

Evolution control in MDE projects: Controlling model and code co-evolution

Jacky Estublier, Thomas Leveque, German Vega

LIG-IMAG, 220, rue de la Chimie BP53, 38041 Grenoble Cedex 9, France
{[@imag.fr">Jacky.Estublier,Thomas.Leveque,German.Vega](mailto:Jacky.Estublier,Thomas.Leveque,German.Vega)}@imag.fr

Abstract. The dream of Model Driven Engineering (MDE) is that Software Engineering activities should be performed only on models, but in practice a significant amount of programming is still being performed. There is a clear need to keep code and models strongly synchronized when they represent the same entities at different levels of abstraction. We observe that versioning is ill supported by MDE tools, and that no strong synchronization is ensured between code and model versions. This, among other things, explains why MDE is not widely adopted in industry.

This paper presents the solution developed in the CADSE project for providing consistent support for model and code co-evolution. It is shown that it requires to (1) define, what evolution policy is to be applied, (2) closely synchronize both ways, the model entities and the computer artifacts, and (3) enforce consistency constraints and evolution policies during the commit and check-out of both model elements and their corresponding artifacts.

Keywords: Domain Specific Languages (DSL), metamodel-based environments, composition of DSL interpreters, composition of models.

1 Introduction

The main claim of Model Driven Engineering (MDE) [9] is that Software Engineering activities will be performed mostly, if not only, on models, all along the life cycle of software products. Almost 10 years later, this vision is still not a reality in industry, except for a few niches and in certain specific conditions [2].

It is an observation that the major progresses in Software Engineering (code engineering) are due to the high quality of environments and tools that assist Software Engineers in their day to day work. A major problem faced, and to a large extent solved, is evolution control; but it took three decades from SCCS [8] before obtaining reliable and convenient version and configuration control systems.

No such history is available for MDE. Today, model versioning and merging is a current research topic in its infancy. Therefore, no evolution-control system of industrial strength is currently available for model engineering. What makes the situation even worse is that in practice, a large fraction of Software Engineering activities consists in developing code and many other files, such as scripts, metadata, or documentation. From a purist MDE point of view, all these files are models, but from a Software Engineering perspective, these files are managed in the “old” way, relying on traditional evolution control tools, while no such tool exist for (real) models.

This situation requires the definition of new evolution control paradigms for software projects that are made of a mixture of models and files. On one hand, these evolution paradigms must take into account the nature of models and must be adapted to

model engineering practices. On the other hand, such paradigms must respect the code-engineering practice and must rely on the tools and systems available in traditional Software Engineering. We believe that this lack of consistent evolution support, on all grounds - policies, methods, tools, and environment, is one of the major reasons why MDE did not succeed in industry so far.

This paper shows that at least three issues have to be addressed for defining such evolution paradigm and its associated tools. We believe that evolution control, in MDE projects, requires solving the three issues:

- Synchronization of model elements with code and files
- Definition and support of evolution policies
- Definition and support of consistency constraints

Section 2 presents how are synchronized model elements and code, section 3 describes how can be defined evolution policies, and section 4 shows how consistency is computed and enforced. Finally, section 5 concludes and proposes future works.

2. Synchronizing model and software artifacts

Model transformations began to be used for maintaining consistency between model and software artifacts when the application code is fully derived from a model [5]. In that case, users only work on the model, while artifacts are (re)generated when needed; in other words, the model is a high-level source code.

In general, models do not contain enough details to be executable. This is why developers work on model and code at the same time. Usually, code skeletons are generated from the model, but the model cannot be reconstructed from artifacts and vice versa. Hence, we fall into synchronization issues where modifications on model and artifacts must be reconciled. Few MDE tools support permanent synchronization both ways between model and artifacts.

In our CADSE environment [4], traceability links can be defined between the model and the artifacts. These links translate the operations performed on the model to modifications performed on artifacts and vice-versa. It is possible, for example, to define that the concept of *service* defined in a metamodel should be mapped to an Eclipse java project with a specific structure and specific files (e.g. metadata information and templates). The synchronization ensures that each time a *service* is defined in a model the corresponding Eclipse project is created. Conversely, changes in some files (metadata) are translated into attributes and relationships in the model.

This synchronization mechanism enables Software Engineering activities be performed either on models or on files, depending on the nature of the activity, while enforcing consistency between both views. This means that a model element and its mapping must evolve in concert, and since versioning is defined at the file granularity, versioning must also be applied on each model element individually. This characteristic alone disqualifies most model versioning approaches where the grain of versioning is a complete model, or a large fraction of it.

The CADSE environment is an eclipse plug-in. The model is found in an Eclipse window, while the artefacts are provided as eclipses projects or files. Under Eclipse,

users can directly create, delete, or change model elements and execute engineering operations such as build, package, or edit on these elements.

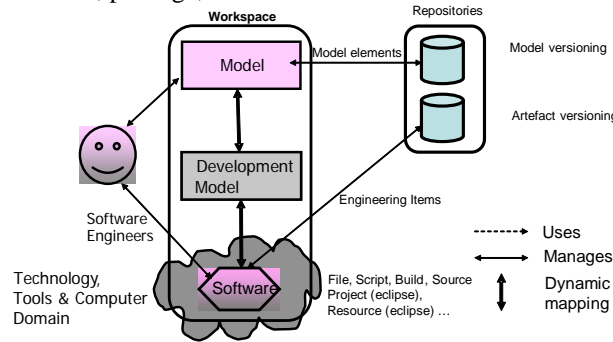


Figure 1. Model and code versioning

3. Evolution policies

Versioning should be a mechanism that serves an evolution policy. Unfortunately, in most systems, only the mechanism exists, while the evolution policy remains undefined and relies on the good will of developers. In the CADSE project, the ambition is to make explicit and automatic the evolution policy. To that end, we have defined a version model and a versioning mechanism and we show how versioning policies can be defined and enforced.

3.1 Data model, version model.

We use ECore [1], which is an Object oriented data model. Basic entities are classes, which have attributes and operations, and which are related to other classes by relations and possibly by the special inheritance relationship. This is enough for our purpose (please refer to [3] for more details), but this data model does not include any versioning concept.

Before addressing evolution policies, we need to define the concept of a version and its associated *version model*. Two objects are versions of each other if they share something (otherwise they are two independent objects) and if they are different in some way (otherwise it is the same object). Following [6], we decided to make explicit the common part constituted of shared attributes, and the difference as two objects. The different parts (not common one) are called “the revisions”. A versioned object (by language abuse often called a revision) is the union of the information found in the common part and the one found in one of its revisions. A branch is the common part and all its revisions. By convention, the name of a branch is the name of the common part. This is a symbolic name, where the name of a revision is an integer and the name of a versioned object has the form “*branch.revision*”.

3.2 Object evolution control

Evolution control refers to the criteria by which new versions of an object are created. Two classes of criteria can be identified: (1) how important is the change that has been made on an object (to which extent it changes its semantics); and (2) when to record the change. In theory, a change should be recorded only when the object is in a stable state. Unfortunately, we do not know enough on the object semantics to evaluate the object state which is mostly on the artifact state: is the code of `videoPlayer.3` consistent, reliable and so on. For that reason, as usual, the decision to commit changes is left to the engineer. Depending on the change importance, versioning may mean:

- Update the current value of the object in the repository
- Create a new revision of the object (in same variant)
- Create a new variant
- Create a new object

Traditionally, this choice is also left to the user. However the importance of a change can be related to the semantics embedded in each attribute.

In our system, a versioning characteristic which can be *mutable*, *immutable*, *final* or *transient* is associated to each attribute and relationship. If a *mutable* attribute is changed, the next commit will simply update its value in the repository. If an *immutable* and *revision* attribute is changed, the next commit will create a new revision of the object. If an *immutable* and *shared* attribute is changed, next commit will create a new branch. Changes in a *transient* attribute are supposed to be transparent for the versioning system and *final* attributes, if they need to be changed, must create a new object.

3.3. Relationship evolution control

Relationships are seen as attributes of the source element and can consequently have the *mutable*, *immutable*, *final*, *transient* evolution characteristics. However, due to our version model, the origin of a relation can be a revision, or a branch, and the destination can also be a branch, or a destination. This leads to the definition of different kinds of relationships:

- Global references (Branch to Branch). Only the last state (entity source and entity destination) of the link is saved. This is called product first in [7].
- Contextual references (Revision to branch). This is the way to define a non-versioned entity.
- Branch to revision. This is possible, but seldom used.
- Version-specific references (Revision to revision).
- Effective references (X to $\text{Set}(\text{Revision})$). Instead of pointing on one or all revisions of an entity, effectivity, which comes from the PDM world [3], associates a subset of all revisions.

In section 3.2, we have shown that each attribute and relationship can be annotated to indicate how the object should react (from the versioning point of view) when that attribute changes. Based on attribute annotations, we can update, create a new revision or create a new variant only when needed. But this mechanism ignores the im-

part of such changes on the surrounding objects. This is what we call change propagation, defined at type level, using the following annotations on relationships:

- **MutableDestination:** changing the revision number of the destination object does not have impact on the origin entity, but now the link targets the new one.
- **ImmutableDestination:** a new revision of the origin entity is created when the destination object revision changes.
- **EffectiveDestination:** the entity is compatible with the previous and the new destination state.
- **FinalDestination:** the destination is not allowed to be modified.

The annotations on attributes and link allow defining (at type level) the evolution strategy to be applied. It allows, for each kind of change, for the precise definition of the evolution control that the system should perform. It also allows the definition of the versions to be created (i.e. update, revision, or variant), and the objects to which the change propagates.

Our objective is to avoid version proliferation, which is why defining state and action propagation is a priority for us. In all cases, it is important to realize that these annotations are defined at type level by software experts. In a workspace, the developers only “see” a versioned object, the distinction branch/revision and the revision numbers are not visible. Additionally, propagation and evolution strategy are fully automated and transparent.

4. Definition and support of consistency constraints.

Remember that the goal of versioning is to store objects in a given state so that they can be used at a later time. One purpose of storing different versions is to recover from crashes and mistakes, but the major purpose is to reuse stored versions in different assemblies. Reuse is the major driver for setting versioning strategies.

Optimizing reuse entails a number of issues. In theory, the lower is the granularity of the objects to reuse, the higher is the number of possible different assemblies. This is theoretical only because: (1) not all assemblies are valid, and (2) the capability for a human to build a complete and consistent assembly without help is limited. Indeed, these reasons explain the current practices. As they are not able to build an assembly piece by piece, developers only use existing assemblies (snapshots, baselines), built and checked by somebody else. Any change in the assembly produces a new revision of everything. Consequently, this results in a huge number of useless revisions, as only the new revision of the baseline will actually be used. Ironically, this practice generates a huge number of revisions of everything, while in practice the grain of reuse is highly coarse. Namely, the reuse granularity is the complete baseline which explains why reuse is far from optimal. Optimizing reuse requires the following:

- Allow all possible assemblies.
- Avoid inconsistent assemblies.
- Help in building complete and consistent assemblies.

From a versioning point of view, these requirements translate as: (1) a version should be at the lowest possible granularity level, (2) version compatibility should be

defined and controlled, (3) the number of versions should be minimized and (4) the dependencies should be known.

The propagation control mechanism described above already solves the first and the last of the above requirements. This is because the granularity is the element, because the minimum side effects of a change are computed (relationship propagation) in order to find out the elements that really need to be saved (modified) and because revisions are only created when required (immutable attributes). All and only the useful versions (updates, revisions and variants) are created.

For two elements, $o.i$ and $d.j$, **link r is said to be consistent** if o or $o.i$ is the origin of r , and if d is the destination (if r is branch), or if j pertains to the effectivity of r . **An assembly is said to be consistent**, if all links between elements pertaining to the assembly are consistent. We define a **required-set** as a set of relationship types $RS=\{R1, \dots Rn\}$. **An assembly is said to be complete**, with respect to a required-set RS , if for all elements of the assembly, each link whose type is in RS leads to another element of the same assembly.

Note that the above definition of consistency relies on the effectivity mechanism. On one hand, the definition is conservative because the system only records in the effectivity those pairs (origin-destination) that have been tested in a workspace. Other pairs may be consistent but have not been tested. On the other hand, the effectivity is not a formal proof of compatibility. More precisely, the system assumes that only assemblies that have been tested are committed on the shared repository.

Workspace management is strongly linked to these definitions. In our system, assemblies can be explicitly built, by importing the element d or revision $d.j$ that the user thinks is needed from the shared repository to the workspace. But before importing d , the system checks which revisions of d are consistent with the elements already present in the current workspace. If such revisions d exist, then the system imports the most recent revision of d . If no such revision of d exists the system sends a warning to the user. This manual process can be widely facilitated by declaring some relationship type as *required*. In this case, importing an element automatically imports all *required* elements. Associating a required-set RS to a workspace is such that importing an element ensures the completeness of that element with respect to RS .

Based on these mechanisms, creating, maintaining and evolving complete and consistent assemblies is easy and does not require from the user any knowledge of dependencies and compatibility.

5. Conclusion

The MDE approach is still novel and as such lacks methods and tools required by practitioners. This work seeks to provide concepts, methods and tools that allow for closely controlling the evolution of a software product, all along its life cycle, while following a MDE approach. First, developers have to deal with both models and files. These files could represent programs in case models are not rich enough to be executable, which is most often the case, and in case those files are needed for compiling, packaging, or documenting. We believe that this situation is not due to the transition

from code to model engineering but that, at least for the foreseeable future, models and files will co-exist and will have to be managed together consistently.

The models we suggest developing are not primarily intended to describe the application to build. Instead, our models provide a high-level description of the information found in the computer during the project life cycle. A large fraction of these models represent application components, as well as other concepts. In a CADSE system, the software expert specifies models and metamodels for describing how model elements are linked with computer artefacts. A model element may be mapped to a few lines in a file, or to large structures such as projects. We believe that synchronizing models and artefacts is a prerequisite to any MDE projects.

The next topic that hampers MDE is evolution control. As a model element can be potentially mapped to large software artefacts that need to be versioned, the model element has to be versioned as well. This raises the issues of versioning model elements and software artefacts in a consistent manner.

The design of our system is based on few principles. First, the developer must be oblivious of the complexity involved in versioning and in evolution control. To that end, the developer works in a workspace in which the concept of version does not appear. Versioning is automatically executed following the evolution strategy that has been defined. Versioning and evolution strategies are established once and for all by software experts at the metamodel level.

The system explicitly targets reuse. To that end, the versioning granularity is rather low (i.e. a single model element), while version proliferation is avoided. This feature ensures that all possible combinations of elements are possible. Among other things, the evolution strategy renders explicit the kind of versions to be created with respect to the changes performed and with respect to the semantic of the links established among objects. The evolution characteristics established on links allow the software expert to express what does mean compatibility and in which way changes propagate.

During the execution of the evolution strategy, the system creates the versions as required, but also automatically updates and records compatibility information. Based on the recorded compatibility information, the system is able to determine if an assembly is consistent, as well as to suggest which changes would be required for making the assembly consistent, or to assist the user in rendering a workspace consistent.

Completeness can be checked, with respect to some relationship types. Therefore, the system is capable to build, or to assess, complete and consistent assemblies with minimal knowledge from the developers.

We believe that this system, which has been daily in production over the last 3 years, is a significant step towards the practical use of the MDE approach in software engineering.

CADSEg can be downloaded from <http://cadse.imag.fr/>

References

- [1] Emf project, <http://www.eclipse.org/modeling/emf/?project=emf>.
- [2] P. Tarr B. Hailpern. Model-driven development: The good, the bad, and the ugly. *IBM SYSTEMS JOURNAL*, 45(3):451, 2006.
- [3] Jacky Estublier and German Vega. Reconciling software configuration management and product data management. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 265–274, New York, NY, USA, 2007. ACM.
- [4] Philippe Lalanda Thomas Leveque Jacky Estublier, German Vega. Domain specific engineering environments. In *APSEC*, 2008.
- [5] Simon Helsen Krzysztof Czarnecki. Classification of model transformation approaches. In *OOPSLA*, 2003.
- [6] Shamkant B. Navathe Raji Ahmed. Version management of composite objects in cad databases. In *ACM*, 1991.
- [7] Bernhard Westfechtel Reidar Conradi. Towards a uniform version model for software configuration management. *LECTURE NOTES IN COMPUTER SCIENCE*, 1997.
- [8] M. J. Rochkind. The source code control system. In *IEEE Trans. on Software Engineering*, pages 364–370, 1975.
- [9] Douglas C. Schmidt. Model-driven engineering. page 7, 2006.