Modern Software Engineering Methodologies for Mobile and Cloud Environments

António Miguel Rosado da Cruz
Instituto Politécnico de Viana do Castelo, Portugal

Sara Paiva
Instituto Politécnico de Viana do Castelo, Portugal

A volume in the Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) Book Series

Information Science REFERENCE
An Imprint of IGI Global

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA, USA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Names: Cruz, Antonio Miguel Rosado da, 1970- editor. | Paiva, Sara, 1979- editor.
Title: Modern software engineering methodologies for mobile and cloud environments / Antonio Miguel Rosado da Cruz and Sara Paiva, editors.
Description: Hershey, PA : Information Science Reference, 2016. | Includes bibliographical references and index.
Identifiers: LCCN 2015046896| ISBN 9781466699168 (hardcover) | ISBN 9781466699175 (ebook)
Subjects: LCSH: Cloud computing. | Mobile computing. | Software engineering.
Classification: LCC QA76.585 .M645 2106 | DDC 004.67/82--dc23 LC record available at http://lccn.loc.gov/2015046896

This book is published in the IGI Global book series Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) (ISSN: 2327-3453; eISSN: 2327-3461)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 7
Cloud-Based Multi-View Modeling Environments

Jonathan Corley
University of Alabama, USA

Huseyin Ergin
University of Alabama, USA

Eugene Syriani
University of Montreal, Canada

Simon Van Mierlo
University of Antwerp, Belgium

ABSTRACT

Complex systems typically involve many stakeholder groups working in a coordinated manner on different aspects of a system. In Model-Driven Engineering (MDE), stakeholders work on models in order to design, transform, simulate, and analyze the system. Therefore, there is a growing need for collaborative platforms for modelers to work together. A cloud-based system allows them to concurrently work together. This chapter presents the challenges for building such environments. It also presents the architecture of a cloud-based multi-view modeling environment based on AToMPM.

INTRODUCTION

Complex systems engineering typically involves many stakeholder groups working in a coordinated manner on different aspects of a system. Each aspect addresses a specific set of system concerns and is associated with a domain space consisting of problem or solution concepts described using specialized terminology. Therefore engineers express their models in different domain-specific languages (DSL) to work with abstractions expressed in domain-specific terms (Combemale, Deantoni, Baudry, France, Jézéquel, & Gray, 2014).

Model-Driven Engineering (MDE) (Stahl, Voelter, & Czarnecki, 2006) is considered a well-established software development approach that uses abstraction to bridge the gap between the problem domain and the software implementation. MDE uses models to specify, simulate, test, verify, and generate code for applications. A model represents an abstraction of a real system, capturing some of its essential properties, to reduce accidental complexity present in the technical space. A model conforms to a metamodel (Kühne, 2006), which defines the abstract syntax of a DSL. The metamodel specifies the permissible concepts, relations and properties that models conforming to the metamodel can have. Models are rep-

DOI: 10.4018/978-1-4666-9916-8.ch007

resented with a concrete syntax (graphical or textual) which defines the notations used to represent each model element. MDE activities typically include the development of modeling languages, the design of models, the implementation of model transformations, the run-time execution of models, and the analysis of models. Several modeling tools exist today, such as AToMPM (Syriani, Vangheluwe, Mannadiar, Hansen, Van Mierlo, & Ergin, 2013), EMF (Steinberg, Budinsky, Paternostro, & Merks, 2008), GME (Ledeczi, et al., 2001), and MetaEdit+ (Kelly, Lyytinen, & Rossi, 1996).

Recently, there has been a growing trend toward collaborative environments especially those utilizing browser-based interfaces. Common tools include Google Docs (Google Inc., 2015), Trello (Trello Inc., 2015), Asana (Asana, 2015), and more. Additionally, this trend can be seen in software development tools including WebGME, a web-based collaborative modeling version of GME (Maróti, et al., 2014) and Eclipse webIDE (The Eclipse Foundation, 2015). These tools bring together developers, including geographically distributed teams, in a collaborative development environment to work on a shared set of software artifacts. However, the introduction of these collaborative environments bring new concerns.

In this chapter, we address the need to provide a collaborative environment for domain-specific modeling. Furthermore, in order to ensure consistency and synchronization among the artifacts produced by each stakeholder, we favor a cloud-based environment. Although there is a growing need for such environments, few modeling tools allow multiple stakeholders to work on their modeled system concurrently. In this chapter, we first define the requirement and challenges for a collaborative modeling environment where we enumerate the possible collaboration scenarios. We then present our tool AToMPM, which was designed for collaborative modeling in the cloud, and describe the detailed architecture of how AToMPM solves the challenges of the collaboration scenarios. Additionally, we present competing approaches to ensure consistency and synchronization among shared artifacts with discussion of how the competing approaches differ from the approach applied in AToMPM.

BACKGROUND

Online collaboration tools are very popular with the rise of new HTML5 technologies. Offerings such as Google Docs (Google Inc., 2015), Trello (Trello Inc., 2015), Asana (Asana, 2015) and many others take advantage of sophisticated features of new web technologies. They enable users to accomplish tasks without the need for a native client. Modern modeling tools are primarily native desktop applications, e.g., EMF (Steinberg, Budinsky, Paternostro, & Merks, 2008) and MetaEdit+ (Kelly, Lyytinen, & Rossi, 1996). AToMPM is the first web-based collaboration tool for model-driven engineering. It takes advantage of the ever increasing capabilities of web technologies to provide a purely in-browser interface for multi-paradigm modeling activities. Nevertheless, Clooca (Hiya, Hisazumi, Fukuda, & Nakanishi, 2013) is a web-based modeling environment that lets the user creates domain-specific modeling languages and code generators. However, it does not provide any collaboration support.

Recently, Maroti et al. proposed WebGME, a web-based collaborative modeling version of GME (Maróti, et al., 2014). WebGME offers a collaboration where each user can share the same model and work on it. It has a Mongo database server as a backend for model repository. In contrast to the modelverse, which is a specialized modeling-optimized repository, WebGME uses a simple branching scheme with a generic document-based NoSQL database to manage the actions of different users on the same

model. A user may request a branch update after manipulating the model. If other users do not submit branch update requests or do not modify the model in any submitted branch updates (i.e., the model is not changed), then the branch update of the current user is applied and the changes are broadcasted to other users of the same model. When a branch update fails (meaning the model has already been modified in the database) the client can either 1) reject the local change and present the new changes in the server to the user, 2) automatically create another branch from the changes made in the local model, or 3) perform a merge of the local changes with the changes in the server. After completing one of the three options, the user retries a branch update. This simple versioning functions similar to the GIT version control system (GitHub, 2015). Using this version control system, WebGME makes the user work with local copies of models and updates the models in the database only after the user requested a branch update. Even though, it supports multi-user single-view and multi-view single-model scenarios, basing a real-time update on user requests prevents WebGME from having a pure collaborative environment as in AToMPM. The MVC structure from AToMPM provides a view-based system, where each user subscribes to changes in that specific view. Therefore, the user does not need to get all updates of the model from the server and only the elements represented in that view. Furthermore, MVC is a live solution where conflicting updates scenarios are limited and operations will either immediately succeed or fail as appropriate.

The GEMOC Initiative (Combemale, Deantoni, Baudry, France, Jézéquel, & Gray, 2014) has produced GEMOC Studio (GEMOC Initiative, 2015), a modeling system handles a multi-view multi-model scenario. The intent of GEMOC Studio is to provide facilities for DSL developers to define composite DSLs combining multiple DSLs relevant to a project into a cohesive system with well-defined interaction of the various models (Combemale, et al., 2013). GEMOC Studio is contained within the Eclipse ecosystem and therefore a single-user desktop application. The solution provided for the multi-view multi-model is similar to the solution provided by MVC, but does not have the added complexity of handling multiple concurrent users. GEMOC Studio provides an alternative solution for the single-view multi-model scenario, by defining interactions of heterogeneous models. It requires the user to define the relations between models in advance. Thus, the solution is also the refactored single-view multi-model. The authors are not aware of a solution which handles the complexity of representing diverse models with arbitrary intent and notation within a single view that does not require user intervention to define the intersection of the models. The problem is likely unsolvable without some refinement of the domain and relations of the various models.

Eclipse webIDE (The Eclipse Foundation, 2015) is a new project aspiring to provide an in-browser interface for the Eclipse development environment. The project would, thus, present another option for in-browser modeling activities through the available Eclipse modeling tools (e.g., EMF and Epsilon). At the time of writing, the project is still under development, and therefore further discussion of the multi-view modeling capabilities is not possible.

The distributed databases community has also dealt with some of the issues of collaborative environments (Özsu & Valduriez, 2011). However, the primitives of modeling and distributed databases are not the same. Databases are concerned with a restricted set of low level data representations, and collaborative modeling systems utilize higher level abstractions that commonly include domain-specific concerns. However, similar to our view system, distributed databases must handle multiple database instances as external views, either co-located or located separately. Furthermore, distributed databases consider multiple users accessing concurrently, but the database systems lack the complexity of modeling systems.

REQUIREMENTS FOR A COLLABORATIVE MODELING ENVIRONMENT

In the following we first discuss the various collaboration situations that may occur in a collaborative development environment and then focus on how these apply to a domain-specific modeling environment.

General Collaboration Situations

In practice, teams of stakeholders work together in order to produce a coherent and complete system. For example, in the design of an automotive system, stakeholders are partitioned into teams based on their expertise: electrical engineers, mechanical engineers, control engineers, ergonomists, parts assemblers, etc. There are typically three situations when individuals within a team or across teams collaborate in order to produce the overall system:

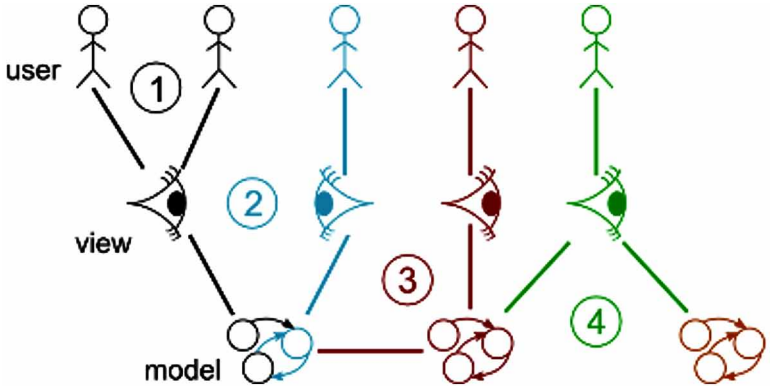
1. *Stakeholders are working on the exact same artifact.* This is equivalent to having both of them share the same screen. In this case, all changes made by one stakeholder are directly reflected and perceived by the other stakeholders, such as in Google Docs. This situation is useful when, for example, two stakeholders are manually inspecting a model together, if one stakeholder is training the other, or in a development process favoring pair development (like in extreme programming) (Beck, 1999).
2. *Stakeholders are working on different parts of the same artifact.* This situation is useful when artifacts are designed incrementally. This is possible when the language, in which the artifact is described, offers a modularity mechanism that allows one to split its instances into different parts, such as partial classes in C# (Corporation, 2001) and aliases in UML diagrams (Object Management Group, 2012).
3. *Stakeholders with differing expertise are working on distinct artifacts that, together, compose the overall system.* In this case, each artifact represents a concern of the overall system, e.g., the electrical, software, and the security concerns of an automotive. This is useful when a system is designed by separating its concerns, such as in aspect-oriented programming (Kiczales & Hilsdale, 2001).

Collaboration Scenarios in Multi-View Modeling

Modeling tools, frameworks, and language workbenches, such as AToMPM, EMF, and GME, typically consider all developed artifacts as models. In the context of a collaborative effort among individuals, such tools must separate views from models. A view is a projection of the model, showing only a part of the model in its own concrete syntax representation. Models are stored in a cloud-based repository and can only be accessed via their views. Therefore, it is necessary to have at least one view for a model to exist.

In order for tools to support collaborative modeling activities, we refine the previous collaboration situations into four possible collaboration scenarios for multi-view modeling. These are illustrated in Figure 1. For simplicity, we restrict our discussion to two users/views/models, although generalizable to an arbitrary number of each component. For each scenario, we briefly discuss what conflicts can occur when users are connected live to the cloud and an idea on how to resolve them. Note that, in practice, any combination of these scenarios is possible, but we will treat each one separately.

Figure 1. Scenarios in multi-view modeling



Multi-User Single-View: (① in Figure 1) Two users are working on the same view of the model. They both see the exact same data in the same concrete syntax. Changes made by one user are reflected automatically to the other. In the case of simultaneous conflicting changes (e.g., one user is deleting an element that the other user is updating), the conflict is resolved in a first come, first served fashion.

Multi-View Single-Model: (② in Figure 1) Two users are each working on a different view of the model. The two views may be presenting the same elements in the same or different concrete syntax. They may also share only some or no elements between them. In any case, the views represent a part of the same model and therefore the models in the views conform to the same modeling language. In the non-intersecting views cases, changes in one view are not reflected in the other view. Otherwise, if an element is present in both views, changes in its abstract syntax (e.g., value of the DC voltage in the electrical diagram model) in one view are reflected in the other view. However, changes that only affect the concrete syntax of the element (e.g., length of a wire) are not reflected in the other view. Note that some changes in the abstract syntax of an element may change its concrete representation (e.g., turning a switch on). In the case of simultaneous conflicting changes on a shared element, a conflict management strategy must be put in place to solve the conflict and maintain the two views consistent with each other.

Multi-View Multi-Model: (③ in Figure 1) Two users are each working on a different view and each view is a projection of a different model. The models define together the overall system. This means that elements in each view are part of models conforming to different modeling languages. However, the two models are subject to global consistency constraints (e.g., a method call to an object in a UML sequence diagram can only be established if the UML class diagram has defined that method on the target class of the object). Furthermore, an element in one model may explicitly rely on elements in the other model, by having a reference to it. Only if that reference is part of the view of the second model, then changes to the referenced element in the first model may be reflected in that view. In particular, if the referenced element is deleted in the original model, then a decision must be made for whether the deletion should cascade to the reference or not. Simultaneous conflicting changes to that element are not existent between the two views, since one view passively references the element.

Single-View Multi-Model: (④ in Figure 1) This is a particular case of the previous scenario where one user is working on a view that projects two models, while another user is working on a view of one of these models. In this case, it is the view that makes the link between elements of the two models as

opposed to the previous case where that link is defined in the model. Typically, the view represents an abstraction of elements of the two models (e.g., a correspondence relation between the resistors of the electrical circuit and the heat sensor of the engine). This view is generally the one that defines the consistency relation between elements of the models. A change that occurs on the abstract syntax of an element in one of the models may therefore affect the view. This scenario is outside the scope of this chapter.

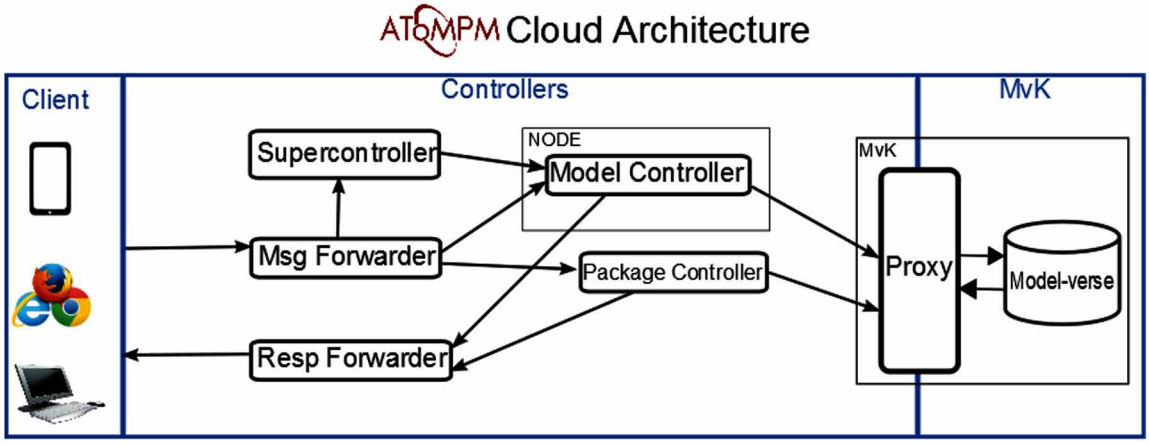
MULTI-VIEW, MULTI-USER MODELING IN ATOMPM

In this section, we describe how we enhanced the architecture of the modeling environment AToMPM to support multiple users and multiple views to collaboratively perform MDE activities on the cloud. Figure 2 illustrates the different components of this architecture. There are three primary components: client systems, Modelverse Kernel (MvK), and a set of controllers coordinating the other two components. The design is similar to the well-known Model-View-Controller pattern, in which the client is the *view*, the MvK is the *model*, and the controllers are the *controller*. The system is designed to be client agnostic and therefore supports a wide variety of clients. The MvK handles processing all modeling actions, such as primitive requests and execution, directly on models that are stored in a repository. The controllers ensure consistency among clients, views, and models. We therefore denote this latter part of the architecture MVC.

Modelverse Kernel

The Modelverse (Van Mierlo, Barroca, Vangheluwe, Syriani, & Kühne, 2014) is a repository, or database, of models. The Modelverse stores any modeling artefact, including, but not limited to, metamodels, concrete syntax, models, operations, and rule-based model transformations. It is accessible through an interface, consisting of basic CRUD operations. As the Modelverse is unaware of the (semantically)

Figure 2. Overview of Model-View Controllers Architecture



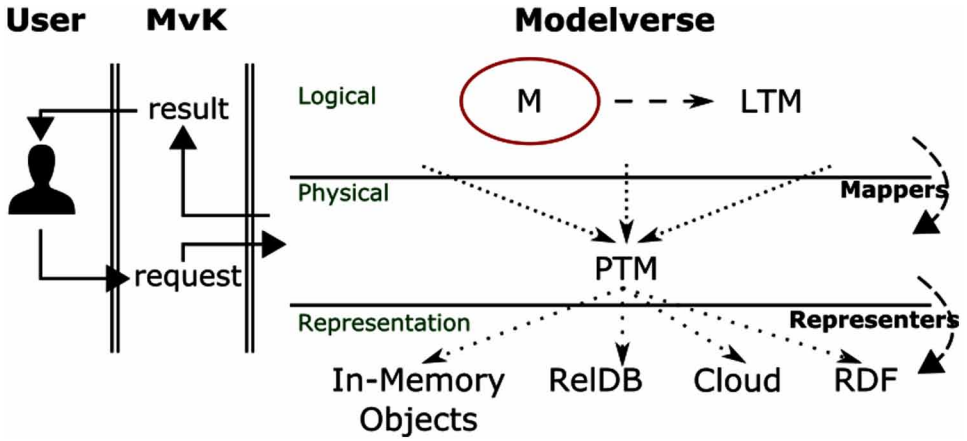
allowed operations, a kernel is necessary to coordinate all operations, and ensure consistency and conformance. This kernel is called the Modelverse Kernel (MvK).

The MvK exposes a public interface consisting of model management operations that allow users to interact with their models while ensuring a consistent data store. It consists of the following operations:

- **Create, Read, Update, Delete** (**CRUD**)**:** Operations allow the user to manipulate and query logical elements.
- **Conformance Checks:** Allow to check whether one model conforms to another model. In particular, a user can check whether a model conforms to a metamodel. This metamodel defines a set of domain concepts (*classes* with *attributes*) that can be connected with each other (*associations*). It can optionally specify a minimum and maximum bound on these connections, called *multiplicities*. Furthermore, arbitrary constraints can be expressed in a constraint language. The check ensures that the model does not violate any constraint, i.e., it instantiates the correct types, and correctly combines them.
- **Executing:** Models, in particular models of computation. The Modelverse has a built-in, explicitly modeled (minimal) action language whose models can be executed. This allows the user to define operations and execute them.

Figure 3. shows the architecture of the Modelverse as a standalone model repository which is accessible through the MvK. In the figure, the central entity is a model *M*. It conforms linguistically to a linguistic type model, or metamodel, *LTM*. In the physical dimension, one type model is defined. It defines the concepts the Modelverse needs to know about in order to function: classes, attributes, associations, primitive data types, action language, and so forth. It acts both as a type model (to which all models in the Modelverse conform), and an interface definition for the implementation, which defines the representation on a physical medium, of those structures. Although the Modelverse can be seen as a database of models, the representation of those models on physical media, such as a relational database or in-memory objects, is not known to the user. This knowledge is not necessary because of the uniform

Figure 3. Modelverse



access through the MvK, as model management operations are performed on instances of the physical type model. A user only interacts with models on this conceptual level and will never need to query the actual physical representation. The representation of physical type model elements onto physical media is catered for by *representers*, one for each physical medium. For example, the default representer maps physical type model elements onto in-memory Python objects. Alternative representers would do the same for relational databases, RDF triple stores, and others. With this level of indirection, we make sure that this representation only has to be defined once: we know how to represent physical type model elements, which means we can represent any linguistic concept in the Modelverse, as all elements by construction conform to the physical type model. To ensure this conformance relation is maintained at all times, physical mappers map linguistic elements onto physical elements. In these mappers, it is possible for a language engineer to encode custom instantiation policies. For the built-in formalisms, such as Class Diagrams formalism, these policies are predefined and languages created using those formalisms automatically get assigned a mapper with the expected instantiation semantics.

Any element in the logical dimension can take the role of the model M – indeed, everything in the Modelverse is a model, and all models have a type model.

A user interacts with the Modelverse through the API exposed by the MvK. This user needs not be a human interacting through code: it can be a front-end, allowing a more user-friendly use of the Modelverse. A few examples of front-ends include a visual front-end, a human-usable textual notation, or any (formalism-specific) simulator, that interacts with the Modelverse to simulate the model.

A user request is handled by the MvK as follows. A user, or front-end, sends a request to the MvK. The MvK then redirects this request, after checking whether it is a valid operation, to the Modelverse, where the appropriate modifications and/or queries are performed. The result is returned in the form of a log, which either lists the changes performed as a result of the operation, or, in the case of a non-modifying operation, returns the result of the operation. In the case of an error, the log specifies an error code and message, which the user can inspect to decide on the best course of action.

Controllers

Messages in MVC are processed by a series of controllers that ensure consistency among clients and between view(s) and the model related to the view(s). All operations are sent in the form of a client changelog. Listing 1 shows a sample client changelog that requests to create a new place in a Petri nets model. A user can take actions such as create, read, update, or delete elements in a model (CRUD operations); load a model; add or remove elements from a view; save or execute a model or set of models; and more. These requests are sent into the system through a message forwarder that routes messages to the proper controller for processing. Similarly, responses return to the clients via a response forwarder. Forwarders make use of a simple publish-subscribe pattern to enable clients (and controllers) to subscribe to messages of interest. For example, a client would subscribe to receive updates for a specific view of interest. A client can be subscribed to many views and would receive all updates for each view to which they subscribe.

Listing 1. Sample Create Client Changelog

In the following, we describe how models are represented in the MVC and present the four controller types.

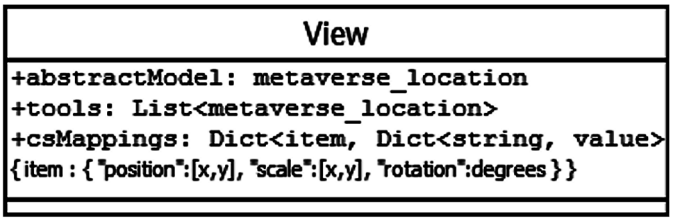
```
{
  "operation_name":"create",
  "location":"hergin.views.petrinet1.view",
  "am_location":"hergin.models.petrinet1",
  "element":"P5",
  "type":"hergin.formalisms.PetriNet.Metamodel.Place",
  "cs_attributes":{
    "position": [ 111,222],
    "scale": [1,1],
    "rotation":0
  },
  "attributes":{
    "token":5,
    "name":"P5"
  }
}
```

Models and Views in MVC

MVC separates models into two categories: abstract models and views. An abstract model represents the abstract syntax of the model of a system. It conforms to the metamodel of a given DSL. A view is a model that conforms to the view metamodel, presented in Figure 4. It references an abstract model and contains a set of elements from the abstract model that are included in the view.

The view also maintains a list of tools that are other models necessary to using the specific view. The most common example of a model to be found in the tools list is a concrete syntax model. By varying the concrete syntax between views, different users may view the same model using varying visualizations (e.g., textual or graphical). Other models included in the tools list could be custom executable models used for simulation or models defining transformation semantics relevant to the model. The tools list is controlled by the user. Models may be added or removed from this list as deemed necessary

Figure 4. The view metamodel



by the user. Finally, concrete syntax information of the model is stored in the view, since it is specific to a representation of the model and, therefore, does not belong to the abstract model. The csMappings dictionary defines the assignment of concrete syntax values to every element from the abstract model that is included in the view. In a graphical view, these values indicate, for example, the absolute position, rotation angle, and size scaling of the element.

This distinction between views and abstract models allows modelers to work with portions of a model and in their own representation, arrangement, and sizing for the elements of their view. The internal consistency can be ensured when elements are updated since all views reference the original elements from the model, but when an element is created or deleted it is possible to lose consistency. Furthermore, internal consistency is not sufficient for a multi-user context. It is also necessary to ensure consistency across many concurrent users.

Model Controller

A model controller manages a given model and all views of that model. Consistency of a model and the associated view(s) is maintained by the model controller. A model controller receives and processes all messages for a specific abstract model and its view(s). The model controller uses the publish-subscribe pattern of the message forwarder to accomplish this goal. Available model operations include CRUD operations on elements and loading a model. Available view operations include adding or removing elements and tools. Whenever a model operation alters a portion of the abstract model, the resulting changelog is sent to users using any view that references the updated portion of the model. The view operations are guaranteed to have no side effect on the model and thus do not need to update users of other views. Users may also send batch changelogs. The batch client changelog must contain only client changelogs intended for a single model controller or the package controller (discussed later). A batch client changelog is merely a convenience for sending large sequences of client changelogs together and is treated as a sequence of operations.

Incoming client changelogs are queued to be processed in the order they are received. The message queue is a FIFO queue. Thus, operations which originate from a later client changelog may fail based on the results of a previous message. To enable automatic resolution of conflicting messages, a message must "win" and a message must fail. By allowing the earlier message to win, the system does not need to process any rollback operations due to conflicting messages. The scenario is mitigated by the publishing of relevant update messages ensuring users each possess a consistent and current copy of the model. Thus, while it is possible that a set of users could send conflicting messages simultaneously, the users would be notified of the resolution for both messages in real time. Furthermore, users are notified of updates generated by other users in real time which reduces the likelihood of conflicting messages being generated.

The execution of operations is not performed at the model controller directly. Rather, the model controller coordinates the processing of operations within the MvK which directly processes all modeling operations. The MvK provides a more restrictive API designed. The MvK provides a pure modeling system where all entities are modeled and every entity is treated as a model. The model controller provides a layer of abstraction on top of the MvK for multi-user, multi-view modeling. The MvK is a simple system designed to process a sequence of basic model operations. The model controller ensures that the sequence of messages is valid and separates models and views which are treated uniformly within the MvK.

Supercontroller and Node Controllers

Model controllers manage all model and view related operations. For each model there is a single model controller. However, the system may have an untold number of models which would need to each have a model controller. If the system maintained these model controllers at all times, a significant amount of resources would be wasted on model controllers that are not in use. To mitigate this issue, we spawn model controllers on an as needed basis. Controlling which model controllers are active and assigning the model controllers to resources is the primary responsibility of the supercontroller.

The supercontroller monitors all client changelogs entering the system. When a client changelog is encountered, which requires a model controller that does not exist, the supercontroller queues the changelog. Any client changelog intended for an active model controller is ignored by the supercontroller. The supercontroller has an associated message processor that handles queued changelogs. The first message encountered for a model controller causes the message processor to generate a temporary backlog for further changelogs and the process of creating a new model controller begins.

The model controller is assigned to a node controller. Node controllers are abstract representations of available hardware computing resources (e.g., nodes in a cluster, CPUs, cores). Node controllers keep track of their assigned model controllers and can generate a load value based on the number of model controllers and other criteria as needed. The load value is used to determine to which node controller a model controller is assigned. Currently, the load value is based on the number of active model controllers, but in the future this value could also take into account frequency of incoming messages or number of users receiving updates for the node's active model controllers.

The supercontroller is responsible for spawning model controllers and maintaining the balance of nodes within the system. The supercontroller ensures there is only ever a single model controller for a given abstract model and its related view(s). The supercontroller also has privileged access to model controllers and may reassign model controllers to new nodes in order to maintain balance in the system. Model controllers could be reassigned based on the appearance of new node controllers (which may be added dynamically) or significant change in the load for a given model controller.

Package Controller

The package controller handles requests not associated with a specific model. In particular, it handles saving, executing and listing models. The latter operation returns a list of available models given certain criteria (e.g., all those conforming to a metamodel or all views of an abstract model). The package controller is a singleton entity meant to serve all active users. To accomplish this goal, the package controller uses a worker pattern. It fairly distributes all client changelogs to a pool of workers that handle the changelog and return a response through the response forwarding service. The execute operation may generate changes to multiple models. In this scenario, the package controller workers can send requests to the model controller through backend communication to ensure consistency is maintained, for example when executing model transformations. Similar to model controllers, the package controller does not directly process operations. Rather, it coordinates operations in the MvK to complete the requested operations.

Client

In this section, we describe a client for the MVC architecture that supports a user interface for collaborative and multi-view modeling. Clients discussed here are mainly graphical user interfaces (GUI); however, textual as well as command-line based clients are also supported.

AToMPM

AToMPM provides an in-browser GUI client for the user. Therefore, there is no installation required to perform modeling tasks. In the back-end, a node.js web server that hosts the pages. The main features of the web-based GUI are:

- Creating DSLs.
  - Creating metamodels.
  - Designing and assigning concrete syntax to metamodel elements.
- Manipulating models through an HTML5 canvas.
  - Creating, updating and deleting elements.
  - Undoing, redoing changes.
  - Copying and pasting elements.
- Executing and debugging model transformations.
- Collaborating with other users in real time.

Statechart-Driven Architecture

Both the back-end and front-end of the GUI client rely on Statecharts for the execution of various features. Basically, when the back-end server starts, it parses the server Statechart and performs the actions encoded in the states. Figure 5 depicts an excerpt of the server Statechart. The main advantage of using a Statecharts-driven architecture is separating the behavior from the structure and layout of the features.

There are two ways in which models are received from MVC: open the model to be edited, or load the model as a tool. From the point of view of MVC, they are both load model requests. However, on the client side, these two requests have a different behavior. The former means that the client renders the elements of that model on the canvas to modify the model. The latter means that the model is to be executed as a tool. Tool execution is performed by executing a required Statecharts model attached to the requested model. As a result of the execution, a toolbar consisting of buttons appears on the canvas. Since the behavior is encoded in the Statechart of the tool, there can be various kinds of actions attached to each button (e.g., creating model elements, executing a model transformation, modifying the appearance, restyling elements). These actions are not limited to the model itself, but can also let AToMPM communicate with external systems, such as an analysis tool that computes the coverability of Petri nets. Figure 6 illustrates a tool that has a toolbar with three buttons, where the first two create Petri net places and transitions, and the third one the model currently loaded on the canvas.

Tools and other models connected to them (e.g., toolbars, Statecharts, buttons) are also models. Thus, AToMPM can read them in edit mode and let the user do the necessary customizations.

Figure 5. Server statechart

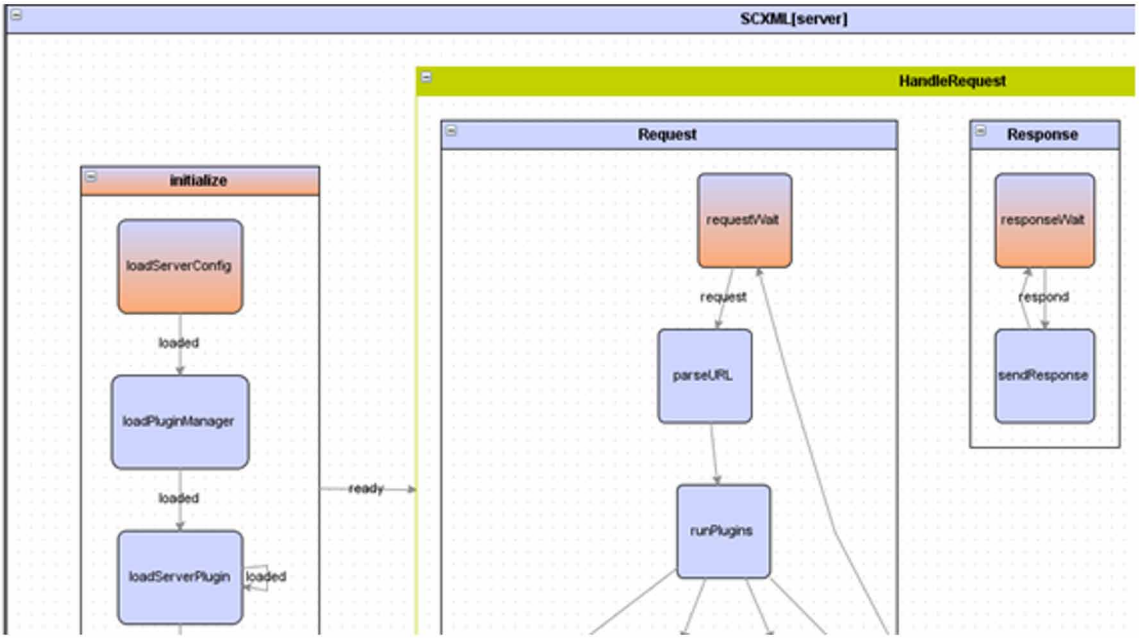
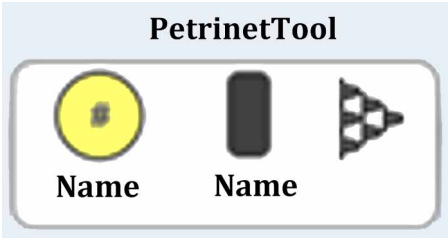


Figure 6. Petri nets Tool



Collaboration Scenarios Supported by AToMPM

AToMPM provides a flexible environment to implement the collaboration scenarios mentioned earlier. Clients do not operate on abstract models directly, but on views. Therefore, in order to create a model, the user must create a view. However, the model itself does not need a view to exist. In order for a user to get updates from a view (i.e. reading a model), he subscribes to that specific view of the model. Many users can subscribe to the same view at the same time, which implements the multi-user single-view scenario. Users can create multiple views for a model by either adding elements from the model to the view or by creating new elements. This implements the multi-view single-model scenario. Users can also subscribe to more than one view, which is required for both multi-view multi-model and single-view multi-model scenarios. As long as a user subscribes to a view, he gets the updates from that view, regardless of how many models that view projects.

Conflicts are inevitable while working with multiple users in a system. AToMPM solves these conflicts in a live online fashion, where all operations either fail or succeed immediately. Additionally,

AToMPM also provides a lightweight chat system between users in case of a manual conflict resolving mechanism, where users can communicate with each other in case of a conflict. The following section discusses how conflicts are handled for each collaboration scenario by the MVC.

SOLUTION AND RECCOMENDATIONS FOR CONFLICTS RESOLUTION IN MULTI-VIEW MODELING

Thus far, this chapter has discussed several challenges related to implementing multi-view modeling especially in a multi-user context. The chapter has also introduced the details of the AToMPM architecture, in particular MVC: a multi-user, multi-view modeling system. However, the details of one sample system do not cover the full breadth of the challenges for multi-view modeling. This section discusses the solutions as present in MVC and contrasting options for addressing these challenges.

Multi-User Single-View

The most basic scenario to consider is a model with a single view used concurrently by multiple users (i.e., multi-user single-view). Here the users are sharing a resource. In this situation, the most straightforward solution is to provide all users with access to a centrally controlled resource where consistency is maintained. This is the approach taken by MVC where the model controllers provide this single central resource that ensures consistency. However, this solution is limited to *always-online* systems. MVC is such an always-online system where access to the central resource is expected. However, some contexts may not favor an always-online solution and prefer periodic updates. In these scenarios, solutions that have been introduced for distributed database systems can be used. The system may maintain local copies and update with the central repository periodically.

To ensure consistency among distributed local copies it becomes necessary to maintain logs of operations and define a commit scheme which handles rollbacks in the case of conflicting operations causing an operation that succeeded locally to fail during commit. In these scenarios, users may lose portions of previously completed work due to the inability to guarantee conflict free scenarios. Always-online systems such as MVC may experience similar scenarios due to latency. Consider the case where a user sends a delete operation while, at the same time, another user sends an update operation for the same element. If the update is received first, both operations will succeed; but if the delete is received first, the update will fail. However, the user's action is not considered complete until the model controller has sent a confirmation of success. Therefore, in practice the system will display the element deleted and then inform the user that the update failed. Thus, the user is aware of the scenario immediately and may take the necessary steps. In an offline model with only occasional update, the user might follow a series of operation which are no longer valid and have to retrace back to the point of conflict.

Here the challenge is ensuring a consistent view of the model for all users. We propose an always-online solution where a central source maintains consistency. The central source might be a distributed or replicated process that uses a strategy to coordinate many nodes, but the operation is only deemed successful when verified by this central source. The advantage is an immediate response to operations and eliminating the need for rollback systems.

Multi-View Single-Model

The second scenario expands the system to enable users viewing distinct segments of model or even distinct notations. Each segment and notation defines a unique view of the model, but users are still editing the same underlying model. In multi-view single-model, edits may be propagated from one view to another. MVC chooses to maintain all views of a given model in the same controller that manages the underlying model, specifically in the model controller. Model controllers are designed to notify users of any view which is affected by an updated element. In this way, the views maintain consistency by all editing the single, centrally located version of the abstract model and updates are published to users of any view that is impacted by the change. This solution follows directly from the MVC solution to multi-user single-view and employs a single central resource to ensure consistency. The MVC follows a two-tier solution (see Figure 7), but this could be expanded to a three-tier solution (see Figure 8). In these figures, boxes represent model controllers. In the three-tier variant, the views each have separate controlling resources. The model treats views as users, and end-users interacting with the views as wrappers of the model. The three-tier variant would reduce computation at the model controlling resource, but at the expense of additional layers of message passing when the model must be updated. MVC is designed with the expectation that the majority operations require execution at the model-controller resource. Therefore, MVC combines these layers to eliminate the network overhead of the additional message passing.

As with multi-user single-view, the MVC solution requires a live connection, but an offline solution as discussed for multi-user single-view would also be viable here. A two-tier solution would provide the same difficulties mentioned previously, but a three-tier solution would exacerbate the complications. Consider a three-tier system where a user might update a view successfully and then later the same

Figure 7. User, Model Controller Two-tier Solution for Scenario 2

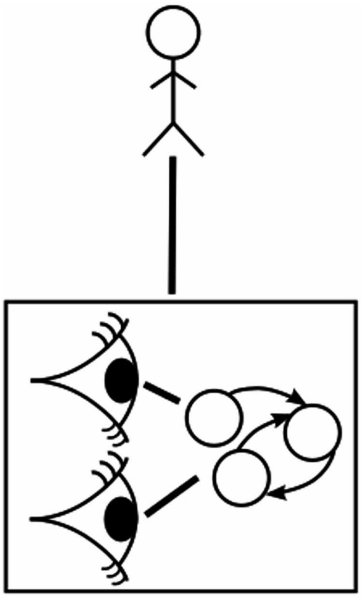
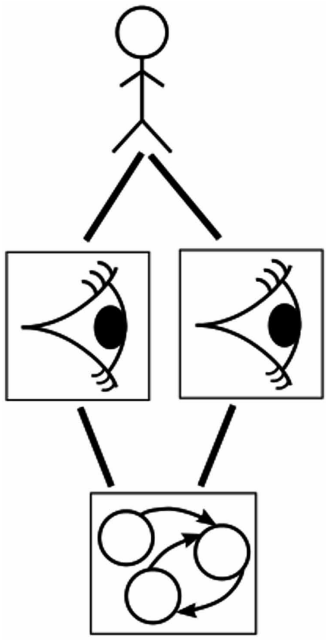


Figure 8. User, View, Model



changes might fail when trying to update the model. Here the view would immediately fail and rollback, but then the failure and rollbacks must be propagated to users which may have processed further operations. The complications become increasingly complex to maintain a functional offline solution with only periodic update. Thus, a live approach is preferable when possible. Further scenarios will only discuss a live approach to maintaining consistency. Though the complexity of conflicting scenarios increases further in subsequent scenarios, an offline solution remains feasible with user intervention required to maintain consistency without forcing roll back to a prior consistent state.

Multi-View Multi-Model

The third scenario introduces the added complexity of interrelated models. In many situations, a system contains many varied types of models. In these heterogeneous modeling systems, often models must rely upon each other. To fully support this scenario, facilities must exist to update models automatically when a related model is altered. However, the update process has several variations: live-link, full-copy, and shallow-copy.

A live-link solution maintains the actual model elements in only one locality; i.e., instances of linked elements are references to the original element rather than true model elements. A live-link system would enable models to reference elements from other models and use those elements as though they were defined locally. The Modelverse implements a live-link behavior for model elements. However, the MVC does not provide full support for updating users in this scenario. A user would need to separately subscribe to updates from the model of interest as well as any models related to the model of interest (see Figure 9). An ideal solution would update the user without need to manually subscribe to secondary (or potentially even further removed tertiary models). A planned solution, for future iterations of MVC, enables the model controllers to notify one another of updates through a separate publish-subscribe relationship. The solution would add further complexity to the messaging system of MVC and would likely impact performance when dealing with propagating these updates. In a full solution, the model controllers become both publishers of updates and subscribers of updates. Thus, operations may create cascading messages within the system depending on the complexity of interrelated models present. The current MVC solution enables only direct update to users, but consistency of a user's view of a model, including elements linked from secondary models, can be managed with the user subscribing to the additional models. Thus, a live-link solution to the multi-view multi-model scenario can be managed within MVC, but the current solution requires additional subscriptions by users to receive all necessary updates.

A full-copy system does not have live relations between models (see Figure 10). The link made at the time an element is added to the current model from a secondary model treats the secondary model as a static-unchanging entity. The system can create a full copy of the element (and any required elements) locally to ensure that updates in the related model do not impact the static elements. In a full-copy system, consistency can be maintained as in multi-user single view or multi-view single-model, because the system does not ever need to update the current model based on changes in other models.

A shallow-copy solution would provide means to accomplish the static relation of a full-copy system without strictly requiring a full copy of those elements. Thus, the system can reduce redundancy at the cost of requiring a live-link consistency management scheme. Here, the model elements can be represented by references until a change is made. Once either the original elements are altered in the secondary model or the reference elements are altered in the current model, the system must either maintain multiple versions of the model or perform a full copy and remove the live links. When the secondary model is

Figure 9. Live-link related models

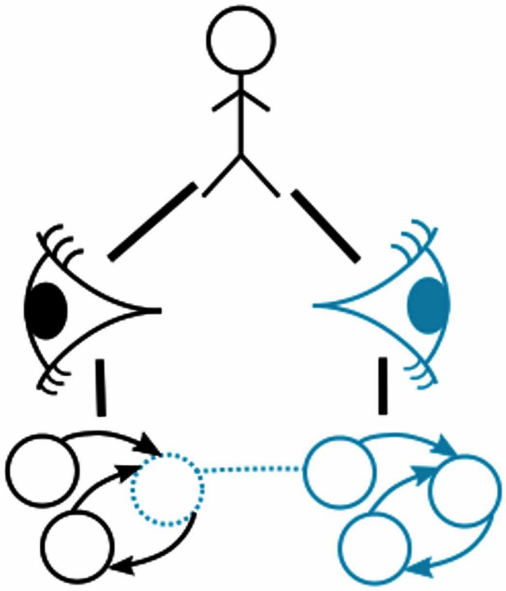
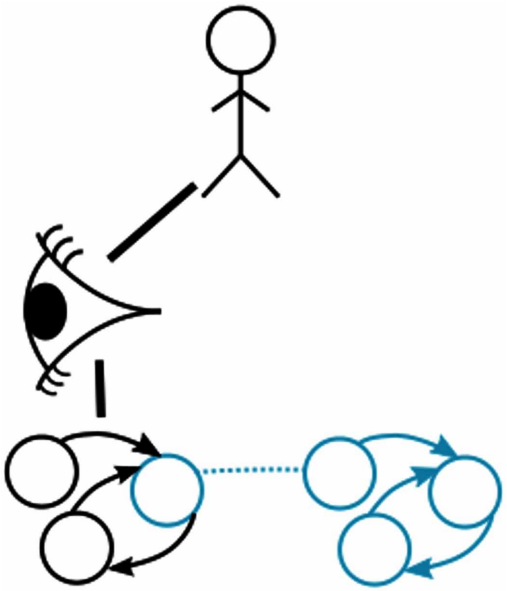


Figure 10. Full-copy related models

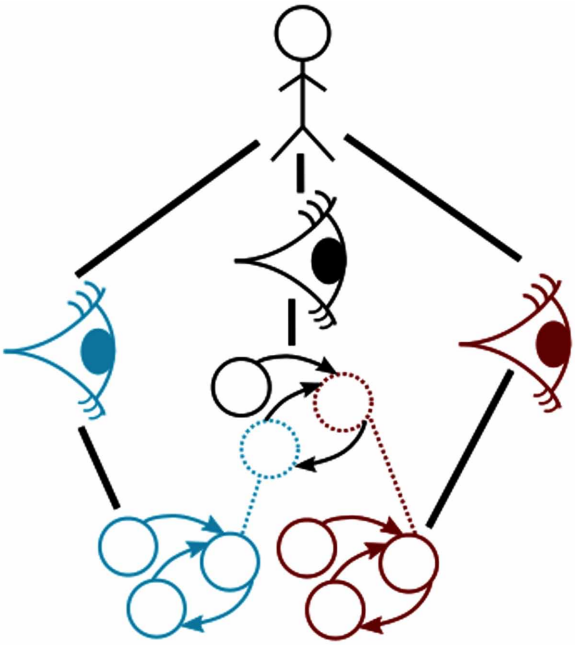


highly unlikely or not allowed to be updated, the shallow-copy solution presents some advantages over the other two solutions. However, in practice, disallowing a secondary model from being updated may not be possible, and the system must be able to handle elements in related models changing regardless of the likelihood of the elements changing.

Single-View Multi-Model

The final scenario for multi-view modeling can be refactored to a special case of multi-view multi-model (see Figure 11). Single-view multi-model covers visualizing elements from many models with potentially diverse notations within a single view. Thus where multi-view multi-model had a current model that might be related to other secondary models, this scenario treats all of the interrelated models as the current model. If we refactor the scenario by adding a single model that incorporates the many models as secondary models (thus adding a level of indirection), the consistency can be maintained using any of the strategies from multi-view multi-model as appropriate, but the system must provide additional facilities to represent these models together. A naïve approach would be to provide separate panels for each notation required, but this approach lacks the ability to represent the logical links between these separate notations. MVC handles the refactored version of single-view multi-model. The disparate models must each be included in a single model using live-links. The unifying model is defined with a notation that unifies the disparate models. The unifying model may also include additional elements which are locally maintained. MVC, thus, can reuse the live-link solution from multi-view multi-model with the many models becoming secondary models. The refactoring requires intentional development in advance to integrate the models.

Figure 11. Refactored Single-View Multi-Model



FUTURE RESEARCH DIRECTIONS

AToMPM represents the first effort in providing a cloud-based architecture to support collaborative modeling. Future works include exploring a more complete solution to fulfill all multi-view multi-model scenarios. For that, the MVC needs to be extended to provide facilities for model controllers to publish updates internally to other model controllers. It is nevertheless of paramount importance to provide a scalable solution that does not overly impact the performance from a user's perspective.

Another possible venue for future research is to explore the combination of online and offline solution. WebGME relies on a document-based NoSQL database to maintain the models within the backend system. This system employs a commit strategy where clients may work offline and commit work at a later point. However, the current system does not explore the collaborative modeling scenarios and relies on user intervention to manage conflicting updates. Future research should explore including more automated conflict resolution within such a system.

CONCLUSION

In this chapter, we have discussed the growing trend toward collaborative environments that fit today's needs when teams of domain experts work in a coordinated manner to construct a system. We identified four distinct collaboration scenarios found in practice, combining multiple users, multiple views, and multiple models of the system under study. We argue that a cloud-based multi-view modeling environment addresses that need, and introduce the architecture of our prototype AToMPM, which is a cloud-based environment for collaborative modeling. AToMPM represents a first step toward supporting the need

for collaborative modeling through cloud technologies. Its core component, MVC, ensures consistency and synchronization among the artifacts produced by each stakeholder.

Furthermore, the chapter discusses the approach to addressing consistency and synchronization needs implemented in AToMPM. The discussion addresses each of the four collaboration scenarios. We also introduce competing approaches and compare to the approach implemented in AToMPM. AToMPM presents an *always-online* approach where users must have a live connection to the system at all times. This approach provides both synchronization and consistency. However, offline approaches are also possible such as that implemented by WebGME which must resolve sets of changes using a commit process. The offline approaches do not guarantee synchronization and must resolve (potentially complex) conflicting series of operations taken by users. Finally, we discuss some of the many challenges that remain to be solved including improved support for the fourth collaboration scenario and automation of conflict resolution in collaborative modeling systems to enable users to work both online and offline.

REFERENCES

Asana. (2015). *Asana*. Retrieved from https://www.asana.com/

Beck, K. (1999). Embracing change with extreme programming. *Computer*, *32*(10), 70–77. doi:10.1109/2.796139

Combemale, B., De Antoni, J., Larsen, M., Mallet, F., Barais, O., & Baudry, B. (2013). Reifying Concurrency for Executable Metamodeling. In *Software Language Engineering* (Vol. 8225, pp. 365–384). Springer. doi:10.1007/978-3-319-02654-1_20

Combemale, B., Deantoni, J., Baudry, B., France, R., Jézéquel, J.-M., & Gray, J. (2014). Globalizing Modeling Languages. *Computer*, *47*(6), 68–71. doi:10.1109/MC.2014.147

Corporation, M. (2001). *Microsoft C# Language Specifications*. Microsoft Press.

GEMOC Initiative. (2015). *GEMOC Studio*. Retrieved from http://gemoc.org/studio/

GitHub. (2015). *git*. Retrieved from http://git-scm.com/

Google Inc. (2015). Retrieved from Google Docs: http://docs.google.com

Hiya, S., Hisazumi, K., Fukuda, A., & Nakanishi, T. (2013). clooca: Web based tool for Domain Specific Modeling. *MODELS'13 Invited Talks, Demos, Posters, and ACM SRC.*, *1115*, 31–35.

Kelly, S., Lyytinen, K., & Rossi, M. (1996, may). MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. *Conference on Advanced Information Systems Engineering*. Springer. doi:10.1007/3-540-61292-0_1

Kiczales, G., & Hilsdale, E. (2001). Aspect-oriented programming. *Software Engineering Notes*, *26*(5), 313. doi:10.1145/503271.503260

Kühne, T. (2006). Matters of (Meta-)Modeling. *Software & Systems Modeling*, *5*(4), 369–385. doi:10.1007/s10270-006-0017-9

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., & Thomason, C. (2001). The generic modeling environment. *Workshop on Intelligent Signal Processing*.

Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., & Jurácz, L. (2014, October). Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. *Multi-Paradigm Modeling*, *1237*, 41–60.

Object Management Group. (2012, Apr.). *Information technology - Object Management Group Unified Modeling Language, Superstructure*. ISO/IEC 19505-2.

Özsu, T., & Valduriez, P. (2011). *Principles of distributed database systems*. Springer Science & Business Media.

Stahl, T., Voelter, M., & Czarnecki, K. (2006). *Model-Driven Software Development -- Technology, Engineering, Management*. John Wiley & Sons.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). *EMF: Eclipse Modeling Framework* (2nd ed.). Addison Wesley Professional.

Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., & Ergin, H. (2013). AToMPM: A Web-based Modeling Environment. *MODELS'13: Invited Talks, Demos, Posters, and ACM SRC. 1115.* CEUR-WS.org.

The Eclipse Foundation. (2015). *Orion*. Retrieved from http://eclipse.org/orion/

Trello Inc. (2015). *Trello*. Retrieved from http://www.trello.com

Van Mierlo, S., Barroca, B., Vangheluwe, H., Syriani, E., & Kühne, T. (2014, oct). Multi-Level Modelling in the Modelverse. In *Proceedings of the Workshop on Multi-Level Modelling.* CEUR-WS.org.

KEY TERMS AND DEFINITIONS

Abstract Model: The abstract syntax of a model representing its essence: entities, relations, and properties that conform to a metamodel.

Always-Online: A system where users are connected to the cloud all the time, as opposed to offline.

Domain-Specific Language: A modeling language that is adapted to a particular application domain, where models are described using a notation fit exactly to the domain.

Metamodel: A model that defines the abstract syntax of a modeling language.

Multi-View Modeling: Modeling activities that users can perform through one or more views.

Node: A computation resource, such as a computer machine, a CPU, or a process.

View: A projection of an abstract model, showing parts or all elements of the model in a specific concrete syntax representation.