A Model Repository for Collaborative Modeling with the Jazz Development Platform

Christian Bartelt  
University of Clausthal  
christian.bartelt@tu-clausthal.de

Georg Molter  
Zühlke Engineering GmbH  
Georg.Molter@zuehlke.com

Tim Schumann  
University of Kaiserslautern  
schumann.tim@googlemail.com

Abstract

Today’s software development projects have a huge demand for a comprehensive and integrated support in the team-based development of models. Since they typically rely on file-based version control systems such as CVS and assume that working with models does not differ from working with code, current solutions solve only one part of the problem. This paper addresses this gap between what is needed in practice and what is provided by existing tools. Based on the collaboration platform Jazz, a model repository prototype is developed that provides adequate versioning support and leverages the collaboration mechanisms provided by Jazz for concurrent model development. Especially the integration of concurrently developed models turned out to be a complex and collaborative task. The results of this investigation are integrated into the prototype yielding a model repository that supports team-based development to a considerable extent.

1. Introduction

Throughout the history of software engineering, the problems of the ever increasing complexity of software systems, intensified utilization in safety-critical domains, and the need for shorter time-to-market significantly influenced the methods for the development of software systems. Raising the level of abstraction and providing environments that allow large teams to collaboratively work together proved to be successful approaches to address these issues.

During the last years, the level of abstraction has been moving from source code to models that describe software systems on a higher level. As stated by Lin et al. [13], models are elevated to first class artifacts. This is reflected in many approaches currently used in industrial practice. The subjects of Model Driven Engineering (MDE) are well-defined models. But similar to source code, models are subject to a continuing evolution and therefore need to be managed by software configuration management (SCM) systems.

The management of evolving models is an ambitious challenge, because often large models are developed concurrently by modelers in large teams that may be geographically distributed. There is a huge demand for both effective and intensive communication in order to get growing software complexity, team sizes, and geographical distribution of the developers under control. Close collaboration plays a key role in today’s and tomorrow’s software development and has to be well supported by tools. Currently, there are several products as the Microsoft Team Foundation Server [22] or the IBM Jazz Platform [9] that provide collaboration platforms for software development to meet this emerging demand. However, these products are focused on code-centric development and currently do not provide support for the efficient management of models.

Besides, it takes more effort to support the collaborative development of models by an integrated collaborative platform. To achieve this, the following subsections explain the shortcomings of the current scientific approaches, the tools/technologies of the industrial practice (Subsection 1.1) and three research objectives (Subsection 1.2) were derive which will be discussed further in this paper. In Section 2 and 3 a solution for the identified problems is proposed and the implementation of the realized software prototype is outlined.

Figure 1. Overview of technologies/tools/terms focused on collaborative modeling
1.1. Shortcomings of current approaches

This subsection provides an overview of the state-of-the-art and picked up its shortcomings (italic font) related to the paper’s aims. Afterwards the problem definition and the objectives of this paper are formulated. Figure 1 facilitates the orientation related to the relevant terms of the three research areas collaboration- and SCM systems and model driven engineering (MDE) which will be referenced in the following paragraphs. Tools and technologies of these areas are stated in the lower line of each box. The arrows between the boxes symbol the dependencies of the provided services.

1.1.1. Established, text-based version control. The most popular SCM systems as RCS [23], CVS [6] or Subversion [21] operate under a linear, file-based paradigm that is purely textual. However, since models are not sequential text lines, but could be represented either as trees or graphs, these SCM systems cannot use the formal structure of models to achieve correct compare or merge results concerning syntax and semantics [10, 13].

1.1.2. Approaches for concurrent model development. As mentioned in [13], one current approach for concurrent development of models is the persisting of model data as structured text files (typically XML) and the management of these files using SCM systems as CVS. Examples for such systems are 4everedit [14] and CASE tools as the IBM Rational Software Architect [12]. The Rational Software Architect (RSA) is a tool for model-driven development with the UML.

Another popular CASE tool with teamwork support is MagicDraw [17]. To enable multiple developers to work on the same UML model, NoMagic provides the Teamwork Server product. “The UML model is stored in the Teamwork Server repository and every developer may lock a part of the model and work on that part individually [principle of pessimistic locking: lock-modify-unlock]. Later changes may be committed to the server and shared with the team.” However, it is doubtful if this pessimistic locking approach is feasible in development of large systems respectively large models [14].

Furthermore the presented tools which support the optimistic locking principle (copy-modify-merge) rely on traditional, file-based SCM-systems. Moreover, all tools assume the work flow “check out” → “modify” → “check in and solve conflicts on-the-fly” performed by a single developer. As outlined in Section 2, this workflow is unsuitable for the collaborative development of well-defined models.

1.1.3. Merging and comparing of higher-level software artifacts. Commercially available tools for software merging and comparison are usually based on textual techniques. Mens gives in [14] another inherent limitation of textual merging: “[...] it [the textual merging] can only detect very basic conflicts. It does not take the specific semantics of software artifacts into account because everything is treated as an ordinary piece of text.” In the research community, merge and compare mechanisms for models have been recognized as a key challenge [13].

In [11], Kelter et al. present a difference algorithm for UML-models realized in the SiDiff project. This algorithm operates on the graph structure of UML models.

Alanen et al. describe in [2] three generic algorithms for comparing (diff), merging (merge), and joining (union) models. These algorithms effectively provide an optimistic three-way merge, however its merge semantics are quite complex.

While the so far presented approaches directly operate on the model structure defined by its meta-model, in [3] a different approach is followed. By regarding models in a meta-model independent, modularized representation, the algebraic merge approach presented in [4] can be performed with only two operations: Add and Delete. This allows that the algebraic description of the merge semantics is clearer. However an implementation of this approach can include a modify or other primitive operations. The merge process operates in two stages. The first stage produces a merged model that might contain inconsistencies. These inconsistencies are identified and resolved in the second stage.

1.1.4. Current approaches to model repositories with versioning. During the last years, some research has been done towards model repositories with versioning support. With Odyssey-VCS, Oliviera et al. [18] implemented a version control system for UML to provide a configuration management support.

Marcus Alanen presented a model repository supporting version control [1]. This model repository is based on the Meta Object Facility (MOF) [15].

Both of these approaches do not focus on supporting team collaboration. Moreover, they establish separate systems which make it hard to leverage tools and information that are available in the team’s development environment.

Another approach is the CoObRA extension of the UML case tool Fujaba [19] which manages the concurrent development of UML models but it is not
meta-model independent and supports only a rudimentary consistency management.

1.2. Problem definition and objectives

The presented state-of-the-art approaches display several shortcomings about collaborative modeling aspects. Hence it is possible to derive the problem definition as two shortcomings:

- Currently available CASE tools provide just rudimentary support for the collaborative development of models. These systems are not based on sound foundations for versioning of models and have limited capabilities with respect to fine-grained version histories or collaboration.

- There are several promising approaches concerning comparing, merging, and consistency management of models, including meta-model independent approaches. But there is not an integrated concurrent versioning system for models such as Subversion for source code.

As mentioned in [5], concurrently working with models differs from concurrently working with textual documents. A consequence of the raised level of abstraction is the increased complexity per unit. This makes the task of merging concurrently developed model versions more complex. This has at least two drawbacks for model-centric SCM systems: Especially the resolution of conflicts or inconsistencies usually cannot be achieved by a single developer who does, in general, not have enough knowledge about other modifications that interfere with his work. On the other hand the change committing policy “first come, first served” that is usually practiced in code-centric development, is not feasible in the context of model-centric development. The reason is again the increased complexity of conflict respectively inconsistency resolution.

Characteristics as meta-model independency are crucial, since widely used meta-models – such as UML – are rapidly evolving, and on the other hand, domain specific languages (DSL) gain importance. As mentioned in [5], the support for efficient collaborative development is essential.

According to the described problem definition this paper explains the concepts which address three identified main challenges and describes the prototypical realization of an integrated, collaborative modeling platform based on Jazz. The following challenges state an outline about the paper:

(a) Concurrent modeling process.
- The popular principle “first come, first serve” during integration of concurrent code development branches is inadequate for model-centric development. Model-centric development requires rather a collaborative integration of branches as explained in Section 2.

(b) Meta-model independent model repository.
- A model repository is required that provides versioning mechanisms for models as currently provided for textual documents by most SCM systems. Section 3 describes one solution for this.

(c) Integration of collaboration and modeling.
- To support effective collaboration, the presented prototype extends the Jazz team collaboration platform with concurrent modeling support. At the end of Section 2 and 3 this integration is shown.

2. Collaborative modeling process

This section explains the weak points of current SCM concepts when applied to model-centric development and shows, how the process of collaborative model development can be improved to meet the new requirements. The requirements and the evaluation test cases were amassed in a project case study of two projects (WEIT, V-Bench) whose care about the development and maintenance of the V-Modell® XT. The V-Modell® XT [24] is the German standard model for system development and lifecycle processes conducted within engineering projects of federal administration and defense. It is publicly available and many companies have successfully adopted it to their needs. More than 35 developers at 8 distributed team locations are involved during the development and maintenance of this mid-size model with approx. 5000 first class entities.

2.1. Basic SCM interactions in concurrent modeling

For a better understanding of the interactions that are necessary for the basic, concurrent development of models, a detailed look at the typical and commonly arising scenarios is considered: In sequential development (a) one developer works exclusively on a model in a repository branch. In simultaneous development on the same model (b) multiple developers work on the same model in the same repository branch. This requires either a pessimistic locking mechanism (mentioned in Subsection 1.1) or synchronous collaboration with intensive communication in order to avoid conflicting or overriding changes and inconsistencies. In Concurrent development (c) multiple developers enhance the same model in multiple branches. Each developer (or sub team) initializes a repository branch and makes modifications to this model according to scenario (a) or
When all developers have finished their work, the resulting models have to be integrated (merged) into one valid and consistent model. This scenario premises asynchronous collaboration which is focused in this paper. While the management of variants (d) a product has a core model that is basically the same in all variants. Separate branches implement single aspects of the supported variants.

Especially the scenario of concurrent development (c) which arises whenever multiple versions have to be merged into one model is focused in this paper. On basis of the scenarios two primary use cases can be derived: IntegrateModels and ResolveConflict/Inconsistency. An extensive exemplification of these use cases can be found in [20]. The use case IntegrateModels represents the integration of different concurrent developed model versions. Since the use case IntegrateModels might heavily differ from merging in code-centric development, a more detailed description of this use case is given in Figure 2. Note, that at least one developer of each model version for integration is necessary to solve inconsistencies collaboratively (line Actors of Figure 2). A further rule is that the model versions for the integration have to be consistent (line “Preconditions” of Figure 2).

Figure 2. Use case “Integrate models”

The use case ResolveConflict/Inconsistency describes the actions to resolve the corresponding conflicts and model insufficiencies, respectively. The resolution of inconsistencies usually demands further model modifications for yielding a consistent model.

Although these conflicts and inconsistencies also arise in code-centric development, they play a special role in model-centric development. On the one hand, elaborated tools which support inconsistency detection in code-centric development, such as unit tests or code analyzers, are not yet widely available for model-centric development. On the other hand, supporting an user in resolving conflicts and inconsistencies is still a challenge in model-centric development, whereas feasible solutions are available for code-centric development.

The support of the presented use cases for the efficient, collaborative development of models is shown which constitutes the building block Process Support. As illustrated by Figure 3, this building block assumes an underlying model repository. Two components were implemented as Eclipse plug-ins – Process Support for Collaborative Model Evolution and Model Repository. The generic Repository and the SCM system are provided by the Jazz Platform and the Generic Modeling Infrastructure is provided by the EMF framework (cf. Figure 3, Figure 1).

Figure 3. Collaborative modeling platform

2.2. Requirements to the process of model evolution

In current SCM systems, the concept of branches serves for two purposes, namely the management of concurrently developed item versions (Version Management) and supporting the organization of work (Task Management). In this subsection, these two aspects of the concept of SCM branches are separated and a more advanced concept of task management that addresses the challenges raised by the concurrent development of models is developed. The aspect of task management is focused in this section, because – as explained in the following – the concurrent development of models requires a better organization of work. The tasks (or work units) that are subject to this task management are denoted in the following as modification tasks.

This section is structured as follows. Subsection 2.2.1 shows the challenges that come along with conflict solving in the concurrent development of models and their implications for task management. Section 2.2.2 presents a proposal of process rules for task management in model-centric development and its connections to SCM concepts.

2.2.1. The complexity of conflict resolving approaches. Since the versioned items in model-centric development are more abstract than in code-centric development and often also directly represent ideas, the amount of (possibly implicit) information per
versioned item is much higher than in code-centric development. Moreover, opposed to the development of source code, there are no elaborated, lightweight test mechanisms widely available to ensure that the modeled system behaves as expected. This is especially true after concurrent modifications have been applied. Consequently, the process of conflict solving in model-centric development is an error-prone task that requires high efforts. Based upon these observations, the following rules of thumb should be observed. Integrate collaboratively: In order to combine concurrently developed ideas, all involved parties have to participate in the merging process. Avoid conflicts: Since the resolution of conflicts is pricey, conflicts should be avoided whenever possible. Resolve conflicts as soon as possible: Due to the raised complexity of conflicts, they should be resolved within a short time after the creation. No “First come, first served”: Modeling is a creative process and it is not an implementation of a specification. The change committing policy “first come, first served” that is usually practiced in code-centric development, where every single developer is allowed to commit his changes, is not feasible in the context of model-centric development. This is supported by the following two arguments. For one thing, following the policy “First come, first served”, a committed change has to be applied in every modification task that is performed in parallel which results in a high number of merging actions and potential conflicts. For another thing, the author of a committed change can – in case of a conflict with other developer’s changes – not be sure that his changes were incorporated correctly, because of his unawareness of these conflicts and the absence of elaborated test mechanisms.

The primary statement resulting from these conclusions is that merging has to be regarded as a separate collaborative task. Therefore, the process of merging two (or possibly more) modification tasks will be encapsulated in an integration task that is performed by developers of each modification task and ensures that the combined model is consistent and correct. This collaborative work can be done in several ways. A lightweight possibility is to have one person (the integrator) performing the integration and requiring the other developers to review and agree the integration results. This proceeding requires effective mechanisms of communication to support the integrator in decisions that need assistance by other developers.

Another possibly quite expensive option is to require all developers to perform the integration together in a session or a meeting. This might be necessary in case that many contradicting or correlating changes have to be integrated or the ideas and decisions of the single modification tasks have to be explained to the whole team.

**Figure 4. Tasks of collaborative modeling**

2.2.2. Process Premises and Rules. As stated before, working with higher-level models requires an adaptation of the development process for organizing the different modification tasks. The main focus of the proposed development process is the support of parallel and collaborative development in certain phases and environments. These are those that require isolated development and separately performed integration. The following terms describe some elements identified in the preceding subsection and are illustrated by Figure 4.

Modification tasks and integration tasks are both referred to as development tasks. Every change that is applied to a model version has to be performed within a Development Task.

All modifications that have to be applied to the product get organized in Modification Tasks. The modifications within a modification task share one common criterion – the modification task’s rationale that gives the justification or motivation. The task of merging the results of different development tasks into one combined, consistent model is performed in an Integration Task. Therefore the integration tasks encapsulate model changes for receiving the meta-model consistency of the merge result.

The proposed process is decoupled from the underlying SCM-system. The only things that are assumed to be provided by the underlying SCM system are explained in the following.

- A mechanism that makes specific model versions from one development line is available to other developers. This mechanism can be accessed by the developers to get the latest model version as a basis for further developments or to make new model versions available to all other developers. In this paper, such a mechanism is referred to as trunk.

- Branching concepts that allow the isolated development of modification tasks and integration tasks.
Figure 5 shows the development tasks and the trunk of a small example. This figure depicts the modification tasks \( \{MT_1, MT_2, MT_3, MT_4, MT_5, MT_6\} \), the integration tasks \( \{IT_1, IT_2, IT_3\} \) and the model versions \( \{v_0.1, v_0.2, v_0.3\} \) that were published on the trunk. The arrows show the input and output relations of the development tasks.

The remainder of this subsection gives a description of the process premises and rules. A precise algebraic, formalization of these premises and rules can be found in [20]. Therefore, a definition of concurrently developed tasks is necessary, which is expressed by the property of being parallelizable. Two or more development tasks are parallelizable if they are based on the same model version on the trunk. They do not have to base directly on this model version, e.g. they can also base on the results of other development tasks, however, all of them must have the same latest ancestor model version on the trunk.

Since the property of parallelizable modification tasks is that they can be performed at the same time, the model versions on the trunk define time slots in which parallel modification tasks can be started. This assumes that newly created modification tasks (possibly indirectly) base on the latest version on the trunk. So, considering the example of Figure 5, \( MT_2 \) has to be created before the model version \( v_{0.2} \) was presented on the trunk. This assumption is reasonable, since all model versions developed in separate modification tasks of one development line finally have to be combined into one model.

Figure 6 shows a screenshot of the extended Jazz platform which implements the proposed collaborative task management support.

### 3. Meta-model independent model repository based on the Jazz platform

The Jazz platform provides mechanisms for managing the concurrent development of code. This section outlines the design of the Jazz-based model repository (building block Model Repository in Figure 3) and introduces the supported model representations (building blocks Model Representation and Generic Modeling Infrastructure). The following questions will be addressed in this section:

(a) What are the requirements to the prototyped model repository?

(b) Which internal representation of models which supports the requirements of the model repository should be used for the system?

(c) How can the users of the system access or modify models that are managed by this system?

#### 3.1 Requirements to the model repository

Within this subsection, the requirements to the model repository prototype are given as demanded by the upper question (a). The description of these requirements also includes references to the affected use cases of model-centric SCM systems in general that were presented in Subsection 1.2.

**Req. Meta-Model Independency** As highlighted in Section 1, the model repository has to manage models independently of the underlying meta-model. To provide access to the meta-model’s information, the meta-model is considered to be specified using EMF Ecore (implementation of EMOF 2.0) [8].

**Req. Merge Algorithm** Since the models managed by the model repository are subject to concurrent evolution, merge algorithms have to be able to process them. Relevant use case: UC IntegrateModels.
Req. External Modifications Users are assumed to make model modifications with an external editor that is separated from the model repository.

Req. Change Detection To detect modifications that were applied to the model externally, mechanisms for comparing models and detecting changes have to be supported.

3.2 A module concept for meta-models

This subsection covers question (b) raised at the beginning of this section. It introduces the module representation [3] as an internal model representation.

As demanded by the requirement meta-model independency of Subsection 3.1, no assumptions about the meta-model underlying the models can be made. To understand the role of a model’s meta-model, the abstract syntax of a model is regarded as a graph, whose nodes are model elements (of a specific type defined by the meta-model) and edges are references between model elements. The meta-model’s constraints among the model elements are separately defined and not contained in the model itself.

The following properties are fundamental for storing and merging models meta-model independency: (i.) Models consist of model elements, (ii.) Model elements are related to each other, (iii.) Model elements are able to contain value specifications, and (iv.) Model elements are associated with one type of a meta-model.

Current and most popular merge algorithms work on sets of information items whereby different versions of each information item are merged into one version.

In [3, 4], an optimistic, set based, three-way merge algorithm is presented that provides the formal basis of merging for this model repository prototype. The aim is to define an internal model representation such that models in this representation can be processed by that algorithm. However, this algorithm assumes that all model elements are of the same type. As explained above, models are linked structures that have different node types (property (iv)). To be able to apply the merge algorithm which was pre-published in [3, 4], the module representation as introduced in [3] was chosen as the internal representation. This module concept is able to represent model structures whereas only one item type is needed which is called InformationItem. Figure 7 depicts an overview of the model representation module and is explained in the following.

Each element of type InformationItem is either a ModelingItem, an ItemProperty, or a TypeInformationItem. Each instance of ModelingItem represents one instance of a type from the specific meta-model. If the type is a primitive data type then the attribute symbol holds the value specification.

Figure 7. Meta-model module representation

Each ItemProperty object represents an instantiated property of the associated modeling item, a so called slot. An item property has to link the meta-model property. The entity TypeInformationItem is a placeholder for an assignment between a modeling item and its type in the domain specific meta-model (as the instanceof relation).

The instances of Coupling play a special role. These objects guarantee that information items cannot directly reference other information items. This is important to avoid open reference ends after the application of a delete operation on a referenced item. A reference between information items is represented only if two information items link the same coupling object. The coupling objects are not information items and therefore neither transparent nor directly changeable. They are managed internally by the system. After this description of the entities shown in Figure 7 their relations can be explained:

- Modeling items may own properties. This is represented by the two references propertyCoupling and modelingItemCoupling.
- Modeling items can reference other model elements or be referenced by them. This is illustrated by the referenceTargetCoupling and the referenceSourceCoupling references.
- Modeling items are associated with a type of the specific meta-model.

An important property of the module concept is that an information item cannot be modified after it has been created. This allows expressing model modifications solely with operations that add or remove elements from the module representation, which significantly simplifies the merge algorithm.
3.3 Handling models meta-model independently within Jazz

The topic of this subsection is to solve the problem, namely, how to manage models as evolving artifacts within Jazz. Jazz was chosen as the realization of the building block Repository from Figure 3. This subsection explains how Jazz was extended to manage models in module representation the same way as source code artifacts. This extension was implemented completely and can be integrated as Eclipse plug-ins in the Jazz platform. The implementation consists of a Jazz independent framework and a Jazz adapter. The Jazz independent framework implements the module representation (Figure 7) of models and useful utilities such as an XMI im/export. Based on this architecture it is possible to implement adapters for further SCM systems in addition to the Jazz SCM.

The Jazz SCM component solely operates on versionable items. In standard Jazz, files are a special type of versionable items; in this paper, the concept is extended to model elements and accessing its elements (building block Generic Modeling Infrastructure).

The Jazz SCM-component treats all entities as versionable items. In fact, it does not make any specific assumptions about the objects being under version control. This is possible, since the data that is stored within the versionable objects is maintained by a repository component and the merging or comparison between different states (versions) of a versionable item is performed using standardized Eclipse mechanisms. However, for the purpose of this prototype, it is feasible to directly base on the Jazz SCM component.

As explained in the preceding subsection, the module concept was chosen as the internal representation of models. To enable Jazz to manage models in this internal representation, two new types of versionable items have to be defined: InformationItem and Coupling. This is achieved by the definition of a new Jazz component. Figure 8 shows its logical model defining the entities of the module concept as introduced in Figure 7.

By declaring Versionable (which is defined by the Jazz SCM component) as the super type of InformationItem and Coupling, Jazz’s SCM mechanisms are also available for the defined entities. A detailed description of the model repository implementation within Jazz and an extensive exemplification can be found in [20].

As proposed in the description of the requirement Meta-Model Independency, EMF Ecore is assumed to be the meta-model’s specification language. EMF Ecore provides facilities to define concrete meta-models that can be enriched with OCL statements expressing the meta-model’s constraints. Therefore, the existence of an EMF Ecore model defining the underlying meta-model is assumed.

Figure 8: Internal Representation as EMF Model

Within this paper, the term external representation denotes the model representation that is provided by external applications which are used for creating and modifying models. Since the meta-model is assumed to be specified with EMF Ecore, the external representation of a model – which is an instance of its meta-model – is considered to be an EMF model. EMF is used to realize the building block Generic Modeling Infrastructure (Figure 3). The use of the EMF technology has the advantage of being supported by many modeling tools using standardized persistence mechanisms. As a follow onto this, the task of storing models within Jazz boils down to a conversion between the internal and external model representations.

3.4 Modifying models by external applications

The preceding sections explained the management of models within the model repository. This subsection addresses question (c) stated at the beginning of this section. As demanded by requirement external modifications to models should be performed using external modeling applications. To support this, the requirements External Modifications and Change Detection from subsection 3.1 have to be realized. Finally, the requirements that have to be fulfilled by external applications are stated and explained.
Figure 9. Model editing with external tools

Figure 9 depicts the model editing scenario supported by the model repository prototype. In this scenario, models can be created using an external modeling application and stored to the developer’s local file system. The format of the stored model is considered to be the external representation supported by the model repository which is, as outlined in the preceding sections, a serialized EMF model. To put this model under control of the model repository, it has to be imported from the local file system. As explained in Subsection 3.3, the models are stored in the internal representation (module concept) within the model repository.

To modify models that are managed by the model repository, the model has to be exported to the local file system first, and can then be loaded and modified by the external modeling application. The modified model can then be saved to the local file system, from where it is imported into the model repository, taking over the modifications.

The operation of exporting a managed model to the local file system involves a converter that takes a model in internal representation, converts it into its external representation (as defined by its EMF Ecore meta-model) and persists the converted model using the standard XMI (XML Metadata Interchange) [25] persistence mechanism provided by EMF.

Figure 10 illustrates the user interface of the extended Jazz platform which controls the im- and export of a model for external modifications by case tools.

4. Conclusions and further work

This paper addresses the need for a comprehensive and integrated support in the team-based development of models. The result is a model repository that supports team-based development to a considerable extent, whereby the following objectives are accomplished.

**Investigation of how to adequately support the collaborative development of models.** Current solutions for concurrent model development follow the paradigm that the collaborative development of code. But it takes more than this to adequately support collaboration in model-centric development. The result is that switching the subject of development from source code to models also raises challenges on higher levels. Especially the topic of quality assurance after merging concurrently developed models is much more complex than in code-centric development. The challenges raised by considering models as subject of development lead to the proposal of a process for structuring and synchronizing tasks in the collaborative development of models.

**A model repository which supports the concurrent development of models.** Current version control systems as CVS work under a file-based paradigm that is purely textual which is not suited for typical version management operations such as comparing or merging.
Especially, this paper presents the using of an approach to manage models with respect to their graph-based structure.

**Development of an integrated solution based on the collaboration platform Jazz.** Collaboration platforms meet this emerging demand and provide advanced support for team-based development. However, these tools are focused on code-centric development.

To leverage existing tools for team collaboration, the presented system was implemented as an extension to the Jazz Platform. The system is full-integrated in the Jazz platform and provides the same versioning capabilities for models as Jazz does for source code. The implemented prototype system based on the concepts of this paper enables multiple modelers to develop models independently and collaboratively.

The results presented in this paper open up some new pathways for further research. Although the presented model repository supports optimistic merging, mechanisms for comparing and consistency management have to be added. Mechanisms have to be implemented for the identification and user-assisted resolution of these conflicts. Ideally, the system should assist users in resolving such meta-model inconsistencies. [4] provides a basis to address these topics that are currently under research.

5. References


