Mobile Modeling with Real-Time Collaboration Support

Max Härtwig* **and Sebastian Götz‡**

*Technische Universität Dresden, Germany; Now at Google, Switzerland

‡Faculty of Computer Sciences, Technische Universität Dresden, Germany

ABSTRACT

Modeling is an essential discipline, especially in software engineering. Students and developers alike employ models to describe systems, capture requirements, and communicate with other teams. For that purpose, UML diagrams are often employed. Meanwhile, mobile devices increased in prevalence and popularity and flexible work arrangements were introduced in a larger number of workplaces. Effective collaboration is more important than ever. However, tools have not kept up with these developments. To the best of our knowledge, there is no semantics-aware mobile modeling approach that supports collaboration in real time: a gap we aim to close in this paper.

For this, we investigate existing approaches for mobile modeling and their shortcomings with a particular focus on conflict-free, real-time collaboration. Based on our findings, we conceptualize and implement *CoMod*, a proof of concept allowing users to collaboratively edit UML class diagrams in real time. The system consists of a Flutter-based client application for Android and iOS and a Node.js-based server executable. These components utilize conflict-free replicated data types (CRDT) to merge participants' changes. Moreover, *CoMod*'s feasibility is evaluated using two case studies (classroom and brainstorming) investigating the system's scalability and performance characteristics.

KEYWORDS Android, collaboration, CRDT, Flutter, mobile, modeling, iOS, real-time, UML, Yjs

1. Introduction

Modeling is an essential discipline in many fields, such as engineering, natural sciences, and psychology (Rothenberg 1990). Models describe or represent systems on an abstract level and can be used for discussions, analyses, and other purposes. They are particularly useful in software engineering and are employed throughout the industry. Models capture requirements and aid in the communication between teams, thus increasing quality and productivity by reducing ambiguity and misunderstandings. With sufficient semantic information, some kinds of models can even be used to generate and verify code.

Traditionally, the process of modeling has taken place in an office setting on whiteboards, or digitally on laptops and desktop computers. However, mobile devices such as smartphones and tablet computers can also be used for modeling purposes, making it accessible to a larger group of users as more people have access to smartphones than laptops and desktop computers. Further, they are usually cheaper to buy and replace. Smartphones and tablets also benefit from the fact that they can be taken anywhere and used near the described problem domain or right where the model is needed. Mobility and context can be especially beneficial for modeling in certain problem domains (Vaquero-Melchor et al. 2017). Furthermore, mobile devices commonly have access to a wide range of sensors, such as camera and GPS, and may feature specialized input methods like styluses and multi-touch gestures than can result in an improved usability.

The aspect of collaboration has usually involved multiple people discussing changes while one person was carrying them out on a whiteboard or a shared screen. While being effective, this approach exhibits a couple of shortcomings. Collaborators have to be present in the same room or online meeting and cannot all edit the model at the same time. A better solution would be the adoption of tools that allow modeling to take

JOT reference format:
Max Härtwig and Sebastian Götz. *Mobile Modeling with Real-Time Collaboration Support*. Journal of Object Technology. Vol. 21, No. 3, 2022.
 http://dx.doi.org/10.5381/jot.2022.21.3.a2

place in an asynchronous and collaborative manner as discussed in (Franzago et al. 2018; Masson et al. 2017). This is especially important for organizations with a rising number of distributed teams and colleagues working from home. This process is further accelerated by the COVID-19 pandemic.

With the increasing prevalence of mobile devices and flexible work arrangements, mobile modeling applications could be valuable instruments for teams to collaborate and might even become essential tools in a world where remote work is the norm rather than the exception. Several mobile modeling applications are available today, but they usually lack vital features that a viable collaborative mobile modeling solution should have. Some general-purpose applications support real-time collaboration, but do not provide the means needed for modeling software systems as we will show in this paper.

To close this gap, we introduce a novel concept and prototypical implementation for semantics-aware mobile modeling applications with real-time collaboration support called *CoMod*. We evaluate the approach using a case study, which represents two typical usage scenarios: a classroom and a brainstorming setting. Our prototype and evaluation setup is available as open-source and can be found on GitHub[1].

The remainder of this paper is structured as follows. First, section 2 outlines requirements that serve as a reference for the examination of existing mobile modeling applications in section 3. Section 4 describes the concept of a new application satisfying these criteria without depending on specific platforms or frameworks. Subsequently, section 5 portrays the implementation of *CoMod*, a proof of concept based on the aforementioned concept. This is followed by section 6, an evaluation of the solution, its viability and scalability by means of a case study. Finally, section 7 summarizes the main findings and concludes this paper.

2. Requirements for Real-time Collaborative Mobile Modeling

This section enumerates a set of requirements that viable mobile modeling applications should satisfy and explains why each one is essential. The preceding literature analysis – in particular (Brunschwig et al. 2022; Franzago et al. 2018) – revealed different criteria that were used by different authors.

Availability: Mobile modeling apps are mainly used on mobile devices which usually run either Android or iOS. A good app should be available on both platforms to enable all members of a project to use it.

Semantics: A model should not just be a drawing or a diagram. A good modeling app understands the model's contents and can discern its individual components. Operations performed on the model should always yield a valid result, e.g. a deleted class in a UML class diagram should not leave relationships without a target.

Performance: Modeling apps should be fast and responsive despite being limited by a mobile CPU. Users should not have to wait for basic actions and expensive work needs to be performed in the background to ensure that the UI never feels sluggish.

Usability: Mobile devices often have relatively small screens and require apps to adapt their UI accordingly. Basic interactions should be discoverable, easy to perform, and require as few steps as possible. The aspect of accessibility also belongs to this category.

Collaboration: Modeling is usually performed by multiple members of a team collaborating with each other. Mobile modeling apps should enable them to do so by either allowing documents to be shared or, even better, worked on in real time.

Offline support: Enabling real-time collaboration is not a trivial task and usually requires an active internet connection. However, mobile devices often have to deal with unreliable cellular connections or no connection at all when wifi is not available (e.g. in the case of most tablets). Modeling apps should account for that and allow models to be edited offline.

3. Related Work

Based on the extensive survey on mobile modeling applications in (Brunschwig et al. 2022), this section examines three representative mobile modeling applications. The selection includes solutions allowing the design of models that can be represented as UML class diagrams.

3.1. Astah UML

Astah UML[2] (formerly *JUDE*) is a UML modeling software for iOS and PC. Its iPad version supports the creation of UML class diagrams by allowing text, notes, classes, and interfaces to be placed on a canvas. These objects can be color-coded and connected using lines and arrows. Details such as the type's name, attributes, and operations can be edited via an overlay that appears when needed. The desktop versions offer more features and also allow other types of UML diagrams to be created.

Availability: The mobile version of Astah is only available on iPad (named *Astah UML pad*). The desktop version comes in two variants, *Astah UML* and *Astah professional*, and runs on Windows, macOS, and Linux.

Semantics: Astah recognizes the individual parts of class diagrams. Documents can be exported as XMI files to be processed by other applications. Code can be imported from and exported to many languages.

Performance: Both the mobile and desktop versions are fast and responsive.

Usability: The limited feature set of the mobile version makes using the app rather simple. Most actions are performed using menus that appear when needed. A bar above the keyboard offers quick access to special characters and the names of data types. The desktop version goes a step further and provides affordances to quickly add a property or operation and even suggests targets when beginning to draw a connection.

Collaboration: Astah lacks support for collaboration. Files can be sent to team members, but they cannot be edited collaboratively.

Offline support: Projects are created locally by default and do not require an active internet connection.

[1] https://github.com/MaxHaertwig/CoMod

[2] https://astah.net/products/

3.2. Lucidchart

Lucidchart[3] is a proprietary diagramming application that focuses on sharing and collaboration. At its core is a canvas upon which different shapes can be placed. Basic shapes include boxes, arrows, and text. An extensive library offers access to more shapes, such as AWS icons, flowcharts, tables, and UML diagrams. All of them can be mixed and matched and placed on the same canvas. Shapes can be formatted and connected with each other using lines or arrows. Many shapes have dedicated areas for text, e.g. the *class* shape accepts text entries for the class's name, its attributes, and its operations.

Availability: Lucidchart offers native apps for Android and iOS and is accessible via a web browser on desktop computers.

Semantics: A major drawback is the absence of semantics. Documents are treated as collections of shapes and are unaware of the semantics of their contents.

Performance: The experience is quite snappy on iOS and does not require the user to wait for any action to complete. However, the version running in the browser might feel somewhat slower than comparable native applications.

Usability: The app is fairly usable and it is easy to get up to speed. The number of presented options is daunting, but not overwhelming.

Collaboration: Other users can be invited to collaborate on documents and any edits appear on others' devices in real time. Contributors can leave comments and associate them with a certain shape to start a discussion.

Offline support: Both mobile apps allow documents to be created and edited while being disconnected from the internet. However, this feature is still in beta for the web version.

3.3. System Designer

System Designer[4] is a free mobile IDE for designing JavaScript systems developed by Erwan Carriou. It allows the design of entire systems, including models, behaviors, instances, and logging. This is done via the Metamodel JavaScript Object Notation (MSON)[5] which can be edited directly. During the process, the individual components of the system are visualized within the app. After a system has been designed, it can be run directly in System Designer and even shows logging output.

Availability: Native apps for Android and iOS are available for download. A progressive web app for desktop computers can be used as well.

Semantics: The specification via MSON enables the application to be aware of every system component on a structural and semantic level.

Performance: Both the native and web versions of the app are performant and immediately respond to user interaction.

Usability: The user interface has a clear structure. Editing system components using the built-in text editor might feel tedious sometimes, but it is fast and features autocompletion.

Collaboration: System Designer does not offer dedicated collaboration features. However, a project can be synchronized via a GitHub repository.

Offline support: None of the application variants requires an active internet connection.

3.4. Summary

The solutions presented above are a small subset of mobile modeling applications available today (Brunschwig et al. 2022). However, each solution represents a certain class w.r.t. our requirements. Applications like *Lucidchart* or *diagrams.net*[6] allow the collaborative creation of nice-looking UML diagrams, but do not understand the models' semantics. Other applications such as *Astah UML pad* can be used to design proper UML models, but they are only available on a small number of platforms or lack certain features developers rely on. Specialized solutions such as *System Designer* are very capable but focus on a specific area, e.g. designing JavaScript systems, and are unsuitable for general-purpose modeling.

In summary, none of the solutions presented in this section nor other available applications at the time satisfy all of our requirements. A gap we aim to close with our approach.

4. Concept

This section describes a concept for an application that satisfies all of the requirements outlined in section 2.

The field of mobile modeling is extensive and impossible to exhaustively cover with a single application. Hence, as a proof of concept, we focus on the domain of UML class diagrams.

4.1. Objectives

Our concept is designed to fulfill the following list of objectives.

UML class diagrams: The app should allow the creation and manipulation of UML class diagrams, including types (i.e. classes, abstract classes, interfaces) and relationships (i.e. associations, aggregations, compositions, associations with associated classes, qualified associations).

Semantics: The app should retain the model's semantics, i.e. it needs to be able to discern the model's individual components and check their validity. Another responsibility is ensuring the consistency of the model while it is being modified, e.g. deleting a type should also delete its relationships and not leave them one-sided.

Availability: The app should run on multiple platforms. As this is mainly targeted at mobile platforms, Android and iOS are the prime candidates. Targeting the web is desirable, but remains optional for this proof of concept.

Collaboration: The app should allow multiple users to collaboratively work on the same model. Modifications need to be non-locking and able to be merged without conflicts that require manual resolution. Users need to be able to edit the model concurrently without having to wait for others. This may cause their local replica to diverge temporarily. To cope with this fact, the app has to implement *eventual consistency*, i.e. all replicas

[3] https://www.lucidchart.com/pages/
[4] https://designfirst.io/systemdesigner
[5] https://designfirst.io/systemruntime/documentation/docs/en/design-your-model.html
[6] https://www.diagrams.net

need to be in the same state once all pending modifications have been transmitted and processed.

Performance: The app needs to be fast and responsive. Mobile devices often lack the most powerful processors due to their small size and thermal constraints. This needs to be accounted for. Expensive computations need to be performed in the background to ensure that the user interface remains responsive at all times. When collaborating, a low latency is desirable. Modifications from other connected participants should appear in real time.

Usability: The app needs to be usable on smartphones. Text should be readable and UI elements have to be large enough to be touched without accidentally triggering an adjacent one. All features have to be accessible via touch without requiring a mouse or a keyboard to be connected.

Offline support: The app has to be usable without an active internet connection. This includes the modification of shared models. Offline changes need to be cached and transmitted once an internet connection is established again. Unreliable connections have to be tolerated without changes being dropped.

4.2. User Interface

In order to supply the features outlined in the first objective of the previous section, the app needs at least four different views:

Models: An editable list of the different models the app has stored. Selecting a model leads to the Model Details view.

Model Details: Shows the selected model's contents, i.e. its types and their relationships. This could be a list or a visual representation. From this view, users should have the option to add, select, or delete types. Selecting a type leads to the Type Details view; a back button leads back to the Models view.

Type Details: Shows the details of a type, including its name, type (class, interface, etc.), supertypes, attributes, operations, and relationships. All these properties need to be editable from this view. A back button leads back to the Model Details view.

Collaboration: Allows users to start or join collaboration sessions. Starting a collaboration session should be possible after having selected a model to edit while joining a remote session should be possible from the Models view.

Fig. 1 depicts what above views might look like when implemented as dedicated screens and how a user might navigate between them. However, these views do not necessarily have to be presented on their own screens. It might also be possible to combine multiple views on a single screen, especially on larger devices such as tablets.

4.3. Data Model

An internal data model is required for the app to keep track of the model's contents at runtime. A UML class diagram can be interpreted as a hierarchy, thus a tree-like structure would be suitable. The tree's root is the UML model itself which in turn contains UML types and relationships. Types contain attributes and operations. Operations contain parameters. As a consequence, the model consists of heterogeneous nodes that, depending on their own type, can only contain certain types of child nodes (cf. fig. 2). Moreover, each type of node may only contain a certain set of properties, e.g. an operation has a name,

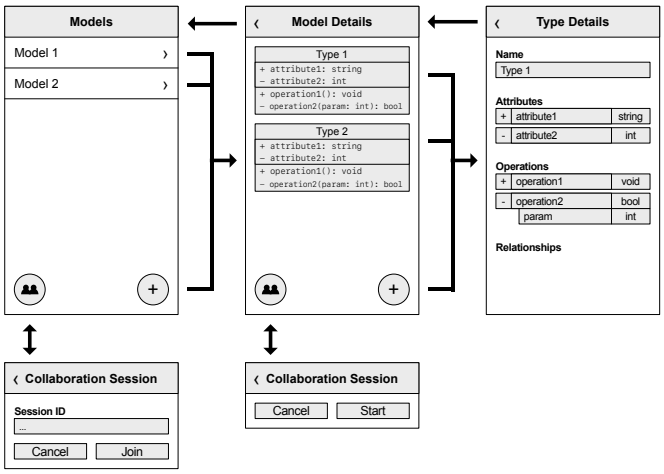

Figure 1 Concept of the user interface.

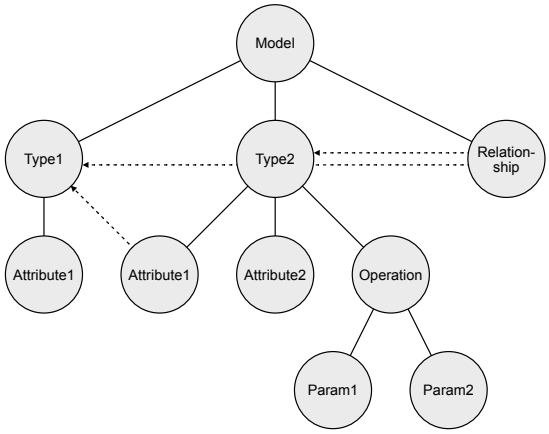

Figure 2 The tree-like structure of the app's data model. The dotted lines indicate the overlay graph.

a visibility annotation, and a return type. Table 1 lists all entities along with their properties and possible children types.

Some relations between different nodes, i.e. attributes based on types in the model and the types involved in a relationship, cannot be represented by a strictly tree-based structure. Both attributes and relationships reference types based on their unique IDs. This approach can be interpreted as an overlay graph (cf. dotted lines in fig. 2). This is important for the model's consistency, which must be maintained at all times. When a type is deleted, all referencing attributes' data types are changed to *string* (as the type they used to reference does not exist anymore) and referencing relationships are deleted.

The data model has to be persisted regularly to preserve its contents across launches of the app. One option is saving all entities as rows in a database with parent-child relations modeled using foreign keys. Another option is to serialize the data model to a binary or text-based format and write it to a file.

4.4. Collaboration

Collaboration is based on the concept of sessions. Each session is tied to a model (identified by its ID). When a model is being

Component	Properties	Possible children types
UML Model	ID	UML Type, UML Relationship
UML Type	ID, Name*, Type (class, abstract class, interface), Supertypes	UML Attribute, UML Operation
UML Attribute	Name*, Visibility, Data Type	
UML Operation	Name*, Visibility, Return Type	UML Operation Parameter
UML Operation Parameter	Name*, Data Type	
UML Relationship	Label*, From Type ID, To Type ID, Multiplicity	

Table 1 A list of the data model's components and their properties and possible children types. Mergeable properties are marked with a star (*).

edited, a collaboration session may be started. This generates a link that others can use to join the session. Starting a session should automatically join an existing session of the same model if one is already in progress. A collaboration session may be left by any user at any point in time.

When joining a session, the session's model has to be sent to the joining device. If a version of the model is saved locally, it needs to be *synchronized*; if no local copy exists, the entire model has to be downloaded. The process of synchronization (sync) includes receiving new modifications from other clients as well as sending local changes that occurred after the session was last left to other clients. Thus, the data structure also needs to support computing a difference between two versions. Being able to identify the difference also enables only sending specific pieces of data over the network and saving bandwidth as a result. This mechanism is also used when clients reconnect to a session.

When modifying a model, each edit has to be transmitted to the other clients of the session where they are incorporated into their respective data models. Thus, the data model needs to be able to emit and incorporate incremental changes. Processing remote changes also has to be *idempotent*, i.e. it must be possible to apply a change multiple times without changing the result beyond the first application. This is important, because the same modifications might originate from several devices.

4.4.1. Conflict handling

If multiple clients concurrently modify the same model, conflicts will inevitably arise, e.g. one participant extends a type's name while another adds a prefix to it. Possible solutions include locking entities while they are being edited or letting participants manually resolve conflicts, but it would stand in conflict with the objective of real-time collaboration. Instead, conflicts need to be automatically and deterministically resolved by the data model.

One data structure that is able to automatically resolve conflicts are Conflict-Free Replicated Data Types (Shapiro et al. 2011). In theory, CRDTs also fulfill the requirements mentioned earlier in this section: they can be combined to form complex structures (such as trees), are able to compute differences, and can incorporate changes without producing conflicts that need to be resolved manually. A number of CRDTs with different implementations exist, but most of them have some properties in common. Individual elements of the CRDT have unique identifiers. This allows them to be tracked across different replicas, even if their other characteristics have been changed. Deletions are usually modeled as tombstones that continue to be part of the data structure. This allows clients to refer to deleted elements after their deletion (e.g. when they receive modifications from a client that did not yet know of the deletion). Versions and differences are determined using *version vectors*[7].

Some of the model's details should not merge their values. They always have to be equal to a value of a set of allowed values, e.g. an operation's visibility needs to equal `public`, `packagePrivate`, `protected`, or `private`, but not a mix of them. This is also true when they are modeled as integers instead of strings. Instead, these properties should be merged using the *last writer wins* principle (LWW), i.e. only the last modification is kept. When using this method of conflict resolution, the last modification is not determined by temporal timestamps, but by comparing the CRDTs' *Lamport timestamps*[8]. If, according to these timestamps, two modifications occurred simultaneously, one of them is chosen deterministically. However, all other components should merge their values. The entire hierarchy can be safely merged; new or deleted types, attributes, operations, parameters, and relationships need to be inserted into or removed from other models. Text directly editable by users such as type names or relationship labels can be merged on the per-character level. This allows multiple users to simultaneously edit the same text without any changes being dropped.

An alternative to CRDTs is the Operational Transformation approach (Ellis & Gibbs 1989). CRDTs have been chosen for this project because, in contrast to operational transformations, high-quality frameworks reducing the development and maintenance effort are available.

4.4.2. System architecture

A client-server model is the most suitable architecture for this project. The server keeps track of each session's current model state. This allows it to directly send complete or partial models to connecting clients without having to request that information from an already connected client. This way, new clients can even receive updates when no other clients are connected at that particular point in time. In order to support real-time collaboration, clients need to be able to send and receive edits with as little latency as possible. Therefore, they should open a bidirectional communication channel to the server enabling them to send and receive messages without resorting to long polling.

This model is not limited to a single server. If the need arises, multiple servers may be employed to handle clients' requests. Each server should handle a distinct set of sessions (identified by their respective models' IDs).

[7] https://en.wikipedia.org/wiki/Version_vector
[8] https://en.wikipedia.org/wiki/Lamport_timestamp

Message Type	Properties	Notes
Connect Request	Model ID, [state vector]	Sent when connecting to a session. Server responds with a Connect Response.
Connect Response	[State vector, update]	Sent in response to a Connect Request.
Sync Request	Update	Optional; sent when the client has information the server does not yet have.
Sync Response		Sent in response to a Sync Request.
Update	Data	Sent by either client or server.

Table 2 A list of the message types used during the client-server communication. Optional properties in [brackets].

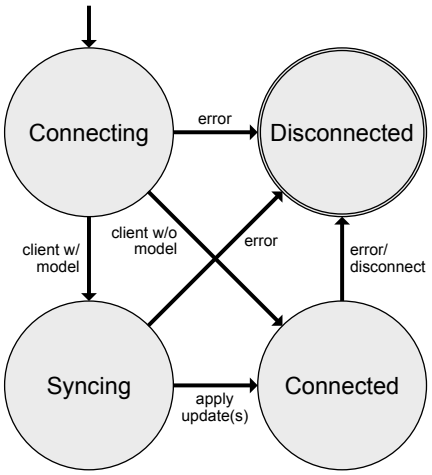

Figure 3 Client-server communication state diagram.

4.4.3. Client-server communication

A client's connection to the server can be in one of four states: *connecting*, *syncing*, *connected*, *disconnected*. Initially it is *connecting* and a bidirectional communication channel is opened. The channel transmits messages of different types, each one with its own semantics and properties (cf. tab. 2). Requests are always sent from clients to the server, responses are always sent from the server to clients. *Update* is the only message type that can be sent by either the server or clients.

Fig. 3 shows the state diagram of this process. The first step is connecting to a session. For that purpose, a **Connect Request** that contains the model's ID is sent. If the client joins without a local version of the model (e.g. via a link), the state vector property remains empty. The server looks up the session using the provided ID and sends back the session model's data. If the server cannot find the session, the connection becomes *disconnected*. If the client has a local model version, the state vector property is populated to enable the server to compute their difference. If the server has no model, an empty **Connect Response** is sent back. Otherwise, the provided version vector is compared to the version vector of the server's model. If the server's model contains changes the client has not seen yet, they are sent as an update in a **Connect Response**. In any case, the server model's state vector is also sent as part of the response. The connection transitions to *syncing*.

The client incorporates the server's update (if any) into its own data model. If the server also provided a version vector, the connection becomes *syncing*, otherwise it becomes *connected*. A *syncing* client is responsible for providing the server with an update. The update is generated by comparing the server's state vector to one of the local model and sent to the server as a **Sync Request**. The server incorporates the update into its own data model, broadcasts the update to other connected clients, and confirms the request by responding with an (empty) **Sync Response** and the connection transitions to the *connected* state.

Whenever a *connected* client has modifications to report, an **Update** message is sent to the server which is immediately broadcast to all other connected clients. Likewise, the client may at any point in time receive **Update** messages that originated from other clients. Their contained data needs to be incorporated into the local data model; a response is not intended.

A connection may become *disconnected* if an invalid message is sent, the server is instructed to join an unknown session, or the network conditions interrupt the communication channel. When that happens, no more messages are exchanged. The client may try to reestablish the connection by opening a new communication channel and transitioning through the *connecting* and *syncing* states again. This ensures that any edits that occurred while the connection was interrupted are not lost.

5. Solution

This section takes a detailed look on *CoMod*, the solution that was implemented based on the concept described in the previous section. It serves as a proof of concept demonstrating that it is indeed possible to build a mobile modeling application with real-time collaboration support. *CoMod* consists of a cross-platform application running on Android and iOS as well as a server executable facilitating the real-time collaboration features. Both components are examined separately, followed by a description of how they communicate with each other.

5.1. Client

The mobile app was implemented using *Flutter*[9], a UI toolkit made by Google to build fast and responsive cross-platform applications. It is written in *Dart*[10] and uses a custom rendering engine called *Skia* to draw its user interface. The UI of a Flutter application is structured as a tree of widgets. Widgets may be stateful (i.e. they have to be redrawn whenever their state changes) or stateless (they remain constant during their lifetime). In addition to the business logic implemented in Dart, the app also relies on a JavaScript subsystem running all the logic related to the CRDT-based data model, which is explained later on.

5.1.1. Walkthrough

This section is structured as a guided tour through all of *CoMod*'s screens and features.

[9] https://flutter.dev
[10] https://dart.dev

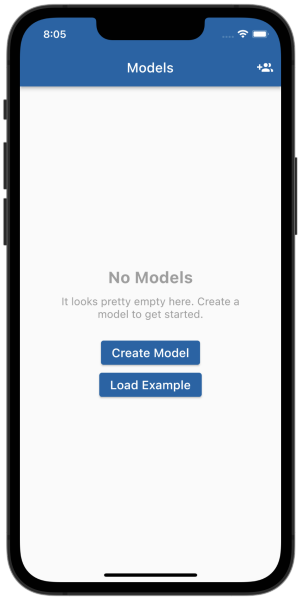

Figure 4 Models screen (empty).

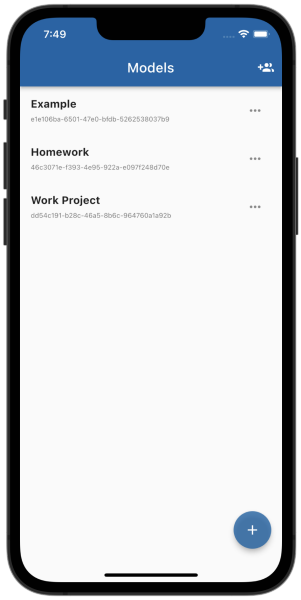

Figure 5 Models screen (non-empty).

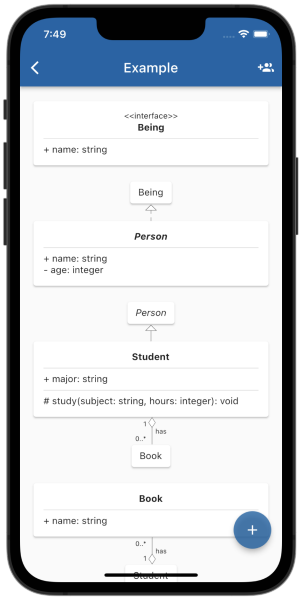

Figure 6 Model details screen.

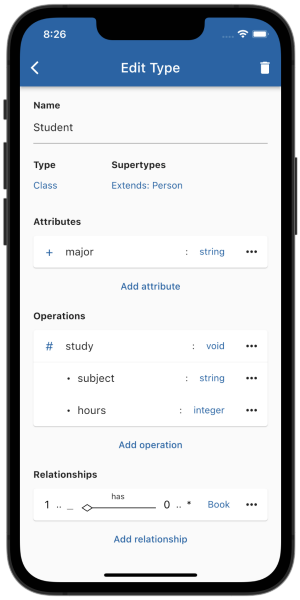

Figure 7 Type details screen.

The application has been designed to accommodate users who are familiar with UML class diagrams. The UI has been kept sufficiently minimal and intuitive to be used without requiring prior training. It uses a stack-based navigation approach, i.e. screens are pushed onto and popped off the stack when navigating forwards or backwards, respectively.

Models screen When *CoMod* is launched for the first time, an empty *Models screen* (cf. fig. 4) is shown. It allows the user to create a model from scratch or load an example to help them explore the application. After at least one model has been created (either manually or by loading the example), the screen shows the list of models (cf. fig. 5). Each model has a name (entered by the user) and a randomly assigned universally unique identifier (UUID). The latter is shown to help users determine which models share a collaboration session, but may be hidden in the release version. Existing models may be renamed or deleted using the ellipsis button (**...**) on the right-hand side of the screen. Tapping on a model navigates to the *Model details screen*. New models may be created using the plus button (**+**) at the bottom-right. A button in the navigation bar opens a dialog that allows users to join a collaboration session by entering its ID or pasting the link to the session.

Model details screen The *Model details screen* (cf. fig. 6) shows an overview of the model's contents as a vertically scrollable list. Each type's UML class diagram card is shown, containing its attributes and operations. Tapping on a card performs a navigation to the respective type's *Type details screen*. Relationships to other types are shown as horizontally scrollable views. New types may be created using the plus button in the bottom-right corner. The chevron button (**<**) on the left-hand side of the navigation bar navigates back to the *Models screen*.

A noteworthy detail is that relationships are only shown as indicators, i.e. the line or arrow does not lead to the relationship's target in the list, but to a smaller placeholder only showing its name. Generalizations and realizations are shown above, associations, aggregations, and compositions below the type's card. The latter kind may also display a label and/or multiplicity annotations. This approach allows types to be viewed in their respective contexts while keeping the screen clear and concise, even if the model contains dozens of different types.

The navigation bar's right-hand button can be used to access the app's collaborative features. A collaboration session may be started from there. During this process, a message will show with an option to copy a link to the session or an error message, should the connection fail. The link is meant to be shared with other participants. They may either open it on their mobile devices, paste it in the appropriate dialog on the *Models screen*, or join via the collaboration button if they have a local version of the model. If a session is in progress, the button offers the options of copying a link to the session or leaving it altogether.

Type details screen The *Type details screen* (cf. fig. 7) shows a type's name, type, supertypes, attributes, operations, and relationships. All of these properties may be edited and the type may be deleted from this screen as well.

The type's name is entered via a text field that only allows characters valid for an identifier in most programming languages. The button for the type's type (class, abstract class, interface) opens a drop-down menu which allows exactly one option to be chosen (like a radio button). The supertypes button also presents a drop-down menu, but allows multiple options to be selected (like checkboxes).

The type's attributes are displayed as a list with a button to add a new one on the bottom. Each row in this list represents a single attribute with controls to modify its visibility, name, and data type. In order to conserve space, only the visibility's corresponding symbol is displayed on the button (e.g. + for

public). However, when it is used to show a drop-down menu, the symbol along with its corresponding label is shown. This allows the row to look just like an entry in a UML class diagram while still giving users sufficient information during the editing process. The data type button allows choosing from a list of primitive data types, such as boolean or string, and other types (classes and interfaces) part of the model. The ellipsis button on the right-hand side can be used to move the attribute to a different position within the list or to delete it altogether.

The type's operations are displayed in a similar fashion. The main difference is that operations may have a variable number of parameters. These are displayed below the main row while still visually belonging to their operation's row. The ellipsis button has the additional option of adding a new parameter. When an operation is deleted, all its parameters are deleted, too.

The type's relationships are displayed similarly as well. Their rows' centerpiece is a drop-down menu that allows the selection of the relationship type (association, aggregation, composition, association with class, qualified association). A matching icon is displayed next to the options and after a selection has been made. A text field above may be used to assign a label to the relationship. If the relationship is an association with a class, the label is replaced by a button allowing the selection of the association class. The relationship's multiplicity annotation may be edited using the text fields to the sides. The button to the right can be used to select the relationship's target. The ellipsis button offers the same options as for attributes and operations.

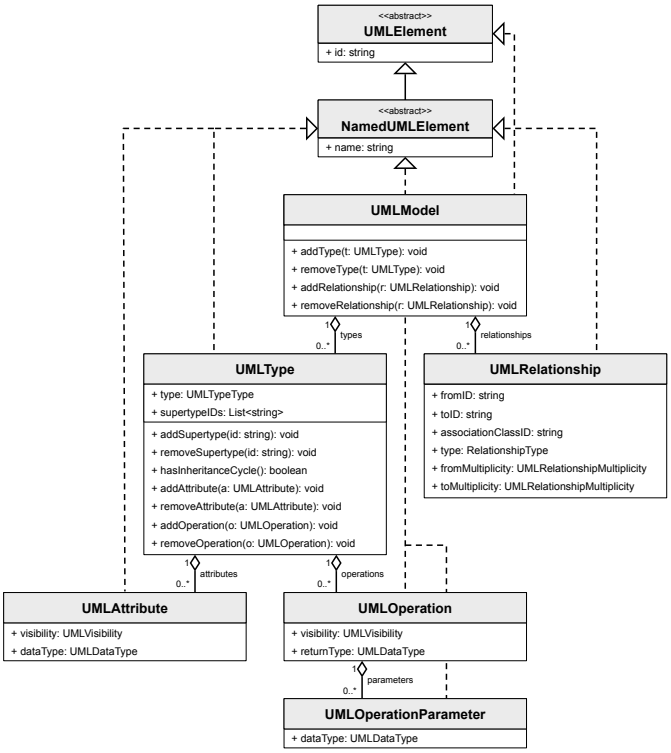

Figure 8 Simplified UML class diagram of *CoMod*'s data model.

5.1.2. Data model

CoMod's data model is based on a number of classes that form an in-memory object graph. Fig. 8 shows a simplified class diagram of the classes involved. The model is kept separate from the user interface and has no dependencies on the visual layer. This allows it to be tested separately and reused should the UI be rewritten in the future. It could even be put into a dedicated package. In fact, the user interface is a function of the model. Whenever part of the model changes, the associated parts of the UI are redrawn. This functional approach reduces the amount of mutable state the visual layer needs to hold and thus, the potential for discrepancies.

As elaborated in the corresponding concept section, the model contains types and relationships, types contain attributes and operations, and operations contain parameters. All classes implement the *UMLElement* interface, i.e. they can be identified by a unique identifier. All model components, except the model itself, also implement the *NamedUMLElement* interface which requires them to have a `name` property. All parent types (containing other types) have methods for adding or removing contained types. The reason is that the contained types are stored in hash maps as opposed to simple lists. This allows them to be efficiently accessed using their ID. Removing a contained type often requires additional logic to ensure the model remains consistent.

Model consistency The model is kept consistent at all times. Whenever a relationship is added, it points to a specific type instead of having no target. Whenever a type is deleted, it is removed as a supertype from all other types and all relationships it is part of are removed as well. Moreover, all attributes, operations, and parameters that use it as their data type are converted to *string*. If, for some reason, a referenced type cannot be found, it is hidden from the UI, as if it did not exist. This behavior is consistent with the approach of removing relationships involving deleted types. Furthermore, all changes related to the deletion of a type are performed as part of a single transaction, i.e. the changes are being applied to the data model atomically to ensure that they cannot be applied partially on other clients.

Nevertheless, it is still possible for users to enter inconsistent information such as inheritance cycles or invalid multiplicities (e.g. `3..1`). In such cases, the data model accepts the changes and shows them in the UI, but marks them as inconsistent, usually with a bright red font. This ensures that all user modifications are being saved while warning users that they have entered inconsistent data.

5.1.3. JavaScript subsystem

CoMod makes use of the *Yjs* framework[11] (Nicolaescu et al. 2015, 2016) providing CRDTs to facilitate its real-time collaboration features. It is a JavaScript-based CRDT implementation, thus, it needs a JavaScript environment to run. Despite this limitation, the framework has been chosen because of its extensive feature set (supporting nested XML structures), solid performance[12], and quality of documentation. It fulfills the requirements for the data model laid out in section 4.4: it can compute the difference between two model versions (using their respective version vectors) and is able to

[11] https://github.com/yjs/yjs
[12] https://github.com/dmonad/crdt-benchmarks

generate and incorporate incremental updates.

Data model The JS environment holds a separate instance of the data model in memory. It is a tree-based model resembling an XML structure. For that purpose, Yjs offers two dedicated node types: `YXMLElement` and `YXMLText`. The former is a generic node in the XML tree that may contain attributes and child nodes while the latter is a type of mergeable string that may be put as a leaf node in XML trees. This structure was chosen because the app's data structure is expressible as an XML document and a purely list-based structure would have been much harder to work with. The native code's data model is essentially mirrored within the JS environment. Each instance of a data model class (cf. section 5.1.2) is represented as a `YXMLElement` with its properties being modeled as XML attributes. Furthermore, each element has an ID attribute that is the same as the corresponding ID property in the native data model. This approach allows changes in either model to be communicated and replicated on the other side. All attributes are string-based, but they are not mergeable (cf. section 4.4.1). Yjs treats them as LWW registers. Concurrent modifications are being resolved deterministically, but unpredictably. Mergeable strings such as type names or relationship labels are modeled as instances of `YXMLText` positioned at index 0 within their parent's children.

Persistence The JS subsystem also has to be involved in persisting the data model to disk. CRDTs assign identifiers and timestamps to their internal components (e.g. each character in a string). Thus, merely saving the data model as an XML file would not work as this information would be missing. Instead, the entire model can be serialized to a binary format and efficiently persisted as a binary blob. The JS environment is sandboxed and does not have access to the file system, so it needs to send the blob to the native code. Likewise, whenever the model is to be loaded from disk, the binary blob has to be sent to the JS subsystem for Yjs to deserialize the data. Afterwards, the loaded model is converted to its XML representation and sent back to the native code to be replicated and displayed in the UI.

Bridge The native code and the JS subsystem cannot access each other's memory directly. JavaScript statements may be executed as strings and results can be received using channels that convey strings. Due to the lack of type checking and proper error handling at that boundary, hereinafter referred to as *bridge*, the communication between the two environments is kept as lightweight as possible. For that reason, a slim API has been defined for the native code to give instructions to the subsystem. The API includes functions for creating a model, loading a model, requesting a state vector, syncing server changes, processing updates, and handling transactions. The methods `insertElement`, `moveElement`, `deleteElements`, `updateAttribute`, and `updateText` can be used to manipulate the model. Each of the API's methods has a specific signature that determines the arguments and data types to be used. Binary data is first converted to its Base64 representation before being sent over the bridge. Whenever the model has an update to emit or it was serialized after processing local or remote changes, a corresponding message is sent over the bridge to be handled by the native code.

Build process The client's JS code is a package that is compiled and tested separately from the native code. The native code calls it whenever necessary. During the compilation process, the TypeScript compiler produces JS files that are subsequently bundled with their dependencies and minified using *Webpack*[13]. The result is a single JS file (~120 KB) that is included as an asset in the Flutter application. This text file's contents are loaded and evaluated when the app is launched.

5.2. Server

CoMod's server facilitates real-time collaboration between clients. It is written in TypeScript and based on the *Node.js*[14] runtime environment. It can be installed and run directly or as a *Docker*[15] container that already includes the runtime environment and dependencies.

When the server is running, it holds a hash map of sessions in memory. Each session is identified by its session/model ID and contains the last known version of its associated data model. The server also makes use of the Yjs framework; thus, it can compare a session's data model version to that of a client and compute the difference and generate an update if necessary. Whenever a client connects, a bidirectional WebSocket connection is established and used for the subsequent communication. An instance of the `Client` class is created keeping track of the client's state and holding a reference to the connection. The first request sent contains the session ID which allows the server to add the client to the corresponding session. Whenever an update is sent by any client, the server incorporates it into the session's data model and broadcasts it to all other clients that are part of the session. Fig. 9 depicts the server's components as a UML class diagram.

Sessions and their associated data models are kept in memory, even if all participants have left the session. This avoids the need of transmitting the entire data model the next time a client connects to that session. In case the server runs low on memory, any data model without any clients connected to its session may be purged. The next time a client connects with its ID, the server will request the data model as part of the syncing step.

5.3. Collaboration

This section elaborates on the technical details of the application's collaboration features. A collaboration session may be started by tapping on the collaboration button in the top right corner of the *Model details screen* (cf. fig. 6). This generates a link containing the session ID that is intended to be sent to other users to allow them to join the session. They can do so by either tapping the link, pasting it into the collaboration dialog on the *Models screen*, or via the collaboration button if they already have a version of the model on their device. Clients that are part of the same session transmit and receive edits in real

[13] https://webpack.js.org
[14] https://nodejs.dev
[15] https://www.docker.com

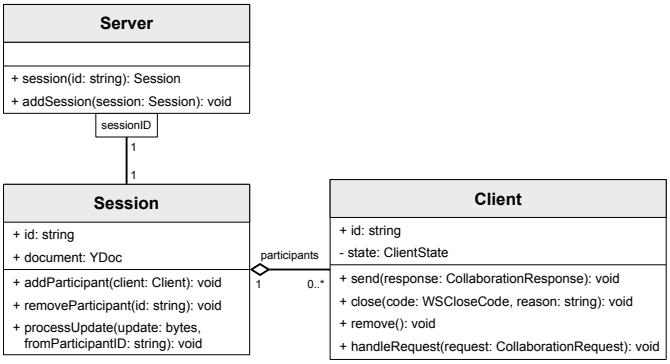

Figure 9 Server UML class diagram.

time. Whenever a client leaves the session, either voluntary or involuntary, it no longer sends or receives any updates. Any changes are recorded locally and transmitted to the server and other clients as soon as this client rejoins the session at a later point in time.

5.3.1. Client-server communication When the collaboration feature is activated, the app opens a *WebSocket* connection (Fette & Melnikov 2011) to the server. This allows the exchange of binary messages over a single full-duplex channel without the need for either party to resort to polling. The connection remains open until it is interrupted or closed from either side. If a client or the server receives invalid data, the channel is closed prematurely.

The messages to be sent are serialized using *Protocol Buffers*[16], a language-neutral mechanism for serializing structured data. A `.proto` file contains the definitions of the message types along with their fields (properties). Implementations for different programming languages (in this case Dart and TypeScript) allow messages to be constructed in code. Each message may be serialized to binary data that can be sent over the wire and deserialized on the other end, regardless of the target system's programming language. The message types and the communication protocol implement the concept described in section 4.4.3.

5.3.2. Client data flow The existence of the JavaScript subsystem that handles the CRDT-related logic leads to a fairly complex data flow within the client application (cf. fig. 10). The native code acts as an intermediary between its JavaScript subsystem and the server. Only the subsystem can provide a state vector, process server responses, and serialize the model. However, it cannot directly communicate with the server. Therefore, any data that needs to be first sent to the server has to be sent over the bridge to the native code.

5.4. Testing

This proof of concept relies on a combination of unit, integration, and end-to-end tests. The share of test-related code is about 14% for the native Dart code, 50% for the JS subsystem code, and 55% for the server code.

[16] https://developers.google.com/protocol-buffers

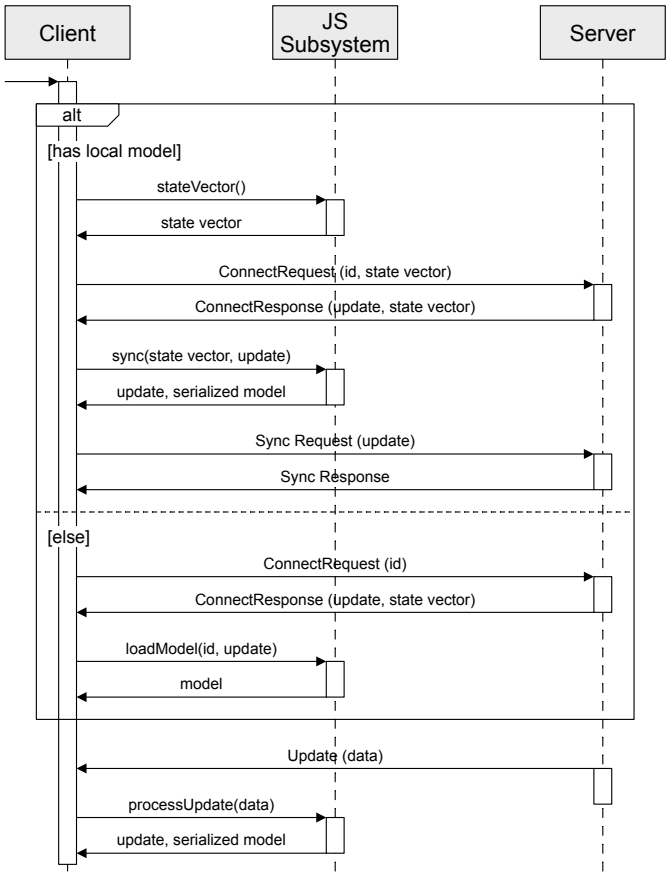

Figure 10 Data flow when starting a collaboration session and receiving an update.

5.4.1. Unit tests The Flutter application's data model is tested using unit tests. Examples include ensuring that the model and its individual components can be loaded from an XML-formatted input string and that attributes and operations are properly converted to strings to be displayed in the UI. All the Dart extensions (code added to standard library types) are covered by unit tests as well. The unit tests for the code running in the JS subsystem make sure that each of the API's methods (e.g. `loadModel()`, `insertElement()`, etc.) behaves as intended. This includes not only checking if the intended changes are reflected in the JS data model, but also ensuring that the appropriate messages are sent back to the native code. The server only contains a limited amount of unit tests as it mainly benefits from integration tests.

5.4.2. Integration tests All Flutter tests involving the user interface belong to this category. They ensure that the app can be launched and its main tasks such as creating models, performing modifications, and collaborating can be performed. However, the application does not communicate with the actual server binary, it instead uses a *fake* that mocks the server's behavior. This allows both components to be tested separately. Meanwhile, the actual server's integration tests check whether it correctly implements the communication protocol and rejects connections with invalid or incomplete data.

5.4.3. End-to-end tests

CoMod's system consists of the Flutter application, its JS subsystem, and the server executable. In contrast to the other test categories, the client application communicates with the actual server instead of a fake. The main challenge of this project's E2E tests is emulating mobile devices to run the application. Unfortunately, the current state of Flutter tooling does not allow the coordination of multiple emulated devices running integration tests. Due to that limitation, the E2E tests start the server, launch a client app, let it create a new model, start a collaboration session, and quit. Afterwards, a second client app is launched on a different device and instructed to join the session. The test succeeds if the second device receives and displays the model created on the first device.

5.5. Extensibility

CoMod implements the concept described in section 4. Nevertheless, it can be extended on multiple levels. Additional platforms (Linux, Windows, Web) may be supported by adding the appropriate targets in Flutter. The support for additional model types is particularly interesting as it would allow the solution's foundation to be (re-)used for editing other model types (e.g. ER (Chen 1976), CROM (Kühn et al. 2014), or custom domain-specific modeling languages (Kelly & Tolvanen 2008)), perhaps even in entirely different modeling domains. The server is able to handle any models that can be represented using the Yjs framework. Because of Yjs's support for XML-based data structures, this includes all models that can be represented using XML documents. Thus, the server's executable can be used as is. Furthermore, the communication protocol is independent from the model's contents as well. So, the server can even handle sessions with different kinds of models at the same time. Code-level changes are only required on the client side. That includes (potentially significant) changes to the UI and the code supporting the data model. However, the JS subsystem does not need to be changed as it exposes an API that allows the manipulation of any XML-like structure using the generic methods `insertElement`, `deleteElements`, `moveElement`, `updateAttribute`, and `updateText`.

6. Evaluation

This section evaluates the solution presented as part of the previous section by means of a case study. Two distinct common cases are presented, followed by a technical analysis of *CoMod*'s scalability. Subsequently, limitations and threats to validity are pointed out and the section is summarized in the end.

6.1. Case Study

The following case study focuses on the collaborative aspects of *CoMod* as this is the main contribution of this paper. The characteristics of the application being used without the collaboration features have been covered by the unit and integration tests described in section 5.4. The correctness of merges and merge conflict resolutions is covered by the unit tests of yjs[17].

This application can be employed in a multitude of different situations. One such case is one person exclusively editing a model while many people receive updates on their devices. This scenario is subsequently referred to as *1WnR* (1 writer, many readers). Examples include an exercise in the context of a class lead by a tutor or a software architect sharing their work with a number of developers in real time.

The other case examined as part of this study is a group with every member contributing to the model, subsequently referred to as *nWnR* (many writers, as many readers). This is a common scenario within group projects in an academic setting or project groups at software companies.

[17] https://github.com/yjs/yjs/tree/main/tests

6.2. Technical Analysis

This analysis evaluates the feasibility of the cases described above by examining how well the system scales to accommodate these cases. Client and server are analyzed separately. Both analyses make use of clock time (as opposed to other metrics such as CPU time) as it better relates to the latency users might perceive. A test data generator has been implemented to provide pseudorandom input data for the experiments.

6.2.1. Test data generator

Both of the following analyses include the transmission of pseudorandom input to the subjects under test. This represents the stream of model updates that is received from other clients. The input is pseudorandom, because it should resemble a series of model changes applied by a user. An entirely random series of changes would less accurately represent how *CoMod* would be used in practice.

Each run of the test data generator (TDG) generates data for a single session based on two parameters: the number of iterations i and the number concurrently editing clients c. At the start, the model consists of a single type. This model serves as the baseline for the first iteration. During each iteration, c concurrent updates are generated. After each iteration, all generated updates are merged with the previous baseline to form a new baseline for subsequent iterations. After all iterations have been processed, the output of $i \times c$ updates is written to a text file. Each update is represented as a Base64 string on its own line and represents a change to the model.

To produce an update, a random number is generated determining the kind of change to apply. There is a 6% chance for a component (type, attribute, operation, parameter, relationship) to be added, a 2% chance for a component to be deleted, a 12% chance for a component's property (e.g. visibility, data type) to be changed, and an 80% chance for a component's name (or label in the case of relationships) to be modified. These percentages have been chosen to let a model grow as more changes are being applied to it (it is likelier for a component to be added to the model than it is for a component to be deleted). The relatively large percentage for textual changes has been chosen, because each keystroke (character addition/deletion) is represented by a separate update.

For each kind of change (insertion, deletion, property, textual), a random component is chosen. In the case of an insertion, it serves as the parent for the new component to be created. If it is the model itself, a type or relationship will be inserted; if it is a type, an attribute or operation will be inserted; and if it is an operation, a parameter will be inserted. The inserted com-

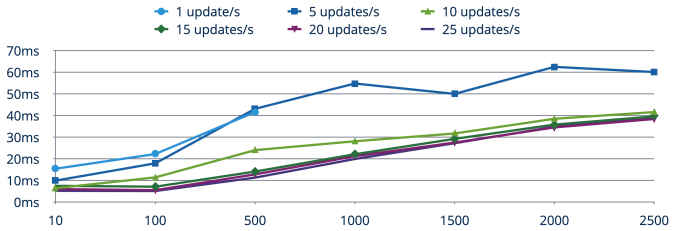

Figure 11 *1WnR* scenario. Average processing time of updates (in ms) measured after the application of a certain number of updates for one editing client.

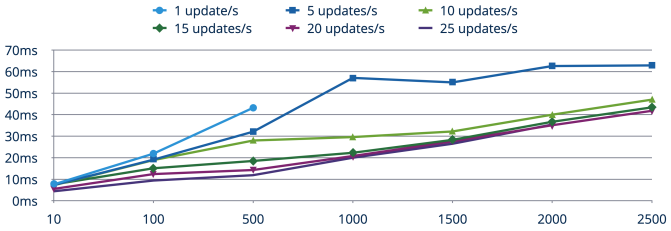

Figure 12 *nWnR* scenario. Average processing time of updates (in ms) measured after the application of a certain number of updates for five concurrently editing clients.

ponent's properties (e.g. visibility, data type) are determined at random. In the case of a deletion, the chosen component and its children are deleted. In the case of a property change, a random property of the chosen component is changed to a random, but different, value. In the case of a textual change, there is a ⅔ chance for a character to be inserted at a random location and a ⅓ chance for a random character to be deleted within the chosen component's name or label. The generator ensures that it only produces changes that are possible for the baseline model, e.g. an empty model does only allow insertions.

6.2.2. Client analysis This analysis evaluates the client application's performance when confronted with an increasing number of incoming updates. For that purpose, a special server has been set up exhibiting largely the same behavior as the actual server with the exception of sending a certain number of updates each second after a client has successfully connected to a session. Its input is a file produced by the TDG. An instrumented instance of the client is launched and instructed to connect to that server. The server sends a certain number of updates equally distributed over the course of each second. The client measures the time it takes for incoming updates to be processed (in ms). All updates are processed by the JavaScript subsystem (cf. section 5.1.3). Thus, they are processed serially on a single thread. Ideally, each update's processing time should be below $1s \div n$ milliseconds where n is the number of updates received per second. If an update's processing time exceeds that limit, the app receives more updates than it can process in real time. In practice, the threshold is lower, because the CPU core the thread is running on may also need to perform other work.

Multiple experiments using two different input files produced by the TDG were carried out on an *iPhone 11 Pro* (2019), each with a different number of updates sent per second. The first input file represents 5000 iterations for one editing client while the other represents 2000 iterations for 5 concurrently editing clients. These two files correspond to the two cases described in section 6.1. Measurements were taken after the application of 10, 100, 500, 1000, 1500, 2000, and 2500 updates and represent the average time (in ms) it took for the application to process the 10 updates preceding the point of measurement. The results can be seen in fig. 11 and fig. 12. An experiment was stopped when it was running too long (in the case of 1 update/s) or when the application became unresponsive (as in the case of 30 updates/s, which is not shown in the figures).

The results show that the more updates were applied before, the longer it takes for subsequent updates to be processed. This difference can be quite significant, e.g. when receiving 5 updates per second, it takes 9.9ms on average to process the first 10 updates, but 60.1ms to process the updates 2491 through 2500. The reason for this finding is the increasing complexity of the model. After the application of each update, the model is serialized to be written to disk. Larger models take a longer time to be serialized. In fact, models are monotonically increasing in size as even the deletion of an element or character leaves a gravestone. The results also show that, on average, it takes slightly longer to process updates originating from 5 concurrent writers than the same number of updates from a single writer. The reason is likely the need for arising conflicts to be resolved. Surprisingly, when a higher rate of updates is received, updates are processed faster. Possible explanations include the just-in-time (JIT) compiler of JavaScript increasingly optimizing the code at runtime and the operating system giving the JavaScript thread an increasingly higher priority the more work it does.

It is apparent that some update rates are unfeasible in combination with larger models. In practice however, update rates exceeding 10-15 updates/s should be a rare occurrence. In certain cases, multiple participants typing text at the same time might still produce a large number of updates. This causes a high peak rate, but not a high sustained stream of updates. In conclusion, the client application is able to handle a large rate of updates (up to 15-20 updates/s) without becoming unresponsive, though the rate of updates should be much lower in practice and momentary peaks can be handled.

6.2.3. Server analysis This analysis evaluates the server's performance when handling an increasing number of concurrent sessions and updates. The following experiments were carried out on an *E2-medium* general-purpose virtual machine with 2 virtual CPU cores and 4 GB of memory on Google Cloud Platform (GCP)[18] located in Frankfurt. Apart from the underlying hardware resources, the server's runtime performance depends on some other factors such as number of sessions, number of clients per session, rate of updates clients send, and, as determined as a relevant factor by the client analysis, the complexity of each session's model.

The experiments' setup is as follows. A specific number of sessions s with a fixed length l (number of updates) is created over the course of the first session's length. TypeScript-based

[18] https://cloud.google.com/gcp

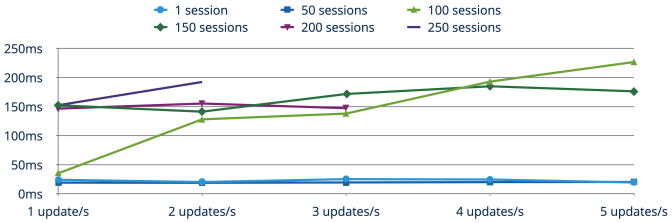

Figure 13 *1WnR* scenario. Average client latency (in ms) when connected to a server with the given number of concurrent sessions (20 participants, one of which is a writer) and rate of updates (per second).

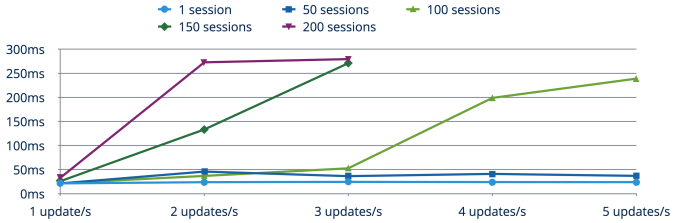

Figure 14 *nWnR* scenario. Average client latency (in ms) when connected to a server with the given number of concurrent sessions (5 writers each) and rate of updates (per second).

clients send a number of updates r each second. The first session starts immediately, the second session starts after $l \div s \div r$ seconds, and the last session starts just before the first session ends. After a session has ended, its clients disconnect and a new session is started. Thus, after the first session has ended, s sessions will be in progress, each at a different point in its lifetime. This is the point in time at which the experiments' measurements begin to be taken. This approach prevents the complexity of models to affect the results the longer an experiment runs. It also prevents potentially hundreds or thousands of clients from connecting at the exact same time, a situation unlikely in practice that could easily overwhelm the server. After the measurement phase has begun, each session reports its average latency, defined as the average time it takes for each of its clients to receive an update a writer sent to the server, when it ends. The average of s sessions' latency is recorded as the result for the experiment. The latency metric was chosen, because it has a direct impact on users as opposed to other metrics such as CPU utilization or memory pressure that are not directly perceivable.

Two different session scenarios were analyzed, one with a single writer and 19 receivers (20 participants in total), the other with 5 concurrent writers. These scenarios correspond to the cases outlined in section 6.1. Both were run with a session length of 1000 updates (200 TDG iterations in the case of 5 writers). Fig. 13 and 14 show the respective results. Ping requests to the server's IP address had a roundtrip time of 20.0ms. This can be thought of as a baseline[19]. The general trend is that the latency increases as the number of sessions and updates increases. This is most likely due to the fact that the server has a higher workload and is not able to process all incoming updates right away. The values marked with a dash could not be collected, as some clients were not able to connect to their session (due to the high number of concurrent connections).

The results show that a commonly available medium-sized server is able to support up to approximately 200 concurrent sessions, depending on the number of participants per session, the rate of updates sent per client, and the complexity of sessions' models. Clients usually experience latencies of less than 200ms or even less than 50ms when the server is not too busy (assuming that they are colocated in a similar geographical region). The server's main limitation is the amount of concurrent WebSocket connections it can handle rather than the processing power required to process incoming updates. Due to the low latencies, the collaborative features of the app are perceived as happening in real time. Consequently, both of the scenarios of our case study can be classified as feasible.

[19] though it is possible to receive WebSocket messages with a lower latency, because TCP might experience more preferential networking conditions than ICMP used by ping

6.3. Threats to Validity

The presented evaluation is subject to several internal and external threats to validity.

Internal Threats. The first limitation of this evaluation is the lack of previous studies in that area. Similar applications exist, but none of them combine mobile modeling with real-time collaboration. Due to the unexplored nature of that niche, a standard process for evaluating such systems is lacking. As a result, the means of a case study including an analysis of the solution's scalability and performance characteristics were chosen as an appropriate evaluation technique. Another limitation is the use of pseudorandom data. The TDG aims to produce updates for a model that is being edited and that is growing as a result. However, it does not reflect the way human beings would create a model. In order to produce more accurate results, a large amount of sample data (collected from the app used in practice) would need to be used either directly or indirectly by analyzing it to improve the TDG, an approach out of scope for this evaluation. Furthermore, clients and their edits had to be simulated because of the difficulty to emulate moderate to large numbers of mobile devices. These simulated clients might produce changes that differ from the changes the actual application would produce. These discrepancies, though minor, might have impacted the results. Finally, the instability of the experimentation environment needs to be mentioned as well. The experiments' results depend not only on the experiments' parameters, but also on external factors such as the available computing resources of the machine they were carried out on and the networking conditions at the time. Factors like these tend to change on a constant basis and might also have impacted the results.

External Threats. When being used in practice, the system might experience unforeseen conditions that were not covered by this evaluation. One example is inconsistent networking conditions. Mobile devices might experience excellent conditions in one area, but high latencies or poor service coverage in others. This might have a significant impact on the real-time collaboration features. Another example is the unpredictable

complexity of models created by human beings. Users might produce huge models over the course of several months that the system has to handle. Models of these proportions might cause the app or the server to exhibit poorer performance characteristics. The last example mentioned as part of this section is the unique composition of each session. A session might have two participants of which only one is editing, or thousands of participants all generating changes at the same time, or anything in between. In practice, it is impossible to predict the conditions of every possible session composition that will be experienced.

7. Conclusion

In this paper, we have shown that despite the existence of a plethora of mobile modeling applications, no application available today satisfies all of the following requirements in combination: availability, semantics, performance, usability, collaboration and offline support (cf. section 2). In particular, we identified a gap for the combination of real-time collaboration support with semantics-aware modeling despite the availability of technologies to facilitate these features.

Hence, as part of this paper, a concept for an application that fills the identified gap was developed (cf. section 4). Its main focus is support for real-time collaboration, a feature that no other semantics-aware modeling application supported. Section 5 described *CoMod*, a solution comprised of a cross-platform client application based on Flutter and a server executable based on Node.js. Its real-time collaboration capabilities are facilitated by the yjs framework and WebSocket connections. The solution was tested thoroughly and its potential for extension were explored. Subsequently, a case study was conducted to evaluate *CoMod*'s scalability and performance characteristics (cf. section 6). Our prototype and evaluation setup is available as open-source and can be found on GitHub[20].

7.1. Future Work

This proof of concept shows that it is indeed possible for a mobile modeling application to support real-time collaboration. The system could be used in practice "as is", e.g. as a tool for software development teams or computer science students' group projects. However, there is still work to be done.

The client application could be extended to support additional platforms, such as the web, allowing users working on desktop computers to take part in collaboration sessions. As mentioned in section 5.5, the app could be extended to support a wider range of models (ER, CROM, custom DSMLs, etc.). This could be expanded to the vision of a CoMod software product line comparable to (Kühn et al. 2018).

The system as a whole also bears the potential for improvement. A useful addition and must-have in an industry setting would be the implementation of a system of roles and permissions. It would allow model creators to assign roles such as *editor* or *reader* to participants and set the permissions they have accordingly.

[20] https://github.com/MaxHaertwig/CoMod

References

Brunschwig, L., Guerra, E., & de Lara, J. (2022). Modelling on mobile devices: a systematic mapping study. *Software and Systems Modeling*, *21*(1), 179–205.

Chen, P. P.-S. (1976). The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, *1*(1), 9–36. doi: 10.1145/320434.320440

Ellis, C. A., & Gibbs, S. J. (1989, 6). Concurrency control in groupware systems. In *Sigmod '89: Proceedings of the 1989 acm sigmod international conference on management of data* (p. 399-407). doi: 10.1145/67544.66963

Fette, I., & Melnikov, A. (2011, 12). *The websocket protocol* (RFC No. 6455). Internet Engineering Task Force (IETF).

Franzago, M., Ruscio, D. D., Malavolta, I., & Muccini, H. (2018). Collaborative model-driven software engineering: A classification framework and a research map. *IEEE Transactions on Software Engineering*, *44*(12), 1146-1175. doi: 10.1109/TSE.2017.2755039

Kelly, S., & Tolvanen, J.-P. (2008). *Domain-specific modeling: Enabling full code generation*. Wiley-IEEE Computer Society Press.

Kühn, T., Kassin, K. I., Cazzola, W., & Aßmann, U. (2018). Modular feature-oriented graphical editor product lines. In *Proceedings of the 22nd international systems and software product line conference - volume 1* (p. 76–86). Association for Computing Machinery. doi: 10.1145/3233027.3233034

Kühn, T., Leuhäuser, M., Götz, S., Seidl, C., & Aßmann, U. (2014, September). A metamodel family for role-based modeling and programming languages. In *Software language engineering* (p. 141-160). Switzerland: Springer, Cham. doi: https://doi.org/10.1007/978-3-319-11245-9_8

Masson, C., Corley, J., & Syriani, E. (2017). Feature model for collaborative modeling environments. In *Models (satellite events)* (pp. 164–173).

Nicolaescu, P., Jahns, K., Derntl, M., & Klamma, R. (2015, June). Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In *Icwe 2015: Engineering the web in the big data era* (p. 675-678). Springer, Cham. doi: 10.1007/978-3-319-19890-3_55

Nicolaescu, P., Jahns, K., Derntl, M., & Klamma, R. (2016, 11). Near real-time peer-to-peer shared editing on extensible data types. In *19th international conference on supporting group work* (p. 39-49). doi: 10.1145/2957276.2957310

Rothenberg, J. (1990, 4). The nature of modeling. *Artificial Intelligence, Simulation, and Modeling*, *20*(2), 48-66. Retrieved from https://www.jstor.org/stable/25061332

Shapiro, M., Preguiça, N., Baquero, C., & Zawirski, M. (2011). Conflict-free replicated data types. In *Sss 2011: Stabilization, safety, and security of distributed systems* (p. 386-400). Berlin, Heidelberg: Springer.

Vaquero-Melchor, D., Palomares, J., Guerra, E., & de Lara, J. (2017). Active domain-specific languages: Making every mobile user a modeller. In *2017 acm/ieee 20th international conference on model driven engineering languages and systems (models)* (p. 75-82). IEEE. doi: 10.1109/MODELS.2017.13

About the authors

Max Härtwig is a software engineer at Google (Switzerland) and former Master's student of Technische Universität Dresden. You can contact him at mail@maxhaertwig.com or visit https://maxhaertwig.com/.

Sebastian Götz is a tenured senior researcher in the Software Technology Group of Technische Universität Dresden. His research focuses on model-driven software development, models@run.time, robotic software engineering, and energy informatics. You can contact him at sebastian.goetz1@tu-dresden.de or visit https://st.inf.tu-dresden.de/sgoetz.