rethinkdb.com/

[130] (2020, Feb) Removed and deprecated functions in Rational Application Developer. IBM. [Accessed: March 2021]. [Online]. Available: https://www.ibm.com/support/pages/removed-and-deprecated-functions-rational-application-developer

[131] (2020, September) Extrinsic Functions - MATLAB & Simulink. The MathWorks, Inc. [Accessed: March 2021]. [Online]. Available: https://www.mathworks.com/help/simulink/ug/calling-matlab-functions.html

[132] (2001) Xerces-J API. Apache Software Foundation. [Accessed: March 2021]. [Online]. Available: https://xerces.apache.org/xerces-j/apiDocs/index.html

[133] D. Sun and C. Sun, "Operation context and context-based operational transformation," in *Proc. of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*. ACM, 2006, pp. 279–288.

[134] L. Yu, W. Xiao, C. Chi, L. Ma, and H. Su, "Test Case Generation for Collaborative Real-time Editing Tools," in *COMPSAC 2007: Proc. of 31st Annual Int. Computer Software and Applications Conf.*, vol. 1. IEEE Computer Society, 2007, pp. 509–516.

[135] (2013) Drive: Know Your Limits. Google Inc. [Accessed: March 2021]. [Online]. Available: https://docs.google.com/document/d/1E7sT_BMkFOXR4EsDMUUhOgeD6BrEjFzS8fHv8tNrfPU/edit

[136] Lehenbauer, Michael. (2015, March) OT Question - Firepad Issue #163 - firebase/firepad - Github. [Accessed: March 2021]. [Online]. Available: https://github.com/firebase/firepad/issues/163

[137] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 2015.

[138] G. Rauch. Socket.IO. Socket.IO. [Accessed: March 2021]. [Online]. Available: https://socket.io

[139] Socket.IO. Socket.IO. [Accessed: March 2021]. [Online]. Available: https://github.com/socketio/socket.io-redis

[140] K. J. Nevelsteen, "'Virtual World', Defined from a Technological Perspective, and Applied to Video Games, Mixed Reality and the Metaverse," *arXiv:1511.08464*, 2015.

[141] Firefox Nightly Builds. [Accessed: March 2021]. [Online]. Available: https://www.mozilla.org/en-US/firefox/channel/desktop/#nightly

[142] Chromium - The Chromium Projects. [Accessed: March 2021]. [Online]. Available: https://www.chromium.org/Home

[143] B. Jones and M. Goregaokar, "WebXR Device API," W3C, Tech. Rep., July 2020, [Accessed: March 2021]. [Online]. Available: https://www.w3.org/TR/webxr/

[144] Unity Real-Time Development Platform - 3D, 2D VR & AR Engine. Unity Technologies. [Accessed: March 2021]. [Online]. Available: https://www.unity.com/

[145] Unreal Engine - Real-Time 3D Creation Platform. Epic Games Inc. [Accessed: March 2021]. [Online]. Available: https://www.unrealengine.com/

[146] A-Frame - Building Blocks for the VR Web. Mozilla Corporation. [Accessed: March 2021]. [Online]. Available: https://aframe.io/

[147] C. Gadea, B. Solomon, B. Ionescu, and D. Ionescu, "A Collaborative Cloud-Based Multimedia Sharing Platform for Social Networking Environments," in *ICCCN 2011: Proc. of 20th IEEE Int. Conf. on Computer Communication Networks*. IEEE Computer Society, August 2011, pp. 1–6.

[148] A. Prakash and M. J. Knister, "A Framework for Undoing Actions in Collaborative Systems," *ACM Trans. on Computer-Human Interaction (TOCHI)*, vol. 1, no. 4, pp. 295–330, 1994.

[149] H. Shen and C. Sun, "Flexible Merging for Asynchronous Collaborative Systems," *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, pp. 304–321, 2002.

[150] PageCloud: Drag and Drop Website Builder. Pagecloud Inc. [Accessed: March 2021]. [Online]. Available: https://www.pagecloud.com/

[151] S. Raval, *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology*. O'Reilly Media, 2016.

[152] M. Raczyski. (2020, February) What Is The Fastest Blockchain And Why? Analysis of 43 Blockchains. Aleph Zero Foundation. [Accessed: March 2021]. [Online]. Available: https://alephzero.org/blog/what-is-the-fastest-blockchain-and-why-analysis-of-43-blockchains/

[153] IPFS is the Distributed Web. Protocol Labs. [Accessed: March 2021]. [Online]. Available: https://ipfs.io/

[154] Take a Look at PubSub on IPFS. Protocol Labs. [Accessed: March 2021]. [Online]. Available: https://ipfs.io/blog/25-pubsub/

[155] (2020, September) Create Help for Classes - MATLAB & Simulink. The MathWorks, Inc. [Accessed: March 2021]. [Online]. Available: https://www.mathworks.com/help/matlab/matlab_prog/create-help-for-classes.html