Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure

Miklós Maróti[2], Tamás Kecskés[1], Róbert Kereskényi[1], Brian Broll[1], Péter Völgyesi[1], László Jurácz, Tihamér Levendoszky[1], and Ákos Lédeczi[1]

[1] Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, USA
`akos.ledeczi@vanderbilt.edu`
[2] Bolyai Institute, University of Szeged, Szeged, Hungary
`mmaroti@math.u-szeged.hu`

Abstract. The paper presents WebGME, a novel, web- and cloud-based, collaborative, scalable (meta)modeling tool that supports the design of Domain Specific Modeling Languages (DSML) and the creation of corresponding domain models. The unique prototypical inheritance, originally introduced by GME, is extended in WebGME to fuse metamodeling with modeling. The tool also introduces novel ways to model cross-cutting concerns. These concepts are especially useful for multi-paradigm modeling. The main design drivers for WebGME have been scalability, extensibility and version control. The web-based architecture and the constraints the browser-based environment introduces provided significant challenges that WebGME has overcome with balanced trade-offs. The paper describes the architecture of WebGME, argues why the major design decisions were taken and presents the novel features of the tool.

Keywords: DSML, metamodel, collaboration, web browser

1 Introduction

Applying Domain-Specific Modeling Languages (DSMLs) for the engineering of complex systems is becoming an increasingly accepted practice. Model Integrated Computing (MIC) is one approach that advocates the use of DSMLs and metamodeling tools [1]. The MIC open source toolsuite centered on the Generic Modeling Environment (GME) [2] has been applied successfully in a broad range of domains by Vanderbilt [3–7] and others [8–13]. (Note that this is just a selected subset of domains.) Design space exploration with sophisticated tool support [14] and the seamless integration of multiple third party software packages [15] are the most striking examples of the power of MIC.

However, the widespread application of MIC has uncovered the limitations of the tools. GME was designed as a desktop tool for the creation of small- to medium-sized models by a single user (or a small group of co-located users). The models are typically stored in a single file in a proprietary format. The elementary modeling concepts are somewhat arbitrary and are closely tied to their respective

visualizations. The metamodels and the instance models are decoupled requiring a translation step, making DSML evolution cumbersome. As a result of the ever-increasing expectations, additional features have been added to the tools. For example, to address scalability concerns both in the size of the models and in the number of concurrent users, GME was extended with a Subversion-based backend to store the models in multiple XML files [16]. However, pessimistic locking to avoid incompatible changes to the model by multiple users proved to be inflexible and non-scalable. It became clear that these are fundamental limitations that need to be addressed at the very core of the architecture.

To address these limitations, we created WebGME, a web-based cyberinfrastructure to support the collaborative modeling, analysis, and synthesis of complex, large-scale information systems. The metamodels and the corresponding domain-specific models are tightly integrated via prototypical inheritance and stored in the cloud. Online collaboration, model version control, complex DSMLs and large instance models are transparently supported. Clients are web browser-based, resulting in platform independence and doing away with installation and upgrade issues. The user interface supports several built-in visualization techniques. Multiple APIs are provided to interface with existing external tools as well as to enable the development of custom domain-specific visualization components and code generation tools.

This paper presents the architecture and major design decisions of WebGME. Section 2 describes the meta-metamodel and the support for DSML specification. Section 3 describes the collaboration approach provided by WebGME. The data model is presented in Section 4, the overall architecture is described in Section 5, while model visualization support is summarized in Section 6. Section 7 illustrates how multi-paradigm modeling is supported by WebGME. The paper concludes with a brief overview of related work and conclusions.

2   Modeling Language Specification

The metamodel specifies the domain-specific modeling language. The metamodeling language consists of a set of elementary modeling concepts. These are the basic conceptual building blocks of any given approach and corresponding tools. It is the meta-metamodel that defines these fundamental concepts. These may include composition, inheritance, various associations, attributes and other concepts. Which ones to include, how to combine them, and what editing operations should operate on them and how are the most important design decisions that affect all aspects of the infrastructure and the domains that will use it.

Hierarchical decomposition is the most widely used technique to handle complexity. This is the fundamental organization principle in this tool, too. Copying, moving, or deleting a model will copy, move, or delete its constituent parts.

The single most important distinguishing feature of GME has been the unique use of prototypal inheritance. Each model at any point in the composition hierarchy is a prototype that can be derived to create an instance model. Derivation creates a copy of the model (and all of its parts recursively, i.e., a deep copy), but

it establishes a dependency relationship between the corresponding objects. Any changes in the prototype automatically propagate to the instance. Of course, instances can be used again as prototypes for further specialization.
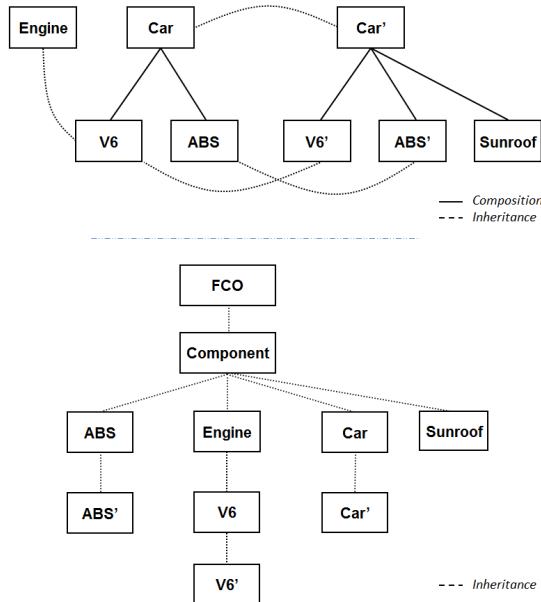

Fig. 1. Prototypical inheritance in WebGME

To illustrate this concept, consider Figure 1. Let's suppose that we have a DSML for modeling cars and the language has a concept called *Component* for modeling various parts of a car. The top half of the figure shows that we have created a simple *Car* model that has an *ABS* brake and a *V6* engine. Notice that an *Engine* model was also created and *V6* is derived from it. Note that all these models are derived from the *Component* model specified in the metamodel (not shown). Also note that solid lines represent composition, while dashed line show inheritance. Now if we take *Car* as a prototype and create a derived *Car'* instance model from it, you can see that it will contain a new instance of *ABS* and a new instance of *V6* as well. Note that we also added a *Sunroof* to this new car model. The corresponding inheritance tree is shown in the bottom half of the figure.

This approach is markedly different from inheritance in OO programming languages or in other modeling languages such as UML. First of all, it combines composition and inheritance. Note that Smalltalk and JavaScript have prototypical inheritance also, but it does not create new instances down the composition hierarchy. Second, inheritance is a "live" relationship between models that is continuously maintained during the modeling process. That is, any changes to a model propagate down the inheritance tree immediately. For example, if we change a property of the *Engine* model shown in Figure 1, it will also change in *V6* and *V6'*. On the other hand, if that property was already changed in *V6*, then the property modification in *Engine* will not have an effect on either *V6* or *V6'*. But if we reset the value in *V6* to the one inherited from *Engine*, that will propagate to *V6'* unless it has been overridden there beforehand. These rules also mean that when a model is deleted, so are all of its instances and instances of all of its children recursively. This variant of inheritance is a very powerful way to help the modeler handle the inherent complexity in large models and intricate DSMLs.

The novel idea in WebGME is to blur the line where metamodeling ends and domain modeling begins by utilizing inheritance to capture the metamodel/model relationship. Every model in a WebGME project is contained in a single inheritance hierarchy rooted at a model called FCO, for First Class Object, as shown in Figure 1. Metamodel information can be provided anywhere in this hierarchy. An instance of any model inherits all of the rules and constraints from its base (recursively all the way up to FCO) and it can further refine it by adding additional metamodeling information. This is a form of multi-level metamodeling with a theoretically infinite number of levels.

As a result of this approach, 1) metamodel changes propagate automatically to every model; 2) metamodels can be refined anywhere in the inheritance and composition hierarchies; 3) partially built domain models can become first class language elements to serve as building blocks; and 4) different (meta)model versions can peacefully coexist in the same project.

2.1 Meta-metamodel

A WebGME project is a collection of metamodels and models in a single composition hierarchy. Note that the words model and object will be used interchangeably throughout the rest of the paper. When the user creates a new project, it contains two objects called *Root* and *FCO*. *Root* is the root of the *composition* hierarchy, so every object in the project will be a node in the composition tree rooted at *Root*. *FCO* is contained by *Root* and it is the root of the inheritance hierarchy. Neither *Root* nor *FCO* can be deleted. Furthermore, the meta-metamodel of WebGME specifies that *Root* can contain an FCO and consequently, it can contain any other type of object. The meta rules of Root cannot be modified either. In addition, the initial meta rules of FCO are empty. The reason is that meta rules are inherited and they can only be extended and never restricted. For example, if we want a DSML where any model can contain any other kind of model, we can simply specify that *FCO* can contain *FCO*s. From then on, there cannot be any restrictions on composition in this DSML.

Additional relationships between objects can be expressed with *pointers* and *sets*. A pointer is a binary directed named association. Any object can have any number of different pointers. A pointer definition includes its name and a list of possible target objects. The latter restricts valid targets of the pointer to an element of the list, i.e., a model or any model derived from it. For example, if we want a pointer to be able to point to any other object in the project, we can specify the list of its valid targets to contain FCO.

A pair of pointers can be visualized as a connection. For example, the default WebGME editor takes any object with two pointers with the reserved names of *src* and *dst*, displays the object as a connection and supports the customary editing operations. Otherwise, connections are ordinary models; they can contain children, have other pointers and can be derived, etc. Therefore, the connection concept as such is not part of the meta-metamodel.

A set is a named association between one object and an unordered set of other objects. It can be considered a collection of pointers. A set is similar to

UML aggregation, but the objects the pointers can point to are not limited to be of the same kind. Any object can have any number of different sets. A set specification is exactly the same as that of a pointer.

An *attribute* defines a property of a model. The metamodel specifies the type of the attribute and a default value. Currently supported types are string, integer, float and enumeration, but these can be easily extended.

An *aspect* (also called *view*) is a subset of a model's children. Each model has a default aspect that contains all its children. Additional aspects help manage the complexity of models with many children. For example, a model of a car might have separate aspects for its mechanical, electrical and hydraulic design.

Prototypical *inheritance* is at the core of the WebGME meta-metamodel. Inherited children cannot be deleted, but new children can be added. Associations within the composition tree rooted at the base are adjusted to refer to the corresponding new instances. Associations pointing outside of the base model tree preserve their targets. New associations, attributes and aspects can be added. Association targets and attribute values in an instance can be modified. The goal of these rules is to be able to extend the inherited model, but prevent restricting it. New meta specification can be added to any object anywhere in the inheritance hierarchy. The rigid line between modeling and meta-modeling simply does not exist any more. A brief video tutorial (available at `http://webgme.org`) explains and demonstrates these concepts.

2.2   Cross-cutting concepts

Cross-cutting concepts are always difficult to model. In GME, the only way to capture relationships between models in different branches and/or levels of the composition hierarchy is through pointers and sets. However, the visual depiction of such associations is not intuitive at all since most tools display models according to composition, that is, they typically show the children of one model in one window (grandchildren may show up as ports). The target of a pointer can be indicated by its name and navigation to it can be supported, for example, by a double click operation, but an intuitive visual depiction of such relationships is sorely missing. For example, a connection between far away objects is supported by the meta-metamodel, yet there is no way to actually show it. To address this problem, WebGME introduces the concept of *crosscuts*.

A crosscut is a collection of objects that the modeler wishes to view together. Currently, the user can manually drag objects into a crosscut view. In the near future, we will define a simple query language that can be used to issue one-time queries to collect models from anywhere in the composition hierarchy. Existing associations between objects in a crosscut are depicted by various lines between the objects. For example, inheritance is shown similar to UML class diagrams, while pointers are visualized with lines and arrows. In addition to visualization, the main utility of crosscuts is that they serve as association editors. The target of pointers and set membership can be edited here. Deleting a model from a crosscut does not delete the object from the project, it simply removes it from the given crosscut.

Each crosscut has a context model, the designated container for new model elements created in the crosscut. (Note that it is atypical to create new models since a crosscut is meant to be a collection of already existing models. However, a connection is a model with two pointers, so allowing new connections in crosscuts was the motivation behind this design decision.) The default context is *Root*. As *Root* can contain anything, crosscuts can be freely constructed. However, if the modeler chooses a context different from *Root*, the composition rules of the metamodel apply (even though crosscut containment is not composition). This is actually a great way to control and manage crosscuts. On the flip side, if one wants a crosscut with no constraints, but wants to avoid creating too many crosscuts in *Root*, one can simply create a model and specify that it can contain *FCO*s. Any instance of such a model can now serve as the context for unconstrained crosscuts.
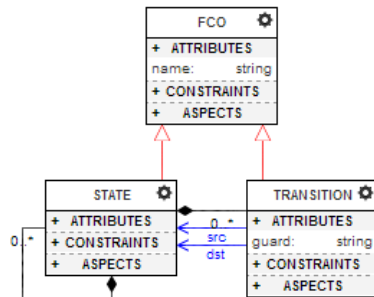

Fig. 2. HFSM Metamodel

One use for crosscuts in WebGME is for metamodeling. Recall that meta information can be specified anywhere in the composition hierarchy. Therefore, there is no single model to show to edit the metamodel of the DSML. In WebGME, a crosscut is created for the metamodel where the user drags in all models that need to contain DSML specification. It is there and only there, where meta information can be specified. Of course, the metamodel is a special crosscut, because a new association created there does not actually create a new instance of a pointer, for example, but instead specifies that the given kind of pointer of a model can point to the selected model (and its instances).

Consider Figure 2 that depicts the metamodel of a simple hierarchical finite state machine (HFSM). It shows that both State and Transition are derived from *FCO*. Note that unlike in any other tool we are aware of, this inheritance relationship was not drawn explicitly by the user. Instead, when the State and Transition models were created in the first place, they were instantiated from a model, in this case, *FCO*. The metamodel only displays these already existing inheritance relationships; they cannot be edited per se. On the other hand, the associations in Figure 2 were created in the meta crosscut. For example, the *src* and *dst* pointer specifications were drawn by the user specifying that a transition represents a relationship between two states. The default WebGME editor, in turn, will show these as connections (explained above) as expected in an HFSM.

3   Collaboration

Large-scale information systems modeling poses unique challenges that ad-hoc use of simple modeling tools cannot adequately address. As the amount of data

stored, manipulated, and analyzed and the number of users and stakeholders spread across multiple institutions increase, the coordination of the modeling process becomes exponentially more difficult. There are known domain specific solutions targeting the versioning and configuration of coarse grained model hierarchies, such as the approach for CAD designs [17], but most domain models are much finer grained. Domain specific (meta)modeling tools need to scale up to support decentralization, collaboration, domain evolution and at the same time, ensure consistency.

Because of the very high level of interconnectedness among the models, even simple operations (such as copy or delete of a hierarchical model) can affect a large part of the model database. In our experience, this renders lock- or partition-based cooperation infeasible, especially when the system must always present consistent information to users and external tools. In large-scale modeling, users should have decentralized access to the model database where concurrent modifications even to the same set of models are possible. WebGME addresses this challenge by introducing a very lightweight branching scheme where different branches structurally share the same models if those have not been modified (see Section 4).

WebGME supports online collaboration where changes are immediately broadcast to all parties and everyone sees the same state. This is similar to how Google Docs works, except here the models have a much richer data model making consistency management more challenging. Since branch updates are very cheap and store only those objects that are explicitly modified, these changes can be broadcast to all participating parties and concurrent editing conflicts can be detected, retried, or rejected with immediate visual feedback.

The exact datamodel supporting the quick dissemination of objects between the server and clients is described in the next section. At this point, it is enough to know that every revision in the object database is uniquely identified by the hash value of a commit object, and that a commit object uniquely describes all models in the model hierarchy at a particular time instant.

Clients can freely create commit objects and send branch update messages to the server. Each branch update request contains the hash values of the old and new commit objects, and the database backend verifies (among others) that the old hash value matches the current branch hash before it is updated. With this protocol, the backend ensures that each branch has a linear history and rejects those branch updates that would fork the branch. If a branch update is accepted, then the new hash of the commit object is broadcasted to all clients. If a branch update is rejected (because another client has made a concurrent change and the old hash value does not match), then the client has the following three options: 1) reject the change made in the user interface and present the new version to the user, 2) automatically fork the branch (creates a new branch) and indicate this to the user, or 3) perform a merge of the local modifications to the branch with the changes already in the database and retries the branch update request. Currently, we support option 1 only, as automatic merging is not yet implemented.
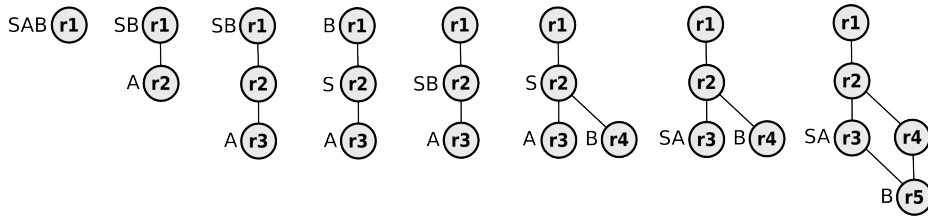
Fig. 3. Evolution of a branch during concurrent editing

As a simple example, let us consider two users concurrently editing a model hierarchy (see Figure 3). Initially, there is only one revision (r1), and both users (A and B), as well as the server (S), agree that the head of the branch is r1. Then user A makes two quick changes on his client computer (r2 and r3), which are immediately displayed (since all changes are performed synchronously without communicating with the server), and two branch update messages (r1→r2 and r2→r3) are sent to the server. After some time the server gets the r1→r2 message, at which time user B still sees revision r1, the server thinks that the head is at revision r2, and user A sees revision r3. The server sends out a notification to all clients that the head of the branch has changed to r2. User B displays the new head (r2), but user A ignores this update because it sees that he is already ahead of r2 and knows that it has already sent those updates.

At this point user B makes a concurrent change (r4) branching the history, but she does not know this yet, and notifies the server with an r2→r4 message. The server gets the r2→r3 message from A first in this example, so it updates the head to r3 and notifies all clients again. Then the server gets the r2→r4 update message from user B, but ignores it since the old revision in the update message (r2) does not match the current revision (r3). User B will get the head change message from the server with revision r3, and sees that this is not an ancestor of his current revision (r4). At this point user B knows that the branch has forked and his version is not the official one. Currently, we notify user B of this situation and discard his change, but with automatic merging, the client program for user B can perform the merge (r5) and can retry the branch head update request with r3→r5. If user A does not modify the branch in the meantime, then the server accepts this update and notifies user A.

In the near future, WebGME will also support merging branches. This will allow for an additional kind of collaboration where users fork the project, work on their own branches and once they are ready with their modifications, merge the changes back into the master branch. The structural consistency of the models are maintained by each basic operation and verified at merging. In most cases, the system will able to perform the merge automatically, but should a conflict arise because of conflicting modifications to the same models, the system will reject the merge and offer manual or guided conflict resolution on the client.

The version control scheme already enables users to work on and analyze consistent snapshots of the database without stopping others from modifying

the models. This means that long running model translators, code generators or analysis tools can run while users carry on model building concurrently.

It is instructive to see why current solutions cannot meet the requirements of large scale modeling. Distributed Revision Control (DRC) is a well-known and scalable solution to source code management, but it requires clients to download full revisions to make changes [18]. However, we need to allow users to delete, copy, and move hierarchical models consisting of thousands of elements without downloading the internal structure of those models to the client. Moreover, as we have explained, the rich inter-model relations (such as inheritance) would also require the client to potentially download and modify an even larger set of objects. Many cloud-based information systems provide extreme scaling (e.g. Twitter, Facebook, Wikipedia, etc.), but do not provide branching and consistent snapshots. For example, Wikipedia has revision control of individual pages, but it does not allow users to concurrently fork the entirety of Wikipedia, update thousands of pages, and merge these changes back. Online collaborative text and diagram editors (e.g. Google Docs, Lucidchart, etc.) do not support branching and store individual artifacts separately with no integration. On the other hand, WebGME enables large-scale collaborative modeling with refactoring capabilities and consistency guarantees.

Beyond supporting collaborative online work, model evolution, and conflict resolution within individual projects, in the future, WebGME could foster model and language reuse on a much larger scale than it is currently practiced. Design publishing, discovery, and change tracking are poorly supported by current desktop-based model editors. We believe that the stimulating effects of Source-Forge, Google Code, and GitHub (among others) on code reuse can and should be replicated for model-based design. The WebGME infrastructure can serve as cloud-based live repository of DSMLs and corresponding domain model libraries as current online project repositories are heavily source-file oriented and, hence, inadequate for model-based design collaboration.

4 Data Model

In this section, we describe the underlying data model of the system as an object graph and explain how the elementary operations (copy, delete, move, instantiate, update, etc.) are efficiently implemented without sacrificing extreme scalability and data integrity. First, we present a simplified view of the proposed architecture, then indicate the real difficulties.

Our primary goal is to ensure structural sharing of model objects between different branches of the database. We achieve this by organizing the objects into a containment hierarchy tree, where each object in the database stores the identifiers of its children, but not that of its parent.

If an object (e.g., G in Figure 4) needs to be updated, we simply make a copy, assign a new identifier (G'), recursively copy the parents (C, A) and replace the old child identifiers with new ones. This way the old and new versions of the graph structurally share a large portion of their objects. Since we do not store

the parent in the child object, we can simply implement the copy operation as adding the identifier of an already existing model to the list of children. From the perspective of the user (and other tools), we have a new deep copy of the model, but in the database we did not have to recursively traverse its children because we just reused old content.
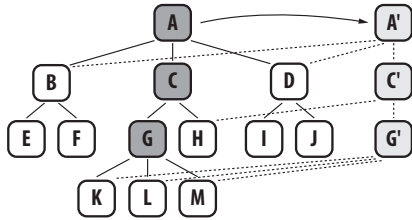

Fig. 4. Structural sharing of objects

We never modify any model object in the database, only create new objects that link to the old ones. Therefore, we can efficiently traverse and compare different versions of the model database and discover if large portions of their objects are the same. Instead of using standard identifiers for each object, we use a hash (SHA1) of the content of the model. Even if the same object is recreated on different clients (or the same modifications are preformed on the same client later), the modified objects will have the same hash value and only a single database object is created. The object database contains one root object for each version of the model, essentially tracking its evolution. The root objects are linked by commit objects that record parent commits, the new root object, and among others, the user who created the new tree. Another benefit of the use of hashes is that clients (browsers) can cache model objects freely, since they are not going to get modified, and can verify the integrity of the models.

The client (browser) does not need to download the whole database in order to perform operations on the models, only those objects that are necessary to be displayed to the user. For example, we can move, copy, or delete whole subtrees without ever downloading the internal objects. Note that the discovery of the model database is inherently asynchronous; the client needs to download new content, but modifications can be performed locally without server interaction except for eventually saving the new version of the model objects to the server. The saving of the new objects can be arbitrarily delayed, unless the user wants online collaboration with other users or wants to minimize accidental branching. Therefore, it is possible to support offline work where the user can continue editing the model without restriction if all the required content is already downloaded to the browser. The integrity of the modifications is maintained, and the changes can be uploaded to the database when connectivity is restored.

So far what we have described is very similar to how Git [19] operates, except we do not want to download the whole repository or the whole tree to the client, and we intend to use the browser as our management and editor tool. The real challenge is to track and update the rich inter-model relations in a consistent way without sacrificing the benefits presented above.

Imagine that we have an association between objects H and M in Figure 4, and we want to delete object G. Since all operations are hierarchical, object M, and therefore, the association between M and H needs to be deleted as well. As we have explained, the client loads only those objects that are absolutely

necessary, which includes the parents of loaded objects, so the browser knows of A, C, and G, but not of H or M. Somehow we need to change H without loading it and remove that association that is visible in the old version of H. WebGME stores associations not at M or H or both, but at their common ancestor, that is, at C. Therefore, when we want to delete object G we immediately know all external associations that tie an object within G to another object that is not inside G (these associations are stored at C and A), so we can remove these when object G is deleted.
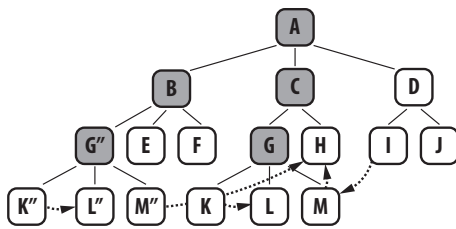

Fig. 5. Copying of associations

During a deep copy operation we also need to copy associations, but this case is more complicated than deletion because we need to distinguish internal and external associations and consider their direction. Suppose that we make a copy G'' of G and insert it in parent B (see Figure 5). An association is internal if both of its endpoints are within G, for example K-L. Internal associations are stored within the subtree of G because their common ancestors are also below G, so these associations are automatically copied. An external association pointing from an object below G to an object outside of G, e.g. from M to H, needs to be copied and the new association will point from M'' to H. An external association pointing from outside of G to an object within G, e.g. from I to M, is not copied because those reference a specific object. As we have seen, we have to maintain the direction of associations to properly maintain the semantics of copy and delete operations, and in this regard, associations behave like pointers in programming languages. Observe, that if G is loaded to the client and the new parent B of G'' is also loaded, then we can perform the copy operation and update all external associations at C and A without loading any new objects.

Movement of objects is the most complicated operation in the containment hierarchy, but even in this case, all associations can be properly updated within the already loaded parent objects. This means that all basic operations (delete, copy, move) can be performed in the client on arbitrary large subtrees without loading any new objects or even talking to the server.

The most important inter-model relationship is inheritance, which is significantly more challenging to support than associations. Simple changes in a base type can have an influence on all subtypes, but again we do not want to load all instances just because we modify the base type. Containment and inheritance interact in surprising ways, for example, deletions can have a cascading effect through containment and inheritance. To combat these, we dynamically compute the inheritance, where each object stores internally only those attributes and children that are different than those that are already present in its base type. This logic works even for associations, however, some extra logic is needed to allow the deletion of associations in instances. Currently, moving of objects

within basetypes that already have instances are not supported on the client because this would still require the traversal of the inheritance chain. Deletion and coping does not have this restriction, and currently we are investigating ways to be able to support moving of objects in this case, as well.

To support this datamodel and online collaboration, the database backend provides the following two services: 1) respond to load and save requests of objects that are identified by their hash, and 2) store the current hash value of the commit object for each branch and broadcast changes of this hash value to connected clients. Since each object is uniquely identified by its hash value, the load and save requests can be completely reordered and no coordination between clients is required (only hash collisions need to be monitored for safety). On the other hand, updates of the current branch hash has to be serialized and all new objects should already be in the database when the new hash value is updated and broadcasted. Once the branch hash value is broadcasted, the clients update their user interface and download all missing objects from the backend.

With the datamodel described above, users can concurrently delete, copy and move entire subtrees consisting of millions of objects with the same efficiency as operating on a single object. Moreover, all these operations can be performed immediately on the client with no database round trip, so WebGME can provide instant visual feedback even if the network connection is slow or disconnected.

5 Architecture

WebGME is designed from the ground-up as a modern web application, using a *single page* interface and advanced AJAX communication patterns. Figure 6 illustrates the high-level system architecture. By choosing JavaScript for implementing all core components and with a re-configurable stack of data access layers (*database driver, cache, remote access*), WebGME allows for different deployment scenarios tuned for scalable collaboration, offline work and/or for high-performance and high-bandwidth model interpretation. In the most common deployment model, a relatively thin server-side component—running as a Node.JS process—acts as a communication bridge between the model storage (MongoDB) and multiple browser-based clients. Beyond providing serialized read/write data access, it sends broadcast messages to all connected clients after each update—using WebSockets as the transport protocol for both tasks. In a high-performance scenario, a single client can be deployed directly and exclusively on top of the database interface in a Node.JS container on the server. Intermediate layers in the data access stack also enable intentional or accidental (e.g., due to network problems) offline work.

The most critical components (Core and Client) are deployed on top of the data access stack. These layers assemble and maintain consistent in-memory snapshots of the model hierarchy and provide Model API, the common basic interface for all higher level components. This interface is directly used by the visualization stack (Section 6) and by high-performance plug-ins implemented in JavaScript. These plug-ins can target specific domains for a wide spectrum

of automated tasks, such as model analysis, simulation, verification, code and report generation, model transformation, and design space exploration. Also, the architecture can be extended by domain-independent plug-ins for providing and integrating new generic tools.
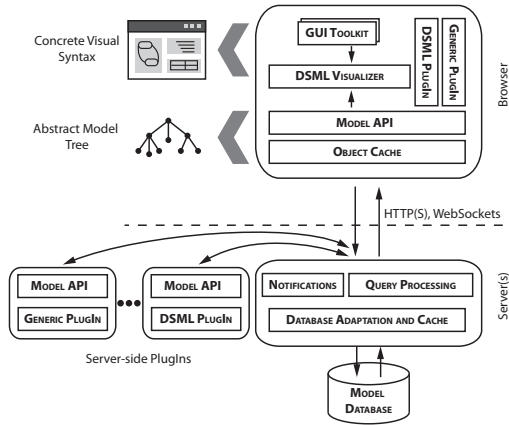

Fig. 6. High-level system architecture

Note that developing new plug-ins against the Model API requires the components to be developed in JavaScript. To overcome this limitation and provide data access for the models for the widest developer community, a REST web services API is also provided by our server. This interface enables language and technology independent access to the same data-model that is available via the Model API. The trade-off of the REST interface is slower access (significantly higher latency and serialization/de-serialization overhead). Note that both *native* JavaScript and REST components can be deployed on client and server side. Although server-side REST components still use the same general infrastructure as those on the client side, these benefit from the physical proximity to the server.

Our experience shows that higher-level domain-specific APIs can dramatically boost the productivity of domain developers. These auto-generated APIs 'speak the language' of the domains, and can significantly reduce the time and effort needed to develop new plug-ins. The definition and generation of these APIs—based on the meta information in the model—is part of our future work.

Another recurring pattern we identified in a large sample of existing third-party tools uses a semi-offline processing approach for the model hierarchy. These tools initially traverse the entire model and store it in an intermediate format for performance and convenience reasons. These types of components can leverage the model export/import facility on our server, which supports full and partial (de-)serialization of the models in JSON format. This capability is integrated with the REST service interface.

Beyond acting as a data bridge, sending broadcast notifications and providing the REST API, the current server-side component is responsible for bootstrapping the browser application with static content and for implementing authentication and authorization tasks. The authentication infrastructure is based on the Passport framework [20], which supports a comprehensive set of strategies and protocols from simple username/password pairs to OpenID and OAuth providers (e.g. Facebook, Google).

6 Visualization

Traditional MIC model editors enable an extremely fast early prototyping phase by providing default visualization and user controls for each element of the DSML language being developed. These editors provide built-in user interface logic for each fundamental (meta-meta) concept. Although, parts of the built-in behavior is customizable—with bitmap and vector images or by providing custom rendering and event handling code plug-ins—many environments make it difficult to implement a fundamentally different user interface experience. Deviating from the default mapping of abstract model elements to visual primitives or implementing radically different visual behavior is becoming exponentially more difficult. Thus, after the early prototyping phase, many DSML designers struggle with developing a refined domain-specific and user-centric editor experience. Typically these custom views include tabular data representation, textual formats (source code, XML, JSON), form-based user interfaces, or complex data visualization with precisely controlled layout and rendering. In all these cases, customization needs to reach beyond the rendering of individual model elements and has to control the overall mapping of abstract data model elements to visual primitives and UI actions to operations on the data.

Hence, WebGME provides a visualization toolkit as opposed to the customizable model editor approach of GME. The key difference is that with the toolkit, the DSML designer has more control over the visualization aspects of the language. Elements of the toolkit include layout managers, line and area rendering primitives, and in-place text editors. These elements handle the rendering and UI interaction tasks. The DSML designer will use the graphical building blocks and provide the mapping between the model database (Model) and the toolkit elements (Views and Controllers).

Visually complex and large models pose scalability challenges for the UI much sooner than for the underlying model database. Graphical model editors typically address the visual scalability problem by either providing flat 'model canvases,' with which the model can be partitioned to multiple sheets with some shared entities, or by using hierarchical decomposition. Examples for the former approach are UML class diagrams, circuit schematics, and Petri nets, while the second approach is more prevalent in modeling signal flow graphs, hierarchical state machines, and design spaces. Both methods have limitations: the model builder has to reason about a 'mentally stitched' model or constantly navigate into a deep model hierarchy while taking extra effort to model cross-cutting relationships across distant model elements. Filtered views (i.e., aspects)—showing only a subset of the elements of the model—is a simple but insufficient feature towards providing a scalable user interface.

Both GME and WebGME supports model canvases, hierarchy and filtered views (aspects), but WebGME takes a significant step beyond these standard techniques with the introduction of crosscuts. Crosscuts decouple visualization from the model hierarchy and provide completely user-defined orthogonal views of the models. The default visualization in crosscuts focus on associations by displaying any existing relationship between members of the crosscut with lines,

as well as provide the means to modify them or create news ones. However, any other type of visualization can be implemented as required by the given domain. Elements of the crosscut can be selected by direct user actions or soon user-defined queries can gather model objects and populate the crosscut. We believe that by providing powerful query-based cross-cutting views one can build truly scalable domain specific interfaces.

7 Multi-Paradigm Modeling Support

The best way to show how WebGME supports multi-paradigm modeling is through an example. Consider a simple DSML, a hierarchical signal flow language similar to Simulink called *SignalFlow*. Figure 7 shows WebGME with the SignalFlow DSML inside a Chrome browser. The metamodel is shown on top. The main concepts are *Compound* and *Primitive*, signal flow operators that are the composite and leaf nodes of the model hierarchy, respectively; *Input* and *Output* ports that provide the signal interface for the operators; and *Flow* that are the connections between ports. Parameters (with *DataType* and *Size* attributes) provide configuration parameters to operators.
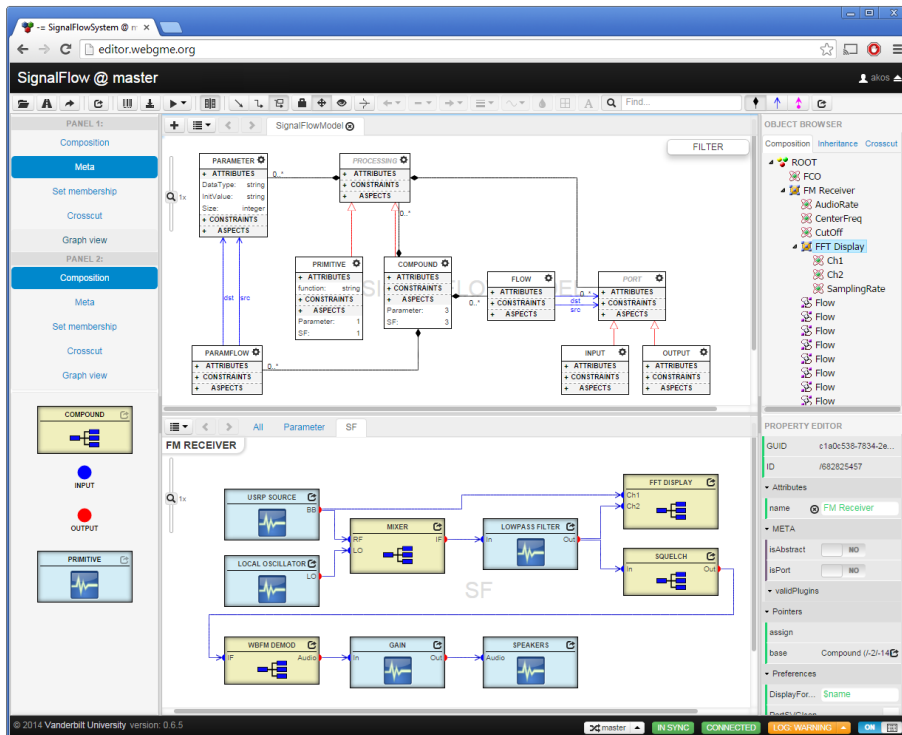

Fig. 7. Example Signal Flow DSML in WebGME

The left side of the screen shows the panel control buttons and below them the Part Browser that displays the models that can be instantiated inside the model loaded in the current panel as defined by the metamodel. Dragging and dropping a model from the Part Browser creates a new instance of the given model. The top right side of the user interface shows the Object Browser that shows the composition hierarchy of the project starting at Root. Below is the Property Editor where attributes, preferences and other properties of the currently selected model can be edited.

The user interface of the tool provides quite sophisticated features especially considering the browser-based execution environment. Drag and drop, context menus, search, autorouting, Bezier curves, and various visualizers in addition to what is shown in the figure are all provided. However, a detailed description of the user interface is beyond the scope of this paper.

Suppose we want to create a new DSML, that supports Signal Flow models, but also allows the modeling of simple multiprocessor hardware *and* the assignment of signal flow components to processors. Similarly to GME, WebGME also supports libraries. We can select any model in a project and export it as a library. This takes the composition tree rooted at the given model and generates
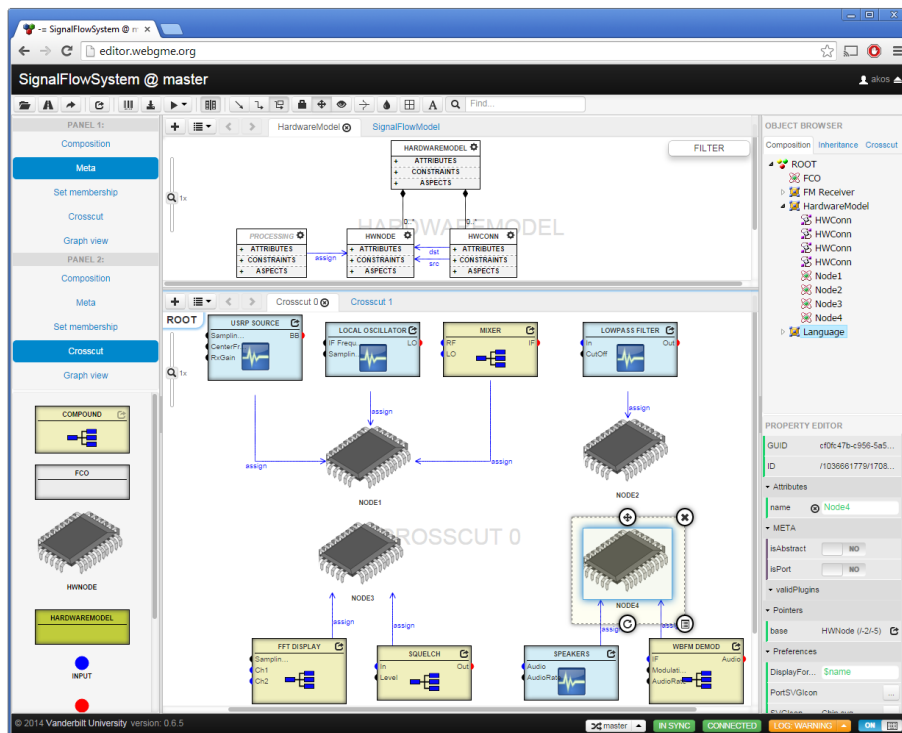

Fig. 8. Processor Assignment for Signal Flow Models

a JSON file from it. In our example, the *SignalFlow* metamodels were stored not directly in *Root*, but in a model contained by *Root* called *Language*. We exported it and then imported it into a new project called *SignalFlowSystem*. Imported libraries are read-only models, but they can be derived and associations can point to any part of them. In this new project, we created a simple computer hardware DSML. Finally, the only thing missing from the metamodels is the specification of the assignment. This is the concept that ties together the two paradigms. This can be done simply by dragging in the *Processing* model (the signal flow base component) in the Hardware metamodel and adding a new pointer to *Processing* called *assign* that can point to *HWNode*s (see top of Figure 8). Note that none of the composition rules were changed, that is, signal flow models cannot contain hardware models and vice versa. The only relation between the two sub-languages is the *assign* pointer, but it cannot be visualized in any of the models nicely. But we can create a crosscut that can contain *Processing* and *HWNode* models and it will show the assignment pointers and also enables the user to edit them as shown in the bottom part of Figure 8.

8 Related Work

There have been promising approaches to (i) collaborative modeling, (ii) web-based modeling environment, and (iii) model versioning. This section reviews the results closest to our solutions.

Collaborative modeling is used in specific domains such as mechanical engineering [21], automotive industry [22], and UML [23]. A collaborative DSML definition process is presented in [24, 25] which could be supported by WebGME. SLIM [26] is a prototype of a collaborative environment executed in a web browser. The Connected Data Objects (CDO) [27] is a model repository and a run-time persistence framework for EMF. It supports locking, offline scenarios, various persistence backends, such as Hibernate, and pluggable fail-over adapters to multiple repositories. As a part of CDO, the Dawn framework supports collaboration on the user interface level with functions such as locking, conflict detection and resolution. It is integrated with multiple graphical editors. In contrast with our approach, CDO supports model integration on the model level, and not on that of the edit operations. If two transactions are trying to modify the same object, CDO signals a conflict for the second transaction as opposed to our approach, where this creates a new branch automatically. CAMEL [28] is also an eclipse plugin that supports collaborative interaction via modeling, drawing, chatting, posterboards, whiteboards, and it is capable of replaying online meetings. Its focus is on collaborative communications rather than versioning and collaborative use of domain-specific languages.

AToMPM [29], a web-based metamodeling and transformation tool for Multi-Paradigm Modeling, is the closest to our work. While many of the authors' architectural decisions are similar to ours, versioned repository, the fusion of metamodeling and prototypical inheritance and crosscuts are the biggest differentiating factors.

A summary of model versioning can be found in [30]. A formal approach is contributed in [31]. [32] describes an extension of AMOR [33] to facilitate model-based merging. [34] describes precise methods for parallel dependent graph manipulations when insertion has priority over deletion. Our philosophy is rather to avoid situations where merge is needed – the fine grained commit cycles serve exactly this purpose. However, especially after offline work, these solutions can extend our approach when two branches need to be merged.

Web technologies have advanced to the point where it is feasible to build user-friendly and visually appealing user interfaces with good performance inside a web browser. Lucid Charts [35] and CircuitLab [36] are excellent examples of what is possible. Some of these even support online collaboration. On the other hand, these tools 1) employ relatively simple, typically flat data models and 2) are very specific to their respective domains. They do not solve the challenges associated with evolutionary language design, configurability, branching, and extensibility.

9   Conclusions

WebGME has a number of novel features and several advantages over desktop-based (meta)modeling tools. The browser-based client is platform-independent and does away with installation and software update issues. The data model and software architecture were designed from the ground up to provide scalability, seamless collaborative modeling and powerful model versioning. The prototypical inheritance and crosscuts are probably the two most unique features of the WebGME meta-metamodel providing DSML and model complexity management.

WebGME is still under development. The two most significant missing pieces are merge support for branches and a constraint manager. The merge operation is critical to enable other modes of collaboration beyond immediate concurrent updates. Constraints also play an important role in DSMLs. GME has an OCL-based constraint manager which proved very useful for people who were willing to learn OCL, but were ignored by most users. What constraint language WebGME will ultimately utilize is still up for debate.

WebGME supports multi-paradigm modeling using inheritance, libraries and crosscuts. Multiple inheritance would be really powerful for metamodeling in general and multi-paradigm modeling in particular. Merge may enable multiple inheritance support, however, it is a really challenging concept because of the complex interplay between composition and inheritance.

9.1   Acknowledgement

References

1. Sztipanovits, J., Karsai, G.: Model-integrated computing. Computer **30**(4) (1997) 110–111
2. Lédeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. Computer **34**(11) (2001)
3. Long, E., Misra, A., Sztipanovits, J.: Increasing productivity at saturn. Computer **31**(8) (1998) 35–43
4. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE **91**(1) (2003) 145–164
5. Mathe, J.L., Ledeczi, A., Nadas, A., Sztipanovits, J., Martin, J.B., Weavind, L.M., Miller, A., Miller, P., Maron, D.J.: A model-integrated, guideline-driven, clinical decision-support system. Software, IEEE **26**(4) (2009) 54–61
6. Lattmann, Z., Nagel, A., Scott, J., Smyth, K., Porter, J., Neema, S., Bapty, T., Sztipanovits, J., Ceisel, J., Mavris, D., et al.: Towards automated evaluation of vehicle dynamics in system-level designs. In: ASME 2012 Computers and Information in Engineering Conference, ASME (2012) 1131–1141
7. Levendovszky, T., Balasubramanian, D., Coglio, A., Dubey, A., Otte, W., Karsai, G., Gokhale, A., Nyako, S., Kumar, P., Emfinger, W.: Drems: A model-driven distributed secure information architecture platform for managed embedded systems. IEEE Software (2014) 1
8. Bagheri, H., Sullivan, K.: Monarch: model-based development of software architectures. Model Driven Engineering Languages and Systems (2010) 376–390
9. Bézivin, J., Brunette, C., Chevrel, R., Jouault, F., Kurtev, I.: Bridging the generic modeling environment (GME) and the eclipse modeling framework (EMF). In: Proceedings of the Best Practices for Model Driven Software Development at OOPSLA. Volume 5., Citeseer (2005)
10. Bunus, P.: A simulation and decision framework for selection of numerical solvers in. In: Proceedings of the 39th annual Symposium on Simulation. ANSS '06, Washington, DC, USA, IEEE Computer Society (2006) 178–187
11. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: An industrial experience report. In: Generative Programming and Component Engineering, Springer (2002) 156–172
12. Stankovic, J., Zhu, R., Poornalingam, R., Lu, C., Yu, Z., Humphrey, M., Ellis, B.: Vest: an aspect-based composition tool for real-time systems. In: Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE. (May) 58–69
13. Thramboulidis, K., Perdikis, D., Kantas, S.: Model driven development of distributed control applications. The International Journal of Advanced Manufacturing Technology **33**(3) (2007) 233–242
14. Mohanty, S., Prasanna, V., Neema, S., Davis, J.: Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. ACM SIGPLAN Notices **37**(7) (2002) 18–27
15. Hemingway, G., Neema, H., Nine, H., Sztipanovits, J., Karsai, G.: Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach. Simulation **88**(2) (2012) 217–232
16. Ledeczi, A., Balogh, G., Molnar, Z., Volgyesi, P., Maroti, M.: Model integrated computing in the large. In: Aerospace Conference, 2005 IEEE, IEEE (2005) 1–8
17. Katz, R.H.: Toward a unified framework for version modeling in engineering databases. ACM Comput. Surv. **22**(4) (December 1990) 375–409

18. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. IJWIS **5**(3) (2009) 271–304
19. : GIT Homepage. `http://git-scm.com` Cited 2013 Mar 14.
20. : Passport, authentication middleware. `http://passportjs.org` Cited 2014 Mar 17.
21. Li, M., Wang, C.C., Gao, S.: Real-time collaborative design with heterogeneous cad systems based on neutral modeling commands. Journal of Computing and Information Science in Engineering **7**(2) (2007) 113–125
22. Kong, S., Noh, S., Han, Y.G., Kim, G., Lee, K.: Internet-based collaboration system: Press-die design process for automobile manufacturer. The International Journal of Advanced Manufacturing Technology **20**(9) (2002) 701–708
23. Boger, M., Graham, E., Köster, M.: Poseidon for uml. Pode ser encontrado em http://gentleware.com/fileadmin/media/ archives/userguides/poseidon_users_guide/book1.html (2000)
24. Izquierdo, J.L.C., Cabot, J.: Enabling the collaborative definition of dsmls. In: Advanced Information Systems Engineering, Springer (2013) 272–287
25. Izquierdo, J.L.C., Cabot, J., López-Fernández, J.J., Cuadrado, J.S., Guerra, E., de Lara, J.: Engaging end-users in the collaborative development of domain-specific modelling languages. In: Cooperative Design, Visualization, and Engineering. Springer (2013) 101–110
26. Thum, C., Schwind, M., Schader, M.: Slima lightweight environment for synchronous collaborative modeling. In: Model Driven Engineering Languages and Systems. Springer (2009) 137–151
27. Stepper, E.: Connected data objects (cdo). Website http://www. eclipse. org/cdo/documentation/index. php, seen November (2012)
28. Cataldo, M., Shelton, C., Choi, Y., Huang, Y.Y., Ramesh, V., Saini, D., Wang, L.Y.: Camel: A tool for collaborative distributed software design. In: Global Software Engineering, 2009. ICGSE 2009. Fourth IEEE International Conference on, IEEE (2009) 83–92
29. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Ergin, H.: Atompm: A web-based modeling environment, MODELS (2003)
30. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. International Journal of Web Information Systems **5**(3) (2009) 271–304
31. Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: Algebraic foundations and the tile notation. In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models. (2009) 7–12
32. Brosch, P., Seidl, M., Wieland, K., Wimmer, M., Langer, P.: We can work it out: Collaborative conflict resolution in model versioning. In: ECSCW 2009. Springer (2009) 207–214
33. Altmanninger, K., Kappel, G., Kusel, A., Retschitzegger, W., Seidl, M., Schwinger, W., Wimmer, M.: Amor–towards adaptable model versioning. In: 1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS. Volume 8. (2008) 4–50
34. Ehrig, H., Ermel, C., Taentzer, G.: A formal resolution strategy for operation-based conflicts in model versioning using graph modifications. Springer (2011)
35. : Lucidchart. `http://www.lucidchart.com` Cited 2014 Mar 17.
36. : CircuitLab. `https://www.circuitlab.com` Cited 2014 Mar 17.