Feature Model for Collaborative Modeling Environments

Constantin Masson
University of Montreal
Montreal, QC, Canada
Email: constantin.masson@umontreal.ca

Jonathan Corley
University of West Georgia
Carrollton, GA, USA
Email: jcorley@westga.edu

Eugene Syriani
University of Montreal
Montreal, QC, Canada
Email: syriani@iro.umontreal.ca

***Abstract*—There has been a growing trend toward collaborative environments especially those utilizing browser-based interfaces which can be seen in modeling tools such as AToMPM and WebGME. In response to the growing interest in collaborative modeling, we explored existing systems and potential solutions to identify the various features relevant to collaborative modeling systems. In this paper, we detail the feature diagram resulting from our exploration of collaborative modeling systems. We also detail the features of an existing collaborative system both to illustrate the use of the diagram and further explore the features. Through this feature diagram we identify key areas for collaborative modeling systems. We hope the feature diagram will be used to guide development, analysis, and discussion around collaborative modeling systems.**

I. Introduction

There has been a growing trend toward collaborative environments especially those utilizing browser-based interfaces. Basic tools include Google Docs[1], Trello[2], and Asana[3]. Additionally, this trend can be seen in modeling tools such as AToMPM [1] and WebGME [2]. These tools bring together developers, including geographically distributed teams, in a collaborative modeling environment to work on a shared set of software artifacts.

In prior work, we defined a set of requirements and challenges for a collaborative modeling environment, enumerating four possible collaboration scenarios [3]. We also investigated an efficient and distributed architecture to support collaborative modeling as a service [4]. Other researchers and developers have also been working in this space: the (aforementioned) WebGME project has provided a browser-based interface to the GME modeling environment; Basciani et al. proposed MDEForge [5], a web-based modeling platform including collaborative features; Obeo has introduced collaborative modeling features [6]; and MetaEdit+ supports offline collaborative modeling through version control [7]. In response to the growing interest in collaborative modeling, we have explored existing systems and potential solutions to identify the various features relevant to collaborative modeling systems.

In this paper, we detail the feature model resulting from our exploration of collaborative modeling systems. The model highlights key features and interdependencies. These features are to be interpreted as key challenges for collaborative modeling environments, such as collaboration scenarios, conflict management, or multi-user support. The features shall be used to guide development, analysis, and discussion around collaborative modeling systems. We also discuss how some existing tools already implement certain features, and identify where future research can focus.

In the remainder of the paper, we give a brief overview of existing work in the area of collaborative modeling in Sect. II. In Sect. III, we present a feature modeling covering primary concerns in collaborative modeling systems. In Sect. IV, we instantiate the feature model on four tools as a form of validation. In Sect. V, we further discuss the various features and significant issues for collaborative modeling systems. Finally, we provide our concluding remarks in Sect. VI.

[1]https://docs.google.com
[2]http://www.trello.com
[3]https://www.asana.com/

II. Background and Related Work

Before discussing the feature diagram that is the focus of this paper, this section will overview collaborative modeling systems and meta-studies in the area.

A. *Collaborative Modeling Systems*

AToMPM was one of the first web-based collaboration tool for MDE [1]. It takes advantage of the ever increasing capabilities of web technologies to provide a purely in-browser interface for multi-paradigm modeling activities. The AToMPM project is also working to provide an API for collaborative modeling services along with the in-browser interface [4]. Clooca [8] is also a web-based modeling environment that lets the user create DSLs and code generators. However, it does not offer any collaboration support.

Recently, Maroti et al. proposed WebGME [2], a web-based collaborative modeling version of GME[4]. It offers a collaboration where each user can share the same model and work on it. In contrast with the Modelverse, WebGME relies on a branching scheme (similar to the GIT version control system) to manage the actions of different users on the same model. WebGME supports the multi-user single-view and multi-view single-model scenario [3]s, but not in an always-online environment, unlike in AToMPM where operations immediately succeed or fail.

[4]WebGME is now available at https://webgme.org/

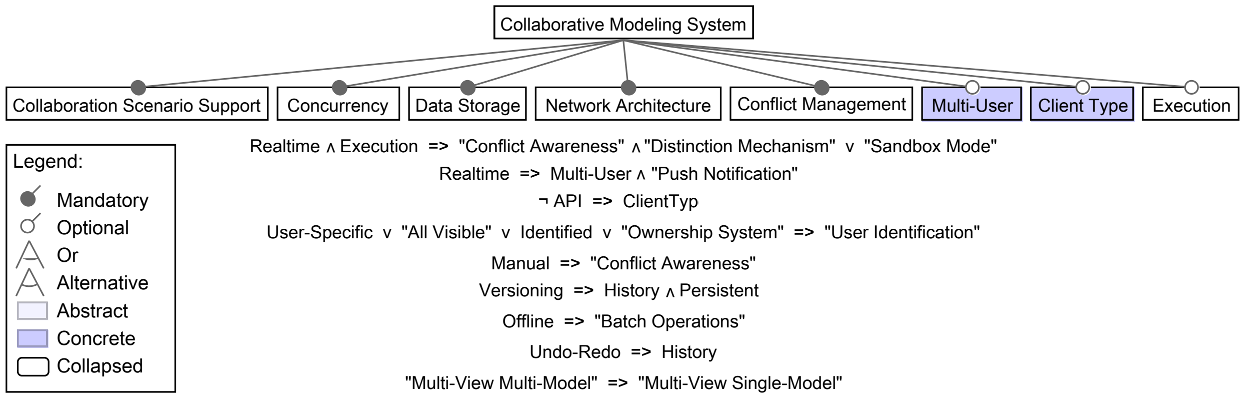
Collaborative Modeling System
Collaboration Scenario Support
Concurrency
Data Storage
Network Architecture
Conflict Management
Multi-User
Client Type
Execution
Legend:
Mandatory
Optional
Or
Alternative
Abstract
Concrete
Collapsed
Realtime ∧ Execution => "Conflict Awareness" ∧ "Distinction Mechanism" ∨ "Sandbox Mode"
Realtime => Multi-User ∧ "Push Notification"
¬ API => ClientTyp
User-Specific ∨ "All Visible" ∨ Identified ∨ "Ownership System" => "User Identification"
Manual => "Conflict Awareness"
Versioning => History ∧ Persistent
Offline => "Batch Operations"
Undo-Redo => History
"Multi-View Multi-Model" => "Multi-View Single-Model"

Fig. 1. Top-level features and constraints

MetaEdit+ has also released a collaborative modeling environment [7] of their prior desktop application [9]. The collaborative environment incorporates Git into the tool to provide for offline collaboration among modelers.

GenMyModel [10] also provides a collaborative modeling environment using an in-browser client. The environment supports many common varieties of models including BPMN, UML, Ecore, and more. Modeling projects are managed completely through the browser interface with provided facilities for managing collaboration.

Gallardo et al. proposed a three-phase framework to create a collaborative modeling tools based on Eclipse [11]. The difference between creating a regular DSL and creating a collaborative modeling tool in their system is the addition of the technological framework, which adds the collaboration support to the DSL. Naming it "TurnTakingTool", multiple users are able to modify an existing model by utilizing a turn-based system. The framework helps the user to create the DSL as a native eclipse plug-in with concurrency controls, graphical syntax and multiple user support.

Basciani et al. proposed MDEForge [5], a web-based modeling platform. MDEForge offers services to the end user by a rest api, including transformation, model, metamodel editing.

B. Collaborative Modeling Meta-Studies

At the 2016 COMMitMDE, Ruscio [12] presented the protocol and preliminary results of a systematic study of collaborative modeling systems. The study identified existing systems and introduced terminology for collaborative modeling systems. Rocco et al. [13] describe the potential and outline challenges of collaborative modeling repositories. Our work complements these works as a formal presentation of features and interdependencies identified from our exploration of collaborative modeling systems. Our work seeks to guide development, analysis, and discussion. The key contribution of our work is a feature diagram that may be used by future researchers and developers. These works further the purpose of exploring the collaborative modeling space by identifying challenges and needs of collaborative modeling spurring further research and development in the area.

C. Collaboration outside MDE

Collaboration is not limited to MDE and is found in many fields. Code editors make use of collaboration, such as Eclipse Che that uses an algorithm based on OT. File sharing systems also use concurrent algorithms to allow several users working on the same file. GoogleDoc, GoogleDrive and DropBox [14] are only some examples. The Google Realtime API used with GoogleDrive allows to concurrently modify a file [15]. Collaboration is also used in various software, such as music editors [16] and game engines, e.g., PlayCanvas Engine.

III. Features for collaborative modeling environments

Modeling environments that directly support the collaboration of many stakeholders on the same model(s) working independently are collaborative modeling environments. These environments may be offline systems utilizing features similar to a version control system to manage the shared artifacts or may allow collaborators to interact remotely in realtime. We explored a variety of existing tools and potential solutions to identify a set of features for the implementation of collaborative modeling environments. In this section, we introduce and briefly discuss each feature. Figure 1 shows the top-level feature diagram and the constraints of the model. The complete feature model is available online in ReMoDD [17].

A. Methodology

The feature model presented in this paper is the outcome of an iterative process. To identify the feature of collaborative modeling systems, we investigated all kinds of collaborative environments. Our sources include: our own software (AToMPM [3]), MDE-specific environments (OBEO [18], MDEForge [5], GenMyModel [10], CDO, ...), and other collaborative environments (GoogleDoc [15], Eclipse Che, Ohm Studio [16], Overleaf, DropBox [14], ...).

We relied on published articles related to the collaboration aspect tools when available. We also reviewed technical documentation as well as relevant blogs, tutorials, and videos that explain the technical implementation of the tools. Finally, we experienced each tool ourself when freely available. In some cases, we studied specific algorithms, in particular conflict management algorithms used by several collaborative environments, such as Operational Transformation, locking mechanisms, and the DropBox synchronization algorithm explained.

From the collected set of data, we identified which features are specific to MDE collaborative environments.

B. Collaboration Scenario Support

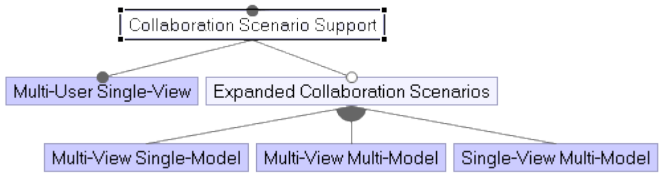

Fig. 2. Collaboration scenario features

An abstract model is the abstract syntax of a model conforming to the metamodel of a given DSL. Conceptually, a view is a projection (in whole or part) of an abstract model utilizing the most appropriate representation of a subset of the model's elements for the needs of the modeler. An environment that does not support views could be categorized as supporting only a single complete view for each model. Therefore, we can detail the possible collaboration scenarios for both tools explicitly supporting views and those not supporting views in similar terms. In collaborative modeling, we previously identified four possible collaboration scenarios [3], that we briefly describe here. In the following, we commonly refer to scenarios with only two collaborators, but recognize the scenarios can scale to an arbitrary number of collaborators.

1) Multi-User Single-View: Users are working on the same view of the model. They both see the same information in the same language and with the same concrete syntax. The changes made by a user are reflected automatically to others.

2) Multi-View Single-Model: Users work on distinct views of the same model. The views may present the same, overlapping, or disparate sets of elements in the same or different concrete syntax. Modifications on the abstract model of elements present in both views are perceived by the other user.

3) Multi-View Multi-Model: Each user is working on a different view and each view is a projection of a different model. These models have some dependency or satisfy a global constraint. Only changes on elements related between the abstract models are perceived by both users.

4) Single-View Multi-Model: This is similar to Sect. III-B3, but the dependency is defined at the view level, not at the model level, such as an aggregation of elements from both models. A user may work on a view that projects several models while another is working on one of the projected models. A change on an element used in the view is propagated to all views with the same element.

C. Concurrency

Concurrency is concerned with issues related to multiple operations occurring at the same time or in parallel.

1) Locking: When several users are working on the same model(s) concurrently, conflicting updates may occur. Simple strategies, such as retaining only the last modification, could result in losing work from one user and goes against the goals of a collaborative environment. One approach to this issue is to avoid conflicts entirely allowing users to work together transparently. However, the collaborative aspect requires that users are able to work concurrently on components even closely linked. A general overview of the file-sharing problem in the MDE environment is presented in the subversion book [19].

Pessimistic locking is a strategy widely used in concurrent systems. It is based on data locking, which only allows one user to modify the locked data. A naive but simple solution is to use a global *Data Lock*. All of the model is locked to a specific user and other users cannot access the model until the lock is released. However, this reduces collaboration, because only one user can work on the model at a time. Another solution is to use a *Fragment Lock* that applies the lock on a fragment (i.e., subset) of the model. In this solution, each user is able to modify a distinct fragment concurrently. The fragments should be as small as possible, minimizing the locked portions of the model. However, the fragment lock approach requires a well structured data format supporting well defined fragments. In the case of XML for example, we might lock the current XML tag and its children. Additionally, fragment locks do not consider dependencies that may indirectly affect the elements locked (e.g., metamodel relationships). OBEO Designer implements data and fragment locking. Another approach relies on *Dependency Locking*. This provides finer granularity that locks the element being modified and its dependencies. Though this technique is seen as only an improvement here, taking dependencies into account may be seen as required by the semantic nature of models. This is discussed in more detail in Sect. V.

Though we try to minimize the set of elements to be locked, pessimistic locking always blocks access to a set of data. This might lead to a situation where a user waits for a resource to be free. OBEO shows an overview of these locking techniques in their documentation [18]. *Optimistic Lock* tries to resolve this issue without locking. Instead, it allows the users to modify, possibly the same, elements concurrently and then merges all changes to create one unified new version of the model. For example, this is how Google Docs allows for concurrent changes and relies on the Operational Transformation (OT) algorithm [20] for merging. Unfortunately, though OT works well for text based data, model merging requires merging graphs, which makes this approach hard to apply [21].

2) Collaboration Type: We differentiate two scenarios of collaboration: *Realtime* and *Offline*. In the former, the model is modified by several users at the same time; changes are applied on the data immediately; and users are updated of changes made by others immediately. The model is the unique source of truth that all users alter concurrently. Here even the changes from a given user may not be considered complete until acknowledged by a central authority or a set of peers. On the other hand, offline collaboration presents an asynchronous

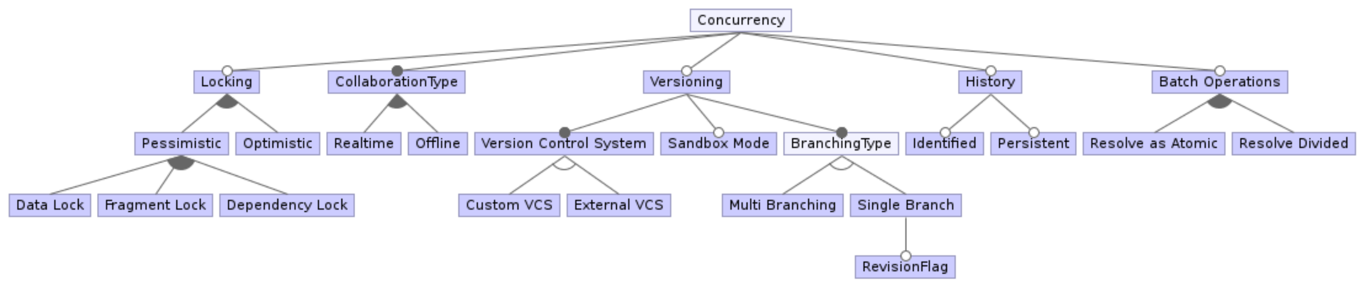

Fig. 3. Concurrency features

approach. Users may apply modifications on the model without sending the changes right away. This principle is often seen in version control systems (VCS) that allow working locally and pushing changes at a later time. This prevents immediate conflicts between users, but local versions may diverge and result in complex merging processes later, as in Git.

3) Batch Operations: All modeling systems support a set of atomic operations (e.g., creating or deleting an element) that provide the ability to develop and manage models. We recognize that some systems also allow collecting together these atomic operations in batches. The processing scheme for these batches becomes significant when discussing collaborative systems and handling potential conflicts. *Resolve as Atomic* resolves a set of operations in an all-or-nothing approach. Every operation in the batch must succeed or none of the operations can be applied. Thus, a single failing operation may result in the need to rollback changes from a set. *Resolve Divided* allows processing each operations separately. However, this may result in partial sets being applied thereby generating unexpected or even invalid results.

4) History: In a collaborative environment, storing the complex history of operations applied to a given element can be beneficial. The series of events leading to a conflict or failure may be complex, and not easily understood without a record of the operations. Here we intentionally separate history from *versioning*. For instance, a basic feature of any VCS is to manage the project history. Therefore, the presence of versioning mandates the presence of some form of history (this is represented as a constraint in the feature diagram). However, a collaborative environment might store some portion or form of history without the presence of a VCS. GenMyModel [10], for example, has a full and complex history feature, that allows replaying the history from a defined start time. History is *Persistent* if it persists when the environment halts. An example of non-persistent history is an undo / redo stack that only lasts until a given session terminates. Using a VCS implies that the history is persistent. However, it may not be enough to use the VCS history directly, but a user interface (UI) wrapper might be required. VCS integrates the set of changes from the whole project. While working on a specific model, the user does not need all of this information. Therefore, an environment might implement a history wrapper that shows only the current data models history. *Identified* history adds the user identity information to each modification. This is often the default behavior of a VCS. Allowing attributing a change to a specific user helps in diagnosing a series of events from multiple users that may have led to a conflict or failure.

5) Versioning: VCS are tools that trace all changes for a system and have facilities for managing this history. A single user working on a data model has a complete view of its state at all points and understands the full history of the model naturally. However, several users working on the same data model may lead to misunderstandings and inconsistencies. When trying to merge artefact(s), the data model(s) might be significantly altered since the last connection. Changes performed by other users may be difficult to understand. This introduces the need for versioning systems with history management. It adds the possibility to look back at the changes performed over the intervening time and to understand the full series of changes. Moreover, VCS also support other features such as documentation, reverting previous changes, or listing the added features for a release. A survey on model versioning was provided by Altmanninger et al. [22].

6) Version Control System: VCS play an important role in software development. Tools like Git and SVN are largely used and are very efficient. Therefore, collaborative modeling environments may opt to integrate an *External VCS* into the environment rather than reinventing the wheel. This places the data under the VCS management and each change is added to the history. However, these VCS use compare and merge mechanisms to integrate the changes into a new version, which require manual intervention. For instance, to integrate new changes made under Git, changes must be commited and pushed manually. Moreover, these systems are subject to conflicts, which require manual resolution. Employing locking with a version control system may avoid conflicts. Merging is then performed automatically in the background. This approach is utilized by MetaEdit+ that integrates Git with their MDE collaboration [7]. They use fine granularity locking to avoid conflict, that allows processing the merge without risk of conflicts. Furthermore, these VCS are specialized for versioning, comparing and merging text files as opposed to complex semantical data structures in MDE. For these reasons, some modeling systems may opt to build *Custom VCS*. We discuss these issues in more details in Sect. V.

7) SandboxMode: Sandbox is used to divorce a user from the typical collaborative environment. This supprts experimental or debugging processes. Working in a sandbox environment allows developing components that temporary break other components or violate general rules/expectations, without disturbing other users. The modifications may then be integrated when complete, potentially resulting in complex merges.

8) Branching Type: Multi Branching is used to divide the project into several branches. In VCS, branches are a divergent copy of the project where users may work independently from other branches. Branches are often used for new incoming feature that are not stable yet. As soon as the feature is stable and needs to be integrated, a merge with the main branch is done. This eases the team work and separates the maintenance from the new release components. However, a manual merge is required and the new feature(s) may be difficult to integrate with the main branch. New features are built on top of the old base code that may be deprecated. This issue appears when maintenance largely diverges from the main branch. Moreover, dependency ambiguities are complex to resolve, which is emphasized by the semantic nature of models.

Single Branch avoids these issues. No branches are used and every change is performed on a single version of the system. In order to separate new features, e.g., to exclude them from the release, *Revision Flags* may be used. In this way, new features depend on the up to date state of the code, while still hidden from release. One great example is Google with the Piper VCS which uses only one trunk for all its teams and flags for new features [23]. It is important to note that the use of branches is not always relative to the actual VCS branching support. Though the used VCS may support branching, users (or even environments employing an external VCS) can make the decision not to use branching. This is the case with MetaEdit+ that uses Git as a VCS but does not use Git branching feature.

D. Data Storage

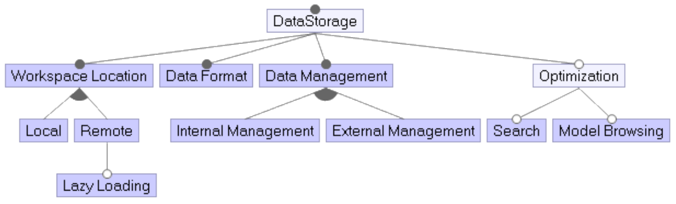

Fig. 4. Data Storage Features

Data storage is concerned with how models are stored and managed in the system to enable reuse among collaborators.

1) Workspace Location: In a collaborative environment, data is often saved in a remote place (i.e., the "cloud"). The workspace location is where the data is stored while users are modifying it. Data may be *Local*, meaning that the relevant models are on the users local machine. This may support an asynchronous workflow as with an offline collaboration type or be a response to other constraints on the system. Collaboration is often reduced to active screen sharing (single-view single-model) as supported by AToMPM. Since network latency is sometimes high, having a local copy may remove the delay between an operation being requested and being applied on the data. In contrast, *Remote* locations do not require loading the project locally. This is useful in the case of a project using a high quantity of memory. *Lazy loading* may even be used to load specific data only when required.

2) Data Format: Models use a specific data storage format. The choice of format is important and may influence or be influenced by other factors (e.g., the VCS). Some formats may be hard to use with a text-based VCS, hard to fragment for locking, or mandated to be compatible with a distributed database utilized by the system. Popular formats are JSON (for web-based modeling environments), XMI (for Eclipse-based tools), or NoSQL database formats when scale is an issue.

3) Data Management: Data management refers to managing the basic operations and long-term storage in the system (e.g., CRUD operations). *Internal Management* implies data is processed by tools internal to the software, whereas *External Management* reuses existing tools like a distributed database. Namespaces (e.g., URI) are crucial to access a model.

4) Format Optimization: Format Optimization identifies the way a system might optimize the model representation/storage for certain actions; e.g., *Model Browsing* or model *Search*. Basciani et al. [5] overview different supported query mechanisms for several systems according to the managed artifacts.

E. Network Architecture

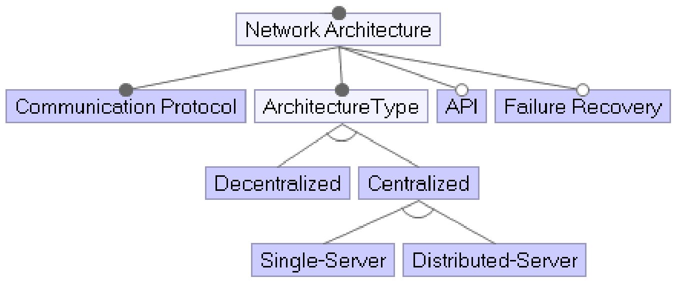

Fig. 5. Network Architecture Features

Collaboration requires the use of the network so that instances of the modeling system on different machines communicate and exchange data.

1) Communication Protocol: Clients and servers must be able to communicate using a common protocol. TCP/IP is a widely used network protocol. It maybe relevant to investigate reliable networks to guarantee no data loss, but at the cost of time performance. Client and server must then use the same data format to communicate information.

2) Architecture Type: This is the network architecture chosen for the collaborative environment. We distinguish two fundamental types of architectures: *Centralized* and *Decentralized*. *Centralized* uses a central authority. It has the advantage of having one source of truth: the centralized storage is the true version of the work. Alternatively, *Decentralized* systems distribute authority. An example of *Decentralized* architecture is Git, thought often used in a *Centralized* way (e.g., Using Github server as the main storage location).

Centralized architecture may be *Single-Server* or *Distributed-Server*. This is an internal detail, since end-users see the cluster as only one single-authority. *Distributed-Server* adds complexity to handle data synchronization across all storage locations. Nevertheless, it adds a layer of security (a single server crash will not affect the whole system) and performance (distributing processes and storage across a large set of nodes). This is useful for servers with high traffic demand. An industrial example of this is the Google Piper VCS [23], which is duplicated into 10 servers across the world using the Paxos algorithm [24]. This divides the number of request performed on each unique server and speeds up response time.

3) API: An API, though optional, may be integrated to allow extension of the system and other client implementations. For example, we are working toward an API for collaborative modeling services to allow building and integrating potentially many distinct clients [4]. Each client may support their own needs and rely upon the common modeling services provided by the API to simplify system design and interoperability. APIs are the basis for modeling as a service systems, such as MDEForge. Additionally, an API may be provided purely for internal use to manage operations between layers or nodes in the architecture. For example, there is a simplified API between the MVC and MvK within the architecture described in our prior work [4].

4) Failure Recovery: Hardware and network systems are subject to failure and error, but user experience should not be affected by a technical issue within the system. The possibility of recovery is closely relative to the chosen *Network Architecture*. Decentralized architectures mitigate this issue as each user owns a state of the project. Though the error might be disconnected from many users, consistency schemes must exist to manage the decentralized authority.

Centralized architectures are notably subject to failure in the case of *Single-Server* where there is a single point of failure necessary to take the entire system down, and recovery can be difficult to impossible depending on the severity of the failure. On the other hand, *Distributed-Server* may implement a failure recovery system. If one server crashes, the system may use another server instead to maintain availability and redundancy may be used to prevent the loss of data.

F. Conflict Management

Divergent modifications on the same data model may lead to a conflict when trying to resolve all modifications to a single consistent model state. It is important to detect conflicts and act upon resolving them.

1) Conflict Resolution: Conflict resolution is the process by which all modifications are combined in order to create a new version of the model. *Automatic* conflict resolution is the ideal solution, where conflicts are resolved automatically.

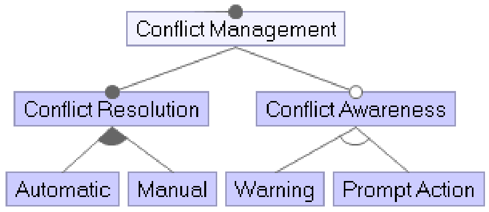

Fig. 6. Conflict Management Features

Other features, such as collaboration type and locking, strongly impact the complexity of managing automatic resolution of conflicts. A live collaboration environment with fine grained locking may be conflict free. High risks of conflict appear when using offline collaboration and versioning. Each user works separately on a divergent version of the work. To share his change with others, the user needs to perform a merge action. This could be accomplished through a diff / merge algorithm. In case of unambiguous changes, this action may be transparent to the user. For example, if each user changed different parts of the model(s) without cross dependencies. However, often *Manual* conflict resolution may be needed. This is the case of WebGME tools that refuse a push request if the server has already been modified [2]. The user must then first pull the latest changes and manually merge them with his own changes before pushing them to the server.

2) Conflict Awareness: When working in collaboration, users should be aware of other users changes. *Conflict Awareness* is the mechanism that warn users of conflicts. Notification may be sent in response to conflict (e.g., 2 users move the same element) or to warn users about potential conflict (e.g., elements being edited by other users). It is disruptive for a user to perform a modification that results in unexpected behavior, because the user was not notified of a conflict.

Warning conflict awareness are only mechanisms to inform user about conflicts and concurrent actions. This feature provides only information about conflicts or potential conflicts. This is often used by the GUI, which uses combinations of colors and animation to display the information. OBEO applies this solution by drawing a lock icon at the bottom of any locked element. However, this is not only restricted to GUI, this can also be applied in an API. For instance, it can be a special return value of a function in case of conflict, or an object state that changes according to it's conflict state. The case of a GUI is discussed in more details in Sect. V.

Prompt Action conflict awareness on the other hand, informs about conflict and requests the user take some action, such as in MedaEdit+. The user is prompted to update the model with the server to remove existing locks.

G. Multi-User

This feature is concerned with the fact that multiple users are interacting with the same or related models.

1) Authentication Method: Collaboration implies that several users are able to access the data models. An authentication

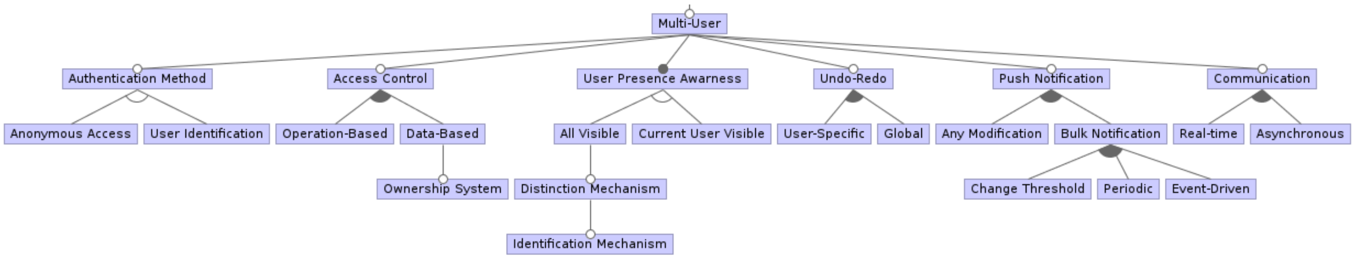

Fig. 7. Multi-User features

mechanism can be incorporated to give access only to registered users or to simply track who is responsible for a given operation. *User Identification* requires such an authentication method. The method may involve utilizing an external authentication method. For example, GenMyModel allows users to connect through their Github account. Alternatively or in addition, *Anonymous Access* may be provided to allow users to connect without having to register. This can be used for example to allow read-only access. One example outside MDE is the IRC channels that do not require registration.

2) Access Control: Users working in collaboration must be able to access the data model. However, each user may be provided a distinct set of permissions. This is handled by the *Access Control* feature. There are two distinct alternatives: *Operation-Based* and *Data-Based*. They may be mixed together in order to have more fine-grained control. *Operation-Based* restricts what a given user is able to do on any model, based on the possible set of operations (e.g., CRUD operations). A user is granted a specific operation permission that applies on any model from the project. This is similar to a database administrator that has full access to all elements. On the other hand, other users may have a restricted set of allowed operations. *Data-Based* access control may be provided to control the elements (at model or element level) and operations for those elements available to the user.

Access Control can be augmented with an *Ownership System* that adds the possibility to restrict exceptional permission to only a set of users, as in GenMyModel. A project can be shared with a team and permissions are given to certain users. They also add project visibility. A public project is visible to everyone, though read only, any user can clone the project in its own session, thereby creating a totally new project copied from this public repository.

3) User Presence Awareness: When collaborating, it is often preferred to know who is working concurrently. *User Presence Awareness* is the mechanism used to know which users are currently working on the same model. *Current User Visible* does not display any user other than the current one. This is the behavior of non collaborative software. This might be useful in collaboration in case of high number of users, to remove the information overload, but instead only show the total number of current users. To fully use the power of collaboration environment, it is recommended to use *All Visible* feature. This implies that the presence of all users working concurrently is known. One implementation would be to highlight the mouse cursor of all users (e.g., Google Docs) or highlight the graphical element actively used by each user. Nevertheless, a large number of users may cause confusion on the canvas. This introduces the use of a *Distinction Mechanism* that adds special attributes to differentiate users. It may only differentiate local user from all others (e.g., using two colors), or differentiate each user (e.g., one color per user). Using a distinction mechanism for each user can also lead to information overload. We can add additional information by using an *Identification Mechanism* to identify the operation/focus of each user distinctly. This is used by GenMyModel, which lists all current collaborators for a model.

4) Undo / Redo: Undo / Redo is a fundamental feature of professional software. The common and expected behavior of undo is to revert the last action performed with further invocations of undo reverting prior action in reverse order of original application. The standard behavior considers only a single user, providing *User-Specific* Undo / Redo. Common patterns are known for implementing this feature (i.e., Command pattern [25]). Though this pattern works well for a single-user environment, additional complexities must be handled in collaborative systems. We need to take into account not only the current user changes but also other user changes. Using a local undo stack is not sufficient: reverting a command in a collaborative environment must consider all commands. Otherwise, unexpected behaviors might appear since our previous state has been altered by others. Consider the following scenario. Alice adds the attribute `name` in an empty element. Bob adds the attribute `age`. Let us assume our undo implementation resets the element to its exact previous state. If Alice undoes her change, the element ends up being empty again. The `age` attribute has also disappeared because the implementation neglected modifications from other users. A real-time API must implement a more complex stack system for its undo / redo management that takes into account the sequence of operations and resulting interdependencies introduced by concurrent collaboration. An example Undo / Redo in a collaborative environment is discussed in detail by Cheng et al. [26]. Another way to handle Undo / Redo is to use *Global Undo / Redo*. The stack is shared across all collaborators. GenMyModel can handle both alternatives: a user undo / redo is present, while history features allow global undo / redo using the general stack of changes.

5) Push Notification: This is the mechanism by which user modifications are automatically propagated to others collaborating on the same model(s). Whenever an operation is executed, all users must be aware of it and their local data model updated. We distinguish between two kinds of *Push Notifications*. *Any Modification* is a continuous notification scheme. Every operation generates a push out to every user. This ensures the server is up to date and all users see what others are doing in real-time (with some tolerance for network latency delays). This technique is used by OBEO, GenMyModel, and WebGME.

Alternatively, *Bulk Notification* regroups changes and sends them only when specific conditions are reached. Bulk notifications may be sent manually in an *Event-Driven* approach (e.g., on a save action), in a *Periodic* approach (e.g., each 2 minutes), or in a *Change Threshold* approach. MetaEdit+ chooses the event-driven option and sends changes only when the user saves. This gives several advantages over *Any Modification* notification. Network load is notably reduced and users can hide from others the temporary broken or intermediate state of the data model. Working on models often requires moving a set of elements, temporarily destroying links, and other temporary or intermediate states. Though the goal of one's action may be acceptable, others may be disturbed by this temporary/intermediate state. Other users are only interested in the result. However, the *Sandbox* feature can be provided to handle similar scenarios.

6) Communication: Communication tools are used to contact other collaborators, ask them questions, add comments on elements, or discuss conflicts detected and merges. Communication tools are closely related to *User Presence Awareness* that allows identifying who is working concurrently, and then communication tools enable contacting them without using external tools. *Real-time* tools allow for direct communication. The users must communicate in real-time. This is the case of video call, audio call, and instantaneous chat. On the other hand, *Asynchronous* communication allows delayed communication with other users (e.g., chat box, comment, or annotations on components). A complete taxonomy of communication tools in MDE collaborative tools is provided by Davide Di Ruscio in his systematic mapping of collaborative model-driven software engineering [12].

H. Client Type

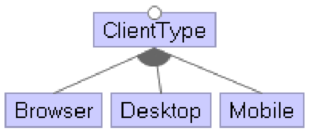

Fig. 8. Client Type Features

The end-user typically works on model(s) through a graphical representation. Modeling tools often make heavy use of mouse cursor. Therefore, *Desktop* (MetaEdit+, OBEO) and *Browser* (AToMPM, WebGME) applications are common choices. On the other hand, very few *Mobile* application are utilized for modeling(e.g., FlexiSketch [27]). *Browser* clients have an advantage in portability. Collaboration involves multiple users that often use different operating system. Supporting a cross platform desktop application introduces additional complexity and in modern systems in-browser environments seem to be preferred. In-browser environments also eliminate the need for installation and dependency management for end-users. However, a system may be provided without an explicit client type being specified, but an appropriate API must be provided to enable access to the system.

I. Execution

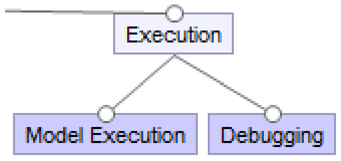

Fig. 9. Execution features

A special attention should be given to executable models in a collaborative context. Let it be model transformation or model simulation, *Executing Models* can be done locally or have an impact on the remaining users' experience. Many issues arise, such as whether the transformation should happen in batch or every step is visible to all users. The latter is supported by AToMPM. Similarly, *Debugging* executable models is often done in a different mode than when editing. This may raise conflict where the state of a model element is modified by the execution while another user is modifying it. Note that supporting execution within a realtime environment imposes additional constraints (included in Figure 1). The constraints address the need to support users working concurrently by preventing or mitigating the impact of conflicting operations on the workflow of a given user.

IV. Examples

We present four tools, among those we examined, chosen because of specific variants of the features they use and because there was enough documentation to support the claims.

A. Instantiation

An instantiation of the feature diagram will include all mandatory features as well as specifying those features with alternative sub-features or any optional features included. The following discussion assumes all mandatory features are included. In our feature diagram, the top level mandatory features are *Collaboration Scenario Support*, *Concurrency*, *Data Storage*, *Network Architecture*, and *Conflict Management*. Without one of them, a collaboration environment wouldn't be possible. On the other hand, *Mutli-User*, *Client Type*, and *Execution* are not mandatory. Though feature such as *Multi-User* are important, some cases may not need it. This is for instance the case of API for MDE collaboration without a provided client.

a) OBEO: OBEO is a collaborative modeling tool in Eclipse based on CDO with *Centralized* server and continuous integration. Concurrency in OBEO uses *Pessimistic Locking* with both *Data Lock* and *Dependency Lock*. Any modification on a model locks the whole model for this user (*Data Lock*) restricting the ability for two users to work on the same model concurrently. However, they can work on separate models, even with strong dependencies, using *Dependency Lock*. Whenever a model is locked, its dependencies are also locked. This is fine grained locking allowing users to work concurrently without conflict. OBEO uses four Easy-To-Customize representations, diagrams, tables, matrices and trees. Collaborators may use any of these representations to work on the same data model, therefore, both scenarios *Multi-User Single-View* and *Multi-View Single-Model* are supported.

b) WebGME: WebGME is an open source project that implements collaborative modeling with a *Centralized* server. Concurrency is integrated with a *Custom Versioning System* that also handles *Multi Branching*. Several users may work on different branches and then merge together, using the integrated merging tool. Each modification automatically creates a commit. *Offline modification* is supported: the working branch is automatically merged with the synchronized version at the next connection. Concurrency is resolved by an interesting *Optimistic Locking* mechanism such that no actual locks are required. In most cases, conflicts are resolved automatically using, for example, a first-win rule. However, some conflicts require manual intervention. In that case, a conflict is automatically resolved by creating a new branch for the user in conflict. His changes are then local to its branch and a manual merge to the trunk is required. Though *Concurrency* is well supported, *User Presence Awareness* is still missing and users may not understand why a conflict occurred. WebGME supports the *Single-View Multi-Users* scenario.

c) GenMyModel: GenMyModel is a browser-based modeling tool that supports collaboration. GenMyModel introduces a complex sharing system similar to that provided by GitLab and Github; i.e., *Data-Based Access Control* with an *Ownership System*. A user is the owner of their project and has all rights on it. Collaborators may then be added to the project with specific rights. An *Operation-Based Access Control* is introduced with the public or private state of a project. In public projects, any user has read-only access, whereas private projects are visible and accessible only by their owner and collaborators. GenMyModel provides *Live Collaboration*, implementing a *Centralized Network Architecture*. Data can be accessed from anywhere and is not cloned on the client side. *History* is implemented without an external VCS. All changes made by users are saved creating a modification timeline. Users can therefore browse the change and go back to a previous version. GenMyModel allows several collaborators to work on the same model with the same view. This is the first scenario *Multi-User Single-View* that is supported. Moreover, it also adds possibility to have a view that project several models, therefore *Single-View Multi-Model* scenario is also supported.

d) MetaEdit+: The collaborative infrastructure of MetaEdit+ is similar to OBEO. It uses a *Centralized* server with continuous integration and applies *Pessimistic Lock* with a fine granularity and distinguishes dependencies. Therefore, only a minimal set of elements is locked and only when needed [9]. It integrates a *Version Control System* using an *External VCS*. The VCS working directory is managed as a separate repository and uses the MetaEdit+ *API* to process the requests. In that sense, MetaEdit+ implements a full support of existing VCS to version control models. *Single-View Multi-Users* scenario is supported by MetaEdit+.

V. Discussion

A. Locking and dependencies

Locking a model may seem safe, since only one user can apply modifications on it. However, dependencies must be taken into account. The current model Alice is modifying (and locked to her) is safe from Bob's updates. But this model may have dependencies with another model that is not currently locked. If Bob modifies this other model, we would have altered Alice's model, even if it was locked. For instance, Alice can divide a UML diagram into several subsets linked together. Changes that affect another subset should spread. Therefore, when an element is modified, other affected elements must be locked as well. However, dependencies are common in models and even small fragments might end up locking a huge set of elements. Dependencies should be taken into account, but supporting transitive links may be too restrictive. Maroti et al.emphasize how simple operations, such as copy or delete, may end up locking a significant portion of a model [2].

B. Versioning for Models

Version control systems are widely used for source code projects. The syntactic nature of source code works efficiently with the text based nature of this diff & merge. The VCS detects modifications in the files and create a new version by merging both changes. EMF Compare is a good example of an MDE tool using text-based comparison for models. A recent article shows a way to use EMF Compare with EGit on Eclipse in order to integrate MDE versioning [28]. However, only diff & merge is often not the most relevant for MDE because of the semantic nature of models. This issue of integration with VCS is explained in details by MetaEdit+ [7].

C. GUI for Conflict Awareness

We have explained the importance of having a *User Conflict Awareness* mechanism. Here, we discuss concerns relevant to GUIs. The following examples emphasize the importance of *User Conflict Awareness* in a GUI. Assume that Alice drags and drops an element from a model. Bob tries to drag and drop the same element at the exact same time, but to a different location. Alice releases the element before Bob. If the conflict management system is optimistic, the last to drop is retained. Therefore, Bob's final action prevails, and the element is placed according to his action. From Alice's point of view, the element disappears or shifts when she releases it. This is confusing and unexpected behavior. Alice may even see this as a bug. In such situations, a GUI warning may be displayed. One solution is to blink the element on Alice's GUI and play an animation that moves the element to the final position (from Bob's action). This way, Alice understands that her change was immediately followed by Bob's. OBEO applies this solution by annotating with a lock icon at the bottom of any locked element. Google Drive, places a special red cross icon on a deleted element. The cross lasts long enough for the user to see the element being deleted before it disappears.

D. Why should we use User Presence Awareness

Knowing who is currently working on the same document has several advantages. For a communication purpose, this warns us of others currently working on the same project. This eases the general knowledge of the team work and who to contact if an element needs to be discussed. For instance, if a fast feed-back is required for a modification, it is easy to request others. Moreover, it has an impact on the learning process: users are able to help each other. If Bob makes use of an action unknown to Alice, asking him is easy and fast via the communication mechanism in place.

E. Note about Push Notification

We discussed in the *Push Notification* feature that user modifications are sent either continuously (*Any Modification*), or regrouped and sent manually or periodically (*Bulk Notification*). In practice, it is often a mix of both. For example, in GoogleDoc, modifications are sent at upon saving the document, which is triggered automatically. The frequency of saves usually starts after a significant change is made in the document. This result in an almost continuous *Push Notification* and releases the network load at the same time.

VI. Conclusion

Following our prior work and the growing trend toward collaborative modeling systems, we have outlined a feature model identifying key concerns for collaborative modeling systems. To develop the feature diagram, we explored existing collaborative modeling systems and potential solutions. We hope that it will encourage and facilitate development, discussion, and analysis on collaborative modeling systems.

As future work, we would like to systematically assess the benefits of each feature. This will help us choose the features and variants to support in our tool AToMPM.

References

[1] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin, "AToMPM: A Web-based Modeling Environment," in *Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition*, ser. MODELS'13, vol. 1115. CEUR-WS.org, 2013, pp. 21–25.

[2] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, and Á. Lédeczi, "Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure," in *Multi-Paradigm Modeling*, vol. 1237. CEUR-WS.org, oct 2014, pp. 41–60.

[3] J. Corley, E. Syriani, H. Ergin, and S. Van Mierlo, "Cloud-based Multi-View Modeling Environments," in *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, A. M. Cruz and P. S, Eds. IGI Global, 2016.

[4] J. Corley, E. Syriani, and H. Ergin, "Evaluating the cloud architecture of atompm." in *MODELSWARD*, 2016, pp. 339–346.

[5] F. Basciani, J. Di Rocco, D. Di Ruscio, A. Di Salle, L. Iovino, and A. Pierantonio, "MDEForge: an extensible Web-based modeling platform," *CloudMDE*, p. 66, 2014.

[6] "Obeo collaborative modeling documentation," http://docs.obeonetwork.com/obeodesigner/8.0/user/CollaborativeModeling_User.html, 2017.

[7] S. Kelly, "Smart model versioning," http://modeling-languages.com/smart-model-versioning/, 2017.

[8] S. Hiya, K. Hisazumi, A. Fukuda, and T. Nakanishi, "clooca : Web based tool for Domain Specific Modeling," in *Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition @ MODELS'13*, vol. 1115. CEUR-WS.org, 2013, pp. 31–35.

[9] S. Kelly, K. Lyytinen, and M. Rossi, "MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment," in *Conference on Advanced Information Systems Engineering*, ser. LNCS, vol. 1080. Springer, 1996, pp. 1–21.

[10] M. Dirix, blog.genmymodel.com/discover-the-revision-history-in-your-genmymodel-projects.html, 2017.

[11] J. Gallardo, C. Bravo, and M. A. Redondo, "A model-driven development method for collaborative modeling tools," *Journal of Network and Computer Applications*, vol. 35, no. 3, pp. 1086–1105, 2012.

[12] Davide Di Ruscio, "Collaborative model driven software engineering: a Systematic Mapping Study," *COMMitMDE at MoDELS 2016*, 2016.

[13] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio, "Collaborative Repositories in Model-Driven Engineering," *IEEE Software*, 2015.

[14] Guido Van Rossum, "How the Dropbox Datastore API Handles Conficts," https://blogs.dropbox.com/developers/2013/08/how-the-dropbox-datastore-api-handles-conflicts-part-two-resolving-collisions/, 2013.

[15] Google, "Google realtime api," https://developers.google.com/google-apps/realtime/overview, 2017.

[16] Irisate, "Real time collaboration technology roundup," https://irisate.com/collaborative-editing-solutions-round-up/, 2017.

[17] E. S. Constantin Masson, Jonathan Corley, "Collaborative modeling environment features," http://www.remodd.org/v1/content/collaborative-modeling-environment-features, Universit de Montral, University of West Georgia, 2017.

[18] OBEO, "Collaborative features," https://www.obeodesigner.com/en/collaborative-features, 2017.

[19] "Subversion book," https://tortoisesvn.net/docs/release/TortoiseSVN_en/tsvn-basics-versioning.html, 2017.

[20] D. Spiewak, "Understanding and applying operational transformation," http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation, 2010.

[21] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer, "A fundamental approach to model versioning based on graph modifications: from theory to implementation," *Software & Systems Modeling*, vol. 13, no. 1, pp. 239–272, 2014.

[22] K. Altmanninger, M. Seidl, and M. Wimmer, "A survey on model versioning approaches," *International Journal of Web Information Systems*, vol. 5, no. 3, pp. 271–304, 2009.

[23] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Communications of the ACM*, 2016.

[24] Leslie Lamport, "Paxos Made Simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 51–58, 2001.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[26] Y. Cheng, F. He, S. Jing, and Z. Huang, "An multiuser undo/redo method for replicated collaborative modeling systems," *Computer Supported Cooperative Work in Design, 2009. CSCWD 2009 13th International Conference on Computer Supported Cooperative Work in Design*, 2009.

[27] D. Wüest, N. Seyff, and M. Glinz, "FlexiSketch: A Mobile Sketching Tool for Software Modeling," in *Mobile Computing, Applications, and Services*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, Berlin, Heidelberg, 2012, pp. 225–244.

[28] Philip Langer, "Version control for models: From Research to Industry and Back Again," in *Workshop on Models and Evolution*, vol. 1706. CEUR-WS.org, 2016, p. 1.