MoVEing Forward:
Towards an Architecture and Processes for a Living Models Infrastructure

Michael Breu, Ruth Breu, Sarah Löw
Quality Engineering Research Group
Institute of Computer Science, University of Innsbruck, Austria
{*Michael.Breu,Ruth.Breu,Sarah.Loew*}*@uibk.ac.at*

***Abstract*—Development, maintenance and operation of complex IT–systems, involving various stakeholders are challenging tasks. Managing such IT–systems with model–based approaches can help to cope with this complexity. As IT–systems are changing during their lifetime, as do the models describing certain aspects of such systems. Document and model versioning repositories are the preferred infrastructure to maintain the documents and models, reflecting the evolution of the IT–system. However, there are more complex requirements to model versioning compared to classical source code or document versioning: Depending on the types of models different modelling tools may be employed and must interface to the repository. The consistency between models must be ensured, and finally, since various stakeholders are involved, changes must be propagated between models. In this paper, we analyse these requirements and present the basic architectural concepts for a Living Models infrastructure that supports the evolution of models.**

***Keywords*-model versioning; model evolution; software engineering; change based process; tool integration; automated change management.**

I. Introduction

This paper is based on an initial presentation in [1]. Model engineering is a widely accepted engineering discipline [2], and a lot of models are developed in manifold contexts in practice.

Within software projects models form a basis for (manual or automatised) development of software. In addition models may also be used in broader context, e.g., security models help to analyse and document security concepts in critical IT–systems, business process models document interleaved business processes and IT–landscape models help to manage IT–landscapes in the context of business processes and organisational structures.

Our own engineering experience in commercial projects shows that it is hard to keep models up–to–date, if no active continuous maintenance process is in force. After the initial development phase (or after a radical renovation), a system evolves step-by-step through change requests. Quite often models die slowly in such a process, i.e., they are getting gradually out–dated, since the physical systems are evolving, without further maintenance of the model.

The reasons for dying models are manifold. One major reason is that model management is in many cases centralised to a designated stakeholder. Thus only a limited set of persons can maintain it. In real life there are various stakeholders, that have the knowledge to maintain a model, but do not have the authorisation to update the model or no tool access. Also if the modelling tool supports multiple users, concurrent maintenance of a model is not always well supported. Neither do classical version management systems like subversion support adequate conflict resolution for structured data like models. Finally, models can become quite complex, dependencies between modelled elements may be intricate, which makes it difficult to maintain the consistency of a model.

Facing this problem, [3] proposes ten principles to ensure an agile and flexible way to maintain models embedded in a change–driven engineering process. In this publication we discuss the requirements for and impacts of these principles on the implementation of an Living Models infrastructure.

Since the ten principles of Living Models [3] are the major motivation for this work, and to make the presentation self–contained, we recapitulate these ten principles shortly in the following:

Persistence Models should be stored persistently and their further evolution shall be supported.

Common System View All models of a current revision should be related by a common system view. I.e., each model maintained in the repository is considered part of a common system model. Such a system model may not exist as an explicit artifact, rather it may exist as an abstraction of the maintained models, ensured by consistency rules between these models.

Information Consistency and Retrieval Based on the common system meta–model consistency rules can be defined. Also new views on the information stored in the system model should be retrievable.

Bidirectional Information Flow between Models and Code and/or the Runtime System Code should be aware of the models and there should be an information flow from the code and the runtime system back to the models in order to enable monitoring and analysis. For source code this is quite often also known as “round trip engineering”.

Close Coupling of Models and Code Changes to code artifacts should be reflected back to the model, in order to keep them consistent.

Model Element States Each model element can have a state that reflects certain aspects in its life cycle (e.g., a risk assessment may have the states *draft, under review* and *final*, or a hardware component may have the states *under acquisition, operative, faulty* and *decommissioned*).

Change and Change Propagation The state of a model element can change between one system model version and the following. A state change may trigger other changes to the model, e.g., if an risk assessment element in a *final* state depends on another element that has changed, its state may be also changed and be reset to *draft*.

Change–Driven Process The software and system development process in a Living Models environment is driven by change events, the states of the model elements and their interrelationships. These change events may either trigger further internal changes or may be forwarded to the resonsible stakeholders to react on such changes.

Stakeholder–Centric Modelling Environments The environment should involve all relevant stakeholders with different goals (and different expectations on the type and abstraction–level of the models, as e.g., an aggregated IT–landscape model, a risk model, or a database model).

Domains and Responsibilities A variety of stakeholders operates on the system model. In order to coordinate the work on the models in an organised way, there should be assigned responsibilities for each model element.

For details about the Ten Principles we would like to refer the reader to the original publication. In this paper, we outline an infrastructure that supports those principles.

The rest of this publication is structured as follows: In the next section, we will discuss related research approaches that cover aspects of the ten principles. In the main section we introduce the major concepts and requirements of an infrastructure for Living Models, its basic architecture, and discuss the most important use cases for (meta–) model versioning and state management. We report on first experiences on a case study carried out in the context of the SecureChange project. A discussion on some collaboration aspects and conflict reduction strategies follows, before we conclude with an overview of the implementation of the initial infrastructure prototype.

II. Related Approaches

The ten principles embrace research topics that are already part of intensive research such as model versioning and merging, and meta model management.

Model repositories with versioning support are a major topic of academic and industrial research projects as, e.g., ModelBus [4], or AMOR [5] show. Both projects try to establish a central repository where models and meta models can be stored and retrieved via adaptors from various tools. Bëzivin et al. [6] coined the term MegaModel for the registry of models and meta models.

Versioning models in a distributed environment leads to the problem of conflicts due to concurrent commits. Thus adequate *model merging* algorithms are an important topic. Kolovos et al. [7] has compared the most important algorithms, as e.g., EMF Compare [8] or UMLDiff [9]. However there is still ongoing research in this area (see e.g., [10], [11]).

Related to the context of model repositories is the concept of *model integration* as a basis for (modelling) tool integration, as shown e.g., in Unicase [12], iRM [13], MOFLON [14], or (again) ModelBus [4], AMMA [6] and AMOR [5]. Mainly the integration is achieved by the implementation of suitable adaptors that connect the tools with a model repository. The category of model integration also comprises the concept of model transformation as ATL [6] or as discussed by Strommer at al. [15] and *round trip engineering*, which is already well established in many modelling tools, as e.g., Eclipse MDT [16] MagicDraw [17], or Rational Software Architect [18].

Atkinson and Stoll [19] choose a different solution by managing a common system view together with derived views.

Managing a common system model also includes the aspect of meta model management as e.g., proposed by [20]. Since also meta models may evolve there is a need for tools that allow for the co–evolution of meta models together with the associated models as e.g., COPE [21], or the work concerned by EMF Refactor [22].

A topic that is not yet sufficiently covered by research is the topic of change–driven modelling. There are certain (industrial–scale) tools that include aspects of change–driven requirements engineering, as e.g., DOORS [23] or in–Step change management [24].

These tools allow the requirements engineer to define state–based transition systems to model changes and their consequences to requirements.

The project MoVE (Model Versioning and Evolution) aims to establish an infrastructure to maintain Living Models. It does not concentrate on a single research topic, rather it strives to combine existing techniques (partially still under research, partially well established mechanisms). MoVE is based on a novel model based approach to implement a change–driven process.

III. Requirements for a Living Models infrastucture

In this section, we will map the generic principles for Living Models of Section I to requirements of a working Living Models infrastructure.

A. Concepts

Before going into details we define some concepts, that we need for the precise definition of the requirements.

A *m*odel captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the appropriate level of detail. (Taken from [25] Section 17.3.1). Models are expressed in specific concrete syntax, which can be graphical, textual or other suitable notation. Examples are state charts, business process diagrams, lists, trees or graph presentations, or any other domain specific language.

So MoVE does not require models to be expressed in specific representation (as e.g., UML notation). However, as we see later in Section V, UML/EMOF or ecore are the main candidates for the MoVE internal model representation in our prototype. All external representations are mapped to these internal representations.

A *model element* is an atomic constituent of a model [26].

A *system model* is an abstraction of all relevant concepts and their relationships in a system. Thus a system model can be seen as a set of consistent models, or vice versa, a model represents a specific perspective on the system model. We speak of a *partial* model, if we want to emphasise that a model is part of the overall system model.

A *meta model* is a model that defines model element (types) and their relationships for expressing a model [27]. In extension to this definition a MoVE meta model additionally associates state machines to model elements, in order to model an event driven process.

Expressing it in terms of the MOF layer model [28], the meta model refers to layer M2, the system model and the (partial) models refer to layer M1.

Being a little bit more formal we consider a MoVE meta model as a tuple $(MM, \mathcal{C}, \mathcal{SM}, m{:}\mathcal{SM} \rightarrow Attributes(MM))$, where

- MM is a meta model expressed as EMOF–model [26], together with
- a set $\mathcal{C}$ of OCL statements,
- a set $\mathcal{SM}$ of state machines (expressed as UML–based behavioural state machines [25]), and
- a mapping m that maps each state machine to a distinguished state attribute in the meta model.

Set $\mathcal{C}$ defines additional OCL constraints that cannot be expressed in a standard EMOF-model (as e.g., logical dependencies of model elements). $\mathcal{SM}$ and m form the definition of a change–driven model maintenance process, which will be explained in Section III-D.

B. Basic Conceptual Architecture

One of the major principles of a Living Models infrastructure is that various stakeholders can cooperate through

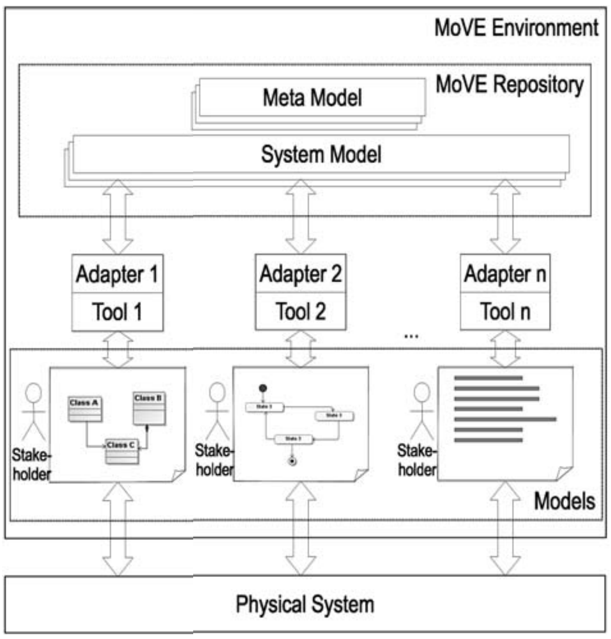

Figure 1. The Living Models infrastructure MoVE: Various tools interact with a common model repository

a set of tools via the common system model. The common system model reflects the actual state of the physical system. Each stakeholder has its own tool (set) to express and to maintain his/her own view of the complete picture. Figure 1 depicts this interaction.

Partial models from different stakeholders may overlap, e.g., in the security model risks elements are related to model elements in the enterprise model, i.e., two views onto the system model may overlap. As long as this view is generated from the system model it can be easily handled as a projection on the relevant concepts of the system model, i.e., by just ignoring irrelevant concepts of the system model.

However, we accept that partial models are the basic artefacts to collect information about the real system, we have to merge modified partial models into the common system model. Depending on the type of underlying meta–meta–model as e.g., EMOF, UML, XML different options for merging algorithms exist as explained in [29].

Merging of models leads to the problem of merge conflicts that need to be resolved manually. In Section VI we propose some heuristics how to reduce such conflicts.

C. Model Versioning

The use case diagram in Figure 2 shows a high level view on the major use cases of a MoVE infrastructure. We can roughly group the use cases into model versioning use cases and change management use cases.

The actors are typically (human) stakeholders, that are in charge either to maintain the meta model as a meta model

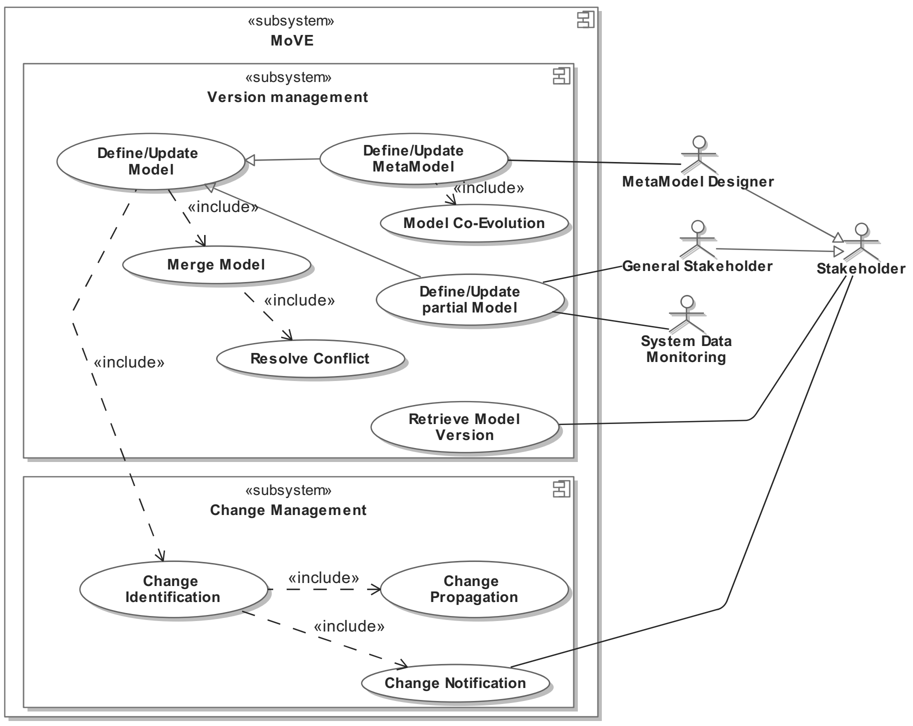

Figure 2. The actors and use cases of MoVE

designer, or to maintain his/her partial model.

Besides human stakeholders there may also be special (automatised) processes that monitor the physical system and forward system data automatically into models. An example could be an inventory process that monitors the state of all IT–applications running on a component and reports these states back into the enterprise model.

One major use case is *Define/Update Model*, where the model can be either a partial model or the meta model.

Updating a partial model may cause conflicts with the system model. Conflicts may result from classical versioning conflicts that are related to concurrent changes by different stakeholders. In this case, classical conflict resolution techniques, e.g., EMF Compare [8], may be applied. Other conflicts may result from the fact that constraints $\mathcal{C}$ from the MoVE meta–model are violated. The most consequent option for handling constraint violations would be, not to allow the commit of the partial model into the system model unless the conflicts are resolved. Since a violation is not necessarily restricted to a single partial model, or requires the interaction of several stakeholders (e.g., a cascading delete crossing model boundaries), we will just notify the concerned stakeholders that commits the change about this violation.

Partial models typically exist both in a proprietary representation used by the specific tool and as a part of the system model. So the tool adaptor must manage the mapping of the proprietary representation to the system model, support conflict resolution, and must keep the proprietary representation in synchrony with changes in the system model.

Besides the system model, also the meta model may evolve over time. In this case, mechanisms have to be provided to propagate changes in the meta model to the model layer. This is related to the large field of model migration, co–adaptation and co–evolution.

D. State Management

The second major functionality of a Living Models infrastructure is the change identification, propagation and notification.

Each class may have one or more distinguished attributes that represent the state of each instance. State transitions of such an instance are governed by an associated state machine. The set of state machines $\mathcal{SM}$ and the mapping m of the state machine to the distinguished attributes are defined in the MoVE meta model $(MM, \mathcal{C}, \mathcal{SM}, m{:}\mathcal{SM} \rightarrow Attributes(MM))$.

The MoVE repository monitors state changes of that dedicated model element attributes in a currently committed partial model and identifies an corresponding transaction in the associated state machine. The transaction may result in further transactions that changes the state of other model element (change propagation). Not all consequences of a change may be carried out automatically. Thus transactions may also trigger a notification of the responsible stakeholder. This stakeholder is then in charge of reacting on the change and to take further actions.

In order to illustrate these concepts, we present a case study in the following section.

IV. ATM Case Study

In this section, we present one of our actual case studies to give a clearer picture of how MoVE works and how it supports its users. The presented case study is taken from the area of Air Traffic Management (ATM) initiated by the EU project SecureChange [30]. It is concerned with the operational processes of managing air traffic in terminal areas, focusing on risk assessment and security analysis.

A. General Parts of the ATM Case Study

Within the SecureChange project three relevent types of models were designed, developed and maintained. In the following we will introduce the different types:

- *Enterprise Model.* This model defines the business processes, the information, the organisational units, etc. of the SecureChange project. The meta model is depicted as the bottom part of Figure 3.
- *Security Model.* This model defines all security (analysis) related objects. It contains risks, threats, security controls, security requirements and business security objectives. The (simplified) meta model is depicted as the upper part of Figure 3.
- *Common Meta Model.* This meta model describes as well the concepts of the Enterprise and the Security Model, as the interrelations between them (see Figure 3). Each element of the Enterprise Model is a generalisation of the "ModelElement"[1] of the Security Meta Model. This means that each object of an instance specification has a "state" attribute, which will be important in case security analysis started.

Both the common meta model and the enterprise model were documented with MagicDraw. The security model was documented with a project specific tool and persisted in a MySQL database. Based on the principle "Information Consistency and Retrieval" (see Section I), the Living Models infrastructure helps to keep the consistency across different models and modelling tools.

The usage of different tools in this project not only originated from different requirements of each model–purpose, but also from the different stakeholders involved, such as the maintainer of the system meta model, the responsible for IT–systems (typically the organisation's CIO or his/her deputy) and the responsible for security (e.g., the organisation's CSO). This is a consequence of the principle "Stakeholder–Centric Modelling Environments".

[1]Unfortunately the SecureChange meta model also used the term *ModelElement* as a type in its meta model, which is a slight inconsistency with the concept of *model element* in [26].

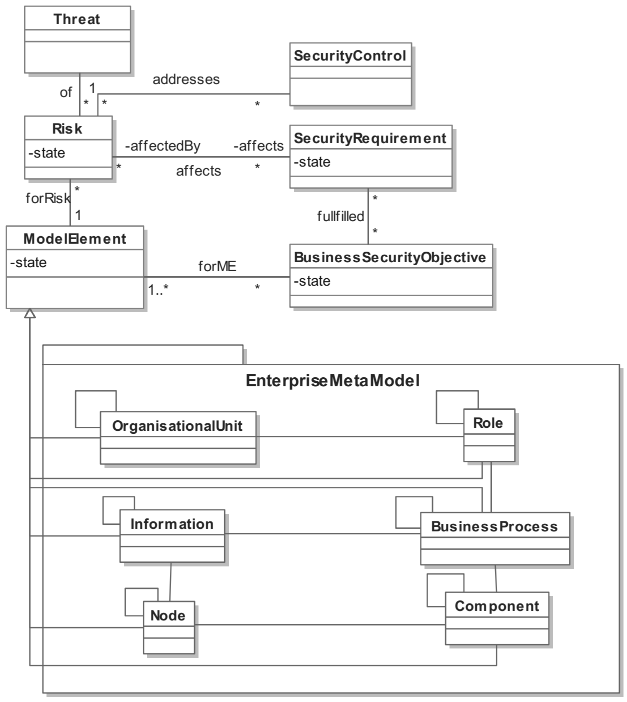

Figure 3. Enterprise Meta Model related to Security Meta Model

The security analysis conducted in the SecureChange project is based on regular system checks as soon as an element of one of the above mentioned models changes or is initially added. These checks may change the (security) state, attached to each model element. To support this security analysis the principles "Model Element States", "Change and Change Propagation" and "Change–Driven Process" have to be encompassed in the underlying infrastructure.

B. Change Management in Action

One of the main requirements of MoVE is to support Change Management and Propagation accross different models and their elements (see Section I). To fullfill this requirement we introduced the following three concepts:

- *"state" attribute:* Each model element has an attribute called "state". This attribute will be set and changed by the MoVE framework automatically (more details will follow in Section IV-C).
- *state machine definition (graphical and textual):* For each class of the Security Meta Model, a UML state machine is defined. These state machines define the possible state transitions from each model element from one state to another. To make the state machines more easy to process for MoVE, we translate them into SCXML [31], which is an XML representation, using OCL [32] for conditional expressions.

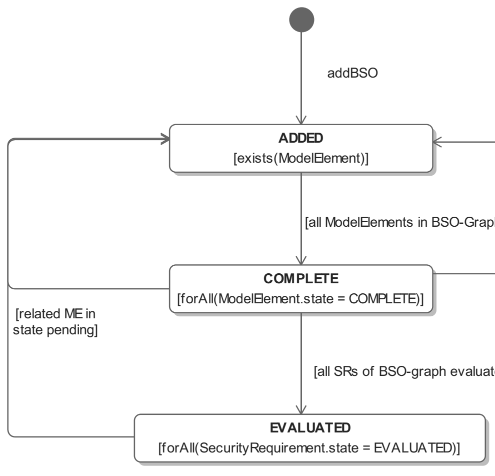
addBSO
ADDED
[exists(ModelElement)]
[all ModelElements in BSO-Graph in state complete]
COMPLETE
[forAll(ModelElement.state = COMPLETE)]
[new SR added]
[related ME in state pending]
[all SRs of BSO-graph evaluated]
EVALUATED
[forAll(SecurityRequirement.state = EVALUATED)]

Figure 4. Example state machine for a SecureChange Business Security Objective (BSO) with informal description

- *set of states:* For each class of the Security Meta Model, a set of possible states is defined. For example, a BusinessSecurityObjective can have the states “ADDED”, “COMPLETE” and “EVALUATED”.

Figure 4 shows the UML state machine for the model element BusinessSecurityObjective.

Each BusinessSecurityObjective is in one of the following states:

- **ADDED**: This business security objective is identified by somebody, but not yet evaluated.
- **COMPLETE**: All model element instances in the enterprise model associated (indirectly by SecurityRequirements) with this BusinessSecurityObjective are identified.
- **EVALUATED**: the implementation of the security objective is evaluated.

As you can see, the conditions for entering a certain state, do not only depend on BusinessSecurityObjectives, but also on various other model elements, such as SecurityRequirements. This means that changes of one model element, can enhance changes of many other model elements, triggered by the state machines.

The same state machine as in Figure 4, is given in the following Listing 1. This textual representation is written with the language SCXML (State Chart XML), an XML dialect to define state machines.

```
1<scxml version="1.0" initialstate="initstate">
  <state id="initstate">
3  <initial/>
   <transition event="addBSO" target="ADDED"
5    cond="let result: Boolean = self.forME->asSet()->
          notEmpty()"/>
  </state>
7
  <state id="ADDED">
9  <transition event="all ModelElements in BSO-Graph in
        state complete" target="COMPLETE"
        cond="
11
     public definitions context EObject
13   def treeIterator: ...
     def getAttributeValueByName: ...
15   enddefinitions

17   public queries context EObject
     result:Boolean = treeIterator(self)->forAll(o1 |
19   o1.eClass().name = 'SR' implies
     getAttributeByName(o1,'State') = 'COMPLETE')
21   endqueries
  </state>
23
 <state id="COMPLETE">
25 <transition event="all SRs of BSO-graph evaluated"
       target="EVALUATED"
        cond=" ... "/>
27 <transition event="related ME in state pending" target
       ="ADDED" cond="
     let result: Boolean = self.forME->asSet()->exists(
         ModelElement me| me.state='pending')"/>
29 <transition event="new SR added" target="ADDED" cond="
       let result: Boolean = self.fulfilled ->
     asSet()->exists(SecurityRequirement r| r.state='
         ADDED')"/>
31 </state>

33 <state id="EVALUATED">
  <transition event="related ME in state pending" target
       ="ADDED" cond="
35   let result: Boolean = self.forME->asSet()->exists(
         ModelElement me| me.state='pending')"/>
  </state>
37 </scxml>
```

Listing 1. SCXML version of Business Security Objective state machine

The structure of Listing 1 looks as follows. Except of the SCML root element, the outermost elements are identified with the tags “state” (see Lines 2, 8, 24 and 33). These states have an ID, which represents the name of the states (compare Figure 4). Within each state element, we can define several “transition” elements. Each transition has two to three attributes:

- *event:* This attribute denotes the name of the transition, e.g., compare Line 25 of the above SCXML and the name of the transition between the states “COMPLETE” and “EVALUATED” in Figure 4, named *[all SRs of BSO-graph evaluated]*.
- *target:* The target attribute denotes the destination state after transition, or also called transition target.
- *cond:* This attribute is optional. In case of existence, it contains valid OCL code, incorporating the guard condition to follow a transition. In case of absence, the transition is an “immediate” transition. It is also possible to build more complex OCL statements including “definitions” and “queries”. Whith the help of these constructs (see Lines 12-21). The definitions “treeIterator” and “getAttributeValueByName” describe helping functions to parse the commited model. The OCL queries are invoked to gather the target ModelElements and the transitions to be triggered. For example, the query starting in Line 17 uses the functions defined above to check if all ModelElements attached to the actual Business Security Objective are in the state

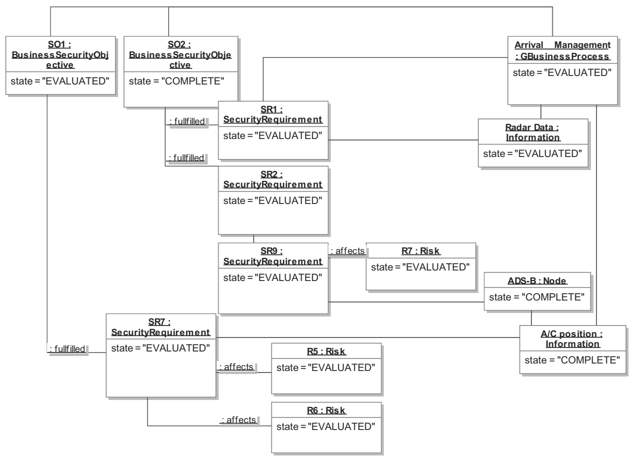

Figure 5. Commited instance model of a possible ATM case

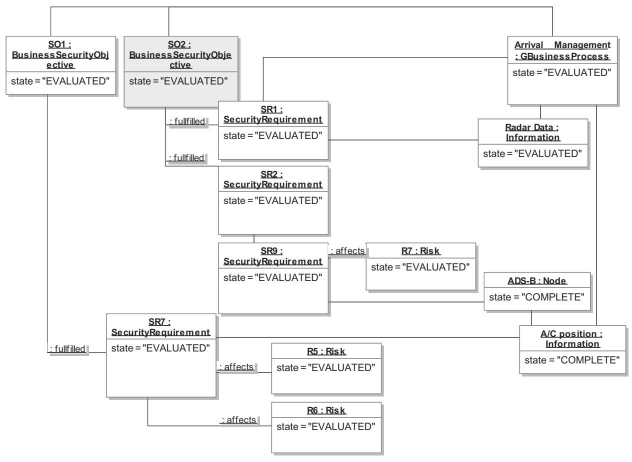

Figure 6. Instance model of Figure 5 after automatic state changes

"EVALUATED". In case the query returns the boolean value "TRUE", the transition is fired and the Business Security Objective is now in the state "COMPLETE".

Based on the state machines defined in SCXML, MoVE can check after each commit whether new state transitions are possible or not.

It may be the case that there are more than one possible transition (e.g., several model elements can change their state). In this case the out–come of the change propagation is non–deterministic. It depends on the selected sequencing of changes by the change propagation algorithm. It is in the responsibility of the meta model designer to define the state machines, such that this non–determinism does not lead to undesired results or even into circular state change loops.

C. Example Models

In this section, we introduce an example instance model of the ATM Case Study. Although the security model is stored in a relation MySQL database, the respective management tool synchronizes the content with the MoVE repository. MoVE represents the contents of the database internally by instance specifications with relationships modelled as slots [25]. Figure 5 shows a possible instance model (already converted into an internal representation) that will be commited by the user to the MoVE repository. Having a look at the states of all elements of this model, we can see that within the state machine "Business Security Objective" the condition for one transition is fulfilled. The according event is called "all SRs of BSO-graph evaluated", which means that in case of the Business Security Objective "S02", all attached Security Requirements (namely "SR1", "SR2" and "SR9") are in the state "EVALUATED". As soon as MoVE receives the instance model via a commit by the user, it triggers the according state transition. The resulting instance model after the transition is depicted in Figure 6. As soon as the transition finished its execution, MoVE parses the model element states again for possible transitions. Only if no further changes are possible and the set of model element states is stable, change propagation is finished and this version of the model will be stored in the repository.

V. Current Prototype

We have implemented an initial prototype in order to study the usage scenarios of Living Models.

The analysis of requirements presented in Section III implies a flexible architecture with exchangeable components. Therefore MoVE consists only of a small number of core components and a large number of exchangeable and extendible components. The architecture of MoVE is twofold: it consists of client–side and server–side components (see Figure 7).

To provide typical features of a standard VCS (version control system) and a stable and well tested communication protocol, Subversion (SVN) [34] is used as a core component on the client– and server–side.

The MoVE repository is built on top of an SVN server to update and commit models and meta–models. The main advantage of this decision is that both MoVE models and other artifacts can be versioned in the same repository. We use SVN properties to tag artifacts that are models, which need special handling by the MoVE infrastructure. MoVE hooks onto subversion by a subversion pre–commit hook, that invokes the dedicated MoVE functionality on the repository contents.

Conflict resolution is based on EMF Compare. State machine support is included based on SCXML.

In context of our case study (see Section IV) we have implemented three adaptors on the client–side, one for MagicDraw, one for Eclipse and another one for a proprietary

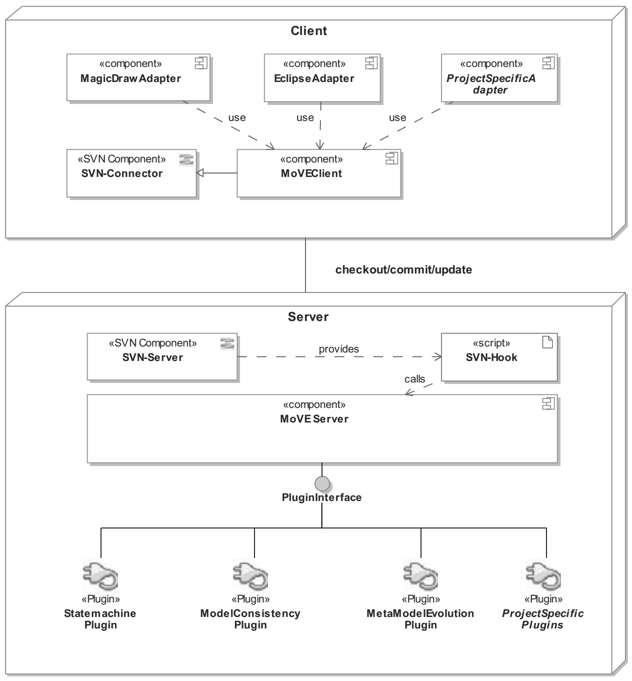
Client
«component» MagicDrawAdapter
«component» EclipseAdapter
«component» ProjectSpecificAdapter
use
use
use
«SVN Component» SVN-Connector
«component» MoVEClient
checkout/commit/update
Server
«SVN Component» SVN-Server
provides
«script» SVN-Hook
calls
«component» MoVEServer
PluginInterface
«Plugin» Statemachine Plugin
«Plugin» ModelConsistency Plugin
«Plugin» MetaModelEvolution Plugin
«Plugin» ProjectSpecific Plugins

Figure 7. The MoVE Architecture [33]

application developed on top of a MySQL data base. A somewhat arbitrary decision was, to use the UML meta model as a representational base for the partial models. This was the best choice for a research prototype, because a lot of UML–support is available in the context of eclipse EMF. Model instances of the SecureChange Security Model were mapped from the database to instance specifications in UML. However the performance issues of handling a UML–model, compared to navigating through an optimized relational database, would not be acceptable in a professional application context.

VI. Conflict Reduction Strategies

Naturally when working jointly in a team on a set of documents (or partial models), concurrent conflicting modification of versioned items may occur.

Finding differences (and conflicts) between models is not as simple as in linearly structured documents as e.g., text files [35]. There exist also algorithms [7] to find differences and conflicts in models (as e.g., EMF Compare). However it turns out that in practical applications conflict resolution in models can be fairly cumbersome, if changes are quite complex. This get's even more complex if a system model is represented in various partial models.

In order to reduce the complexity of changes to the system model, we propose some rules to constrain most changes locally to partial changes and to manage changes that cross the boundaries between partial models.

MoVE currently uses the UML/EMOF meta model as its internal meta meta model. A MoVE meta model (e.g., as in Figure 3 is an instance of the EMOF meta model. The elements in the partial models are typically on the instance level, e.g., instances of SecurityRequirements or BusinessSecurityObjectives. The main elements for change on the instance level are instances of classes, attributes and associations. In Table I we identify atomic operations on this instances that can be sequentially combined to a more complex change.

object type	change type	
class	create	a new instance is created
	change	basic type attributes (e.g., the name) are changed
	delete	the instance is removed (together with its attributes and associations to other class instances)
attribute	create	an attribute is added to a class instance
	change	an attribute value is changed
	delete	an attribute is removed from a class instance
association	create	an association instance is added between two class instances
	change	not relevant
	delete	an association instance is deleted between two class instances

Table I
Atomic changes to a (partial) model

The main rule is, that each class, attribute, and association has its owner tool. Changes to instances of classes, attributes, and/or associations are only allowed by the tool that "owns" this class or attribute.

Such a rule is not unusual in practical applications, because every model element typically has its stakeholder that manages all instances of this type. A *security objective* instance is maintained by the CSO in his/her tool to maintain the security model, a *service* instance is maintained by the CIO in his/her tool.

However there may be situations where changes to some object type may inflict immediate inconsistencies, as e.g., an class instance cannot be deleted, because it would result in a constraint violation of the multiplicity of an association. A simple example could be that the class instance is partner in an association that must hold exactly one object of this type. Such a constraint can be compared to a foreign key constraint violation in a relational database.

A simple example could be that every *BusinessSecurityObjective* instance must be assigned to at least one *ModelElement* instance as in association *forME* shown in Figure 3. Since *BusinessSecurityObjective*s are managed by a stakeholder A, who is different from stakeholer B in charge for *ModelElement*s, a *ModelElement* could never be deleted by B, before the stakeholder B deletes the corresponding

BusinessSecurityObjective. This would be very unfunctional and counter–intuitive.

This type of problem can occur with any consistency constraint that crosses model boundaries (i.e., involves instances from several partial models). The solution is to "extend" the final state of the class' state machine by adding a extra state named "toBeDeleted" (see Figure 8) into the state machine of a ModelElement. A plug–in that is in charge for maintaining the consistency between partial models, converts the deleted instance into an object of state "toBeDeleted".

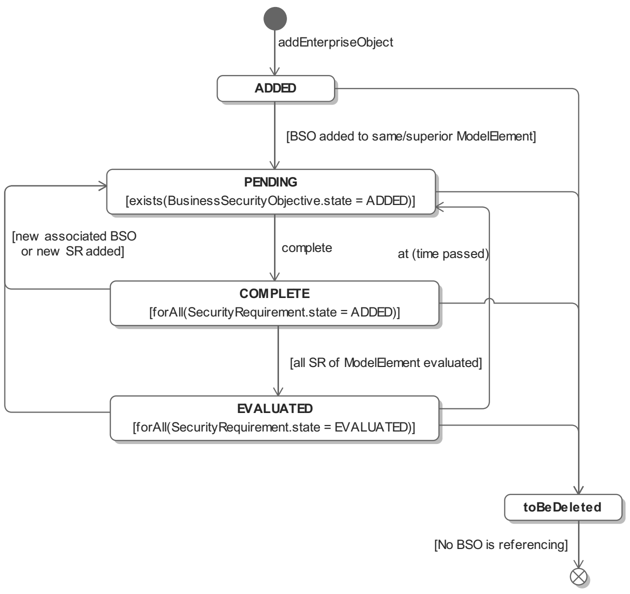
addEnterpriseObject
ADDED
[BSO added to same/superior ModelElement]
PENDING
[exists(BusinessSecurityObjective.state = ADDED)]
[new associated BSO or new SR added]
complete
at (time passed)
COMPLETE
[forAll(SecurityRequirement.state = ADDED)]
[all SR of ModelElement evaluated]
EVALUATED
[forAll(SecurityRequirement.state = EVALUATED)]
toBeDeleted
[No BSO is referencing]

Figure 8. Extended state machine for ModelElement

Due to the notification mechanisms in the MoVE infrastructure the relevant stakeholder can be notified that he has to correct his partial model, in order to allow for the final deletion of this instance.

VII. Conclusion and Future Work

Managing IT systems with *Living Models* and to keep them alive, requires a rigorous development and maintainance process. This process must be change driven, in order to effectively coordinate the interaction of various stakeholders, taking jointly the responsibility for a complex system.

We have shown the major requirements and architectural implications for a *Living Models* infrastructure. Modelling relevant aspects of an IT–system and keep them up–to–date, needs an appropriate tool and cooperation support.

It strongly depends on the project or system context how comprehensive such a common system model can be. Establishing a "universal" system model where all tools describing any aspect of a system can contribute, will be doomed to failure. Our experience is that one should find a level of detail in the common system model, which is enough to control the relationships between domains and responsibilities of the stakeholders.

The central issues are model versioning and change–driven model evolution. Especially the second issue needs excellent tool support in order to ensure a lively modelling culture. Manual checks would fail, because models and their interdependencies tend to become more and more complex.

The challenge is to both combine results from different model research areas, and from well established software engineering disciplines, as e.g., classical versioning, flexible plug–in architectures together. The MoVE infrastructure represents an initial prototype that implements a working environment.

The prototype allowed us to study first impacts of a change–driven system management process based on models. We are sure that this is a natural evolution of the ideas of "model driven software engineering". Besides technical challenges like scalability, and extendibility, we still see a lot of open research questions. Challenges are:

- The implementation of adapters for various modelling tools is still a tedious work. The reason for this is, that not all tools technically provide usable interfaces to implement adapters, but also that the semantic gap between the tool's modelling representation and the chosen internal MoVE representation in EMOF can be quite huge.
- Change propagation can only be done partially automatic. I.e., efficient change notification mechanisms are needed to forward required actions to the relevant stakeholders. So MoVE should interface to work flow systems, which have a lot of mechnisms to handle event notification, and process control.
- A change–driven process also has social implications. It is a cultural shift from a capability based process to a change–driven process. What is the best way to bring such a process into an existing organisation?

This will be the starting point for further research on "Living Models".

This work is partially sponsored by the projects SecureChange (EU Prj. No. ICT-FET-231101), QELaB, and COSEMA (funded by the Tiroler Zukunftsstiftung).

References

[1] M. Breu, R. Breu, and S. Löw, "Living on the MoVE: Towards an Architecture for a Living Models Infrastructure." in *The Fifth International Conference on Software Engineering Advances (ICSEA 2010)*, IEEE Computer Society. Nice, France: IEEE Computer Society, 08/2010 2010, pp. 290–295.

[2] R. Huber, "Agile Practices and Model Driven Development in Avionics Software Development," in *Software Engineering Live*. Valtech GmbH, 2009.

[3] R. Breu, "Ten Principles for Living Models — A Manifesto of Change–Driven Software Engineering," in *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS))*, 2010, pp. 1–8.

[4] A. Aldazabal, T. Baily, F. Nanclares, A. Sadovykh, C. Hein, and T. Ritter, "Automated Model Driven Development Processes," in *ECMDA Workshop on Model Driven Tool and Process Integration*. Stuttgart: Fraunhofer IRB Verlag, 2008.

[5] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, W. Schwinger, M. Seidl, and M. Wimmer, "AMOR — Towards Adaptable Model Versioning," in *1st Int. Workshop on Model Co-Evolution and Consistency Management, in conjunction with Models' 08*, 2008.

[6] J. Bezivin, F. Jouault, P. Rosenthal, and P. Valduriez, "Modeling in the Large and Modeling in the Small," in *Model Driven Architecture*. Springer, 2005, pp. 33 – 46.

[7] D. S. Kolovos, D. D. Ruscio, A. Pierantonio, and R. F. Paige, "Different models for model matching: An analysis of approaches to support model differencing," in *Proc. of the Int. Workshop on Comparison and Versioning of Software Models (CVSM) in conjunction with ICSE, Vancouver*. IEEE Computer Society, 2009, pp. 1–6.

[8] Eclipse Foundation, "EMF Compare," July 2010, http://wiki.eclipse.org/index.php/EMF_Compare. Accessed on Jan 05, 2011.

[9] Z. Xing and E. Stroulia, "UMLDiff: An Algorithm for Object–Oriented Design Differencing," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2005, pp. 54–65.

[10] M. Schmidt, S. Wenzel, T. Kehrer, and U. Kelter, "History–based Merging of Models," *ICSE Workshop on Comparison and Versioning of Software Models*, vol. 0, pp. 13–18, 2009.

[11] M. Kögel, J. Helming, and S. Seyboth, "Operation–based Conflict Detection and Resolution," in *CVSM '09: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 43–48.

[12] B. Brügge, O. Creighton, J. Helming, and M. Kogel, "Unicase – An Ecosystem for Unified Software Engineering Research Tools," in *Third IEEE International Conference on Global Software Engineering, ICGSE*, 2008.

[13] I. Petrov and S. Jablonski, "An OMG MOF Based Repository System with Querying Capability — The iRM Project," *Proceedings of iiWAS*, vol. 4, 2004.

[14] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, and A. Schürr, "Metamodel–based tool integration with MOFLON," in *Proc. of the 30th Int. Conf. on Software Engineering. New York, NY : ACM*. ACM, Januar 2008, pp. 807–810.

[15] M. Strommer and M. Wimmer, "A framework for model transformation by–example: Concepts and tool support," *Objects, Components, Models and Patterns (TOOLS). LNBIP. Springer, Heidelberg*, pp. 372–391, 2008.

[16] Eclipse Foundation, "Model Development Tools (MDT)," http://www.eclipse.org/modeling/mdt/?project=uml2. Accessed on May 17, 2010.

[17] No Magic, Inc., "MagicDraw," http://www.magicdraw.com/. Accessed on May 17, 2010.

[18] IBM, "Rational Software Architect," http://www-01.ibm.com/software/awdtools/swarchitect/websphere/. Accessed on May 17, 2010.

[19] C. Atkinson and D. Stoll, "Orthographic Modeling Environment," in *FASE*, 2008, pp. 93–96.

[20] B. Baudry, C. Nebut, and Y. Le Traon, "Model–driven engineering for requirements analysis," in *11th IEEE International Enterprise Distributed Object Computing Conference, 2007. EDOC 2007*, 2007, pp. 459–459.

[21] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "COPE — Automating Coupled Evolution of Metamodels and Models," in *23rd European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, S. Drossopoulou, Ed., vol. 5653. Springer, 2009, pp. 52–76.

[22] T. Arendt, F. Mantz, L. Schneider, and G. Täntzer, "Model Refactoring in Eclipse by LTK, EWL, and EMF Refactor: A Case Study," in *Models and Evolution, Joint MoDSE–MCCM 2009 Workshop*, 2009.

[23] IBM, "Rational DOORS," http://www-01.ibm.com/software/awdtools/doors/. Accessed on May 17, 2010.

[24] micro TOOL, "in–Step," http://www.microtool.de/instep/. Accessed on May 17, 2010.

[25] OMG, "Unified Modeling Language 2.2," 2009. [Online]. Available: http://www.omg.org/spec/UML/2.2/Superstructure/PDF/

[26] OMG, "Meta Object Facility (MOF) Core Specification, Version 2.0," January 2006.

[27] OMG, "Unified Modeling Language 2.0 Infrastructure Specification," September 2003.

[28] OMG, "Meta Object Facility (MOF) Core Specification, Version 1.4," April 2002.

[29] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, "An Analysis of Approaches to Model Migration," in *Models and Evolution, Joint MoDSE–MCCM 2009 Workshop*, 2009.

[30] A. Armenteros, B. Chetali, M. Felici, V. Meduri, Q.-H. Nguyen, A. Tedeschi, F. Paci, and E. Chiarani, "D1.1 Description of the Scenarios and their Requirements," SecureChange, Tech. Rep., 2010, http://securechange.eu/sites/default/files/deliverables/D1.1_Description_of_Scenarios_and_their_requirements.pdf. Accessed on January 14, 2011.

[31] J. Barnett, R. Akolkar, R. J. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. McGlashan, and T. Lager, "State Chart XML (SCXML): State Machine Notation for Control Abstraction," World Wide Web Consortium, Working Draft WD-scxml-20080516, May 2008.

[32] OMG, "UML 2.0 OCL Specification," *(ptc/03-10-14)*, 2003.

[33] P. Kalb, "UML Model Merging in the Context of a Model Evolution Engine," Master's thesis, Leopold-Franzens-University Innsbruck, 2010.

[34] C. M. Pilato, B. Collins-Sussmann, and B. W. Fitzpatrick, *Version Control with Subversion*. O'Reilly Media, 2008.

[35] The GNU Team, "Comparing and Merging Files," April 2002, http://www.gnu.org/software/hello/manual/diff.html. Accessed on Jan 05, 2011.