Transformation Patterns for Multi-staged Model Driven Software Development

Alexandre Bragança[1] and Ricardo J. Machado[2]
[1] *Dep. Eng. Informática, ISEP, IPP, Porto, Portugal,*
alex@dei.isep.ipp.pt
[2] *Dep. Sistemas de Informação, Universidade do Minho, Guimarães, Portugal,*
rmac@dsi.uminho.pt

Abstract

Model driven approaches are shifting software development from a code based activity to a model based activity. Models can be refined and transformed from requirements into code specific to a platform. Although several model transformations can occur, they usually take place at a single development stage. In the case of software product lines, and particularly of software factories, the modeling of a system can occur at several stages, for instance, at the software-house, at the systems integrator and at the final customer site. Basically, this requires that the model used at a particular stage can be refined at the next stage. In this paper, we explore the issues related to such an approach and we propose model transformation patterns that can be generically applied to models so that they can be used in multi-staged modeling approaches. We show how to realize the approach with the Eclipse Modeling Framework and present an insurance case study.

1. Introduction

The model driven approach is rapidly evolving and with a potential of becoming the next mainstream paradigm for software development. In this new paradigm, models play the central role, as the code does for traditional approaches. Models are used to construct abstractions of the system at several levels and from different perspectives. Models at higher abstraction levels can be transformed into models at lower abstraction levels and, eventually, models are transformed into code that can be executed by a specific platform. Usually, this is done at a single stage. For instance, a software house can apply this approach to build its software packages. However, in the case of software product lines, and particularly of software factories, the modeling of a system can occur at several stages, for instance, at the software house, at the system integrator and at the final customer site. Generically, one can say that in this case, the software system can be specialized at all the stages (or tiers) of the supply-chain. Such scenario requires that the models used at a particular stage can be refined at the next stage.

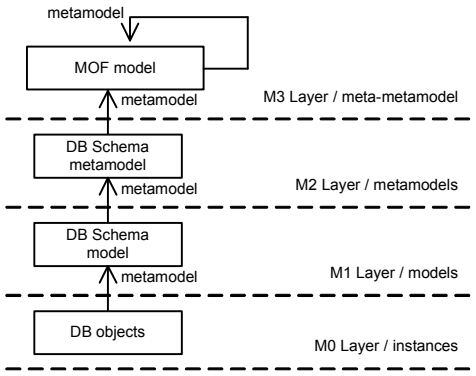

Fig. 1. Example of MOF metadata architecture.

The Model Driven Architecture (MDA) is the Object Management Group's (OMG) approach to model driven development [1]. At the core of this architecture is the Meta Object Facility (MOF) standard [2]. MOF provides a metadata management framework and a set of metadata services to enable the development and interoperability of model and metadata driven systems. Figure 1 presents an example of the MOF metadata architecture for supporting database schema modeling. The figure represents the relationships between models at different levels of the MOF architecture. This figure also represents very well the metadata architecture for single-staged software development approaches. In this paper, we will address in a practical way multi-staged model driven software approaches and how they differ in their nature from single-staged approaches. As we will see, multi-staged approaches result in a series of method recipes for applying model driven technologies in a way similar to design patterns [3]. As such, we will present the multi-staged model driven software

978-0-7695-3303-2/08 $25.00 © 2008 IEEE
DOI 10.1109/SPLC.2008.41

approach as a series of model driven transformation patterns. We will do so using the Eclipse Modeling Framework (EMF) [4], an eclipse based metamodeling framework that conforms to the MOF standard, and an insurance software supply chain as a case study.

The remainder of this paper is structured as follows. In Section 2, we present model driven engineering approaches with actual technologies (e.g., EMF) and motivate the reader to our approach for multi-staged modeling. In Section 3, we discuss model transformations and the particularity of multi-staged model transformations in the context of a software insurance supply chain case study. Section 4 is dedicated to present and illustrate the multi-staged model transformation patterns. Section 5 is dedicated to the discussion of the approach and of related work. In Section 6, we provide some concluding remarks.

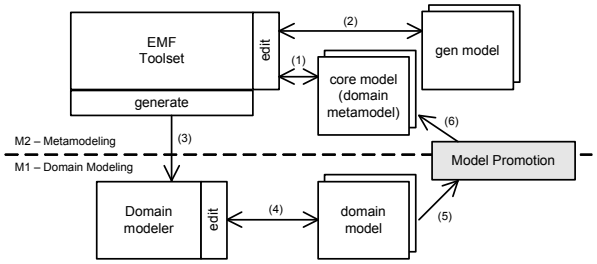

Fig. 2. Using EMF toolset to generate a domain modeler.

2. Motivation

To motivate the reader, we will use in this section the very simple and common example of domain modeling database schemas. So, using metamodeling tools, the goal is to build a database schema modeler application. We will briefly illustrate how this can be achieved with EMF.

Figure 1 presents the relationships between database concepts and the MOF architecture. In Figure 2, we see how the EMF toolset can be used to support the building of a domain specific modeler. As the figure suggests, in EMF, the domain metamodel is specified in the form of a core model, which is the format of EMF metamodels. The metamodel of these core models is called ECore. ECore was influenced by MOF [2] and, to a certain extent, we can consider it as a subset of MOF. In our simple example, the metamodel for database schemas would then be specified with a core model. Based on a core model (i.e., a metamodel), EMF is then able to generate source code to support the core functionality of a modeling environment (see Figure 2). In fact, since core models are platform independent, it is necessary to add platform specific information regarding code generation in what is called a genmodel. A genmodel is basically a decorator of the core model with details regarding code generation. EMF can then be used to generate code that supports the creation of models that conform to the core metamodel and a tree-based visual modeling editor. Therefore, EMF provides the core functionalities of a metamodeling tool capable of generating domain specific modelers.

Figure 3 presents and example of a possible metamodel for database schemas. Used as input to a metamodeling toolset such as EMF this metamodel could be used to generate a database schema modeler. Using this generated modeler, database schemas that conform to database schema metamodel can be created. However, these database schemas represent also metadata since they specify the structure of the objects that compose databases that conform to that database schema (e.g., tables, columns and foreign keys). Therefore, a database schema can be seen as a metamodel of a database instance (in Figure 1, a database schema model is a metamodel of database objects) and it should be possible to generate code to support database instantiation based on a database schema model. Although this is true, EMF (and, to our knowledge, other metamodeling tools) does not directly support it since a domain model is not a core model and the generative capabilities of EMF can only be applied to core models.

The approach we present and discuss in this paper regarding multi-staged domain specific model driven engineering is *inspired* on the identified *restriction* of metamodeling tools. To tackle this restriction, we propose the *promotion* of domain models to *native* metamodels of the metamodeling tool (core models in the case of EMF). The approach is depicted in gray in Figure 2.

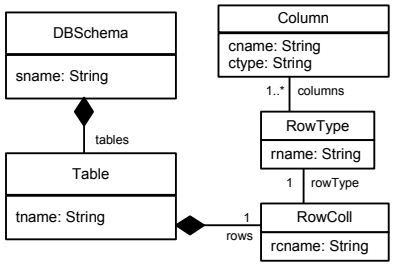

Fig. 3. Possible metamodel for a database schema.

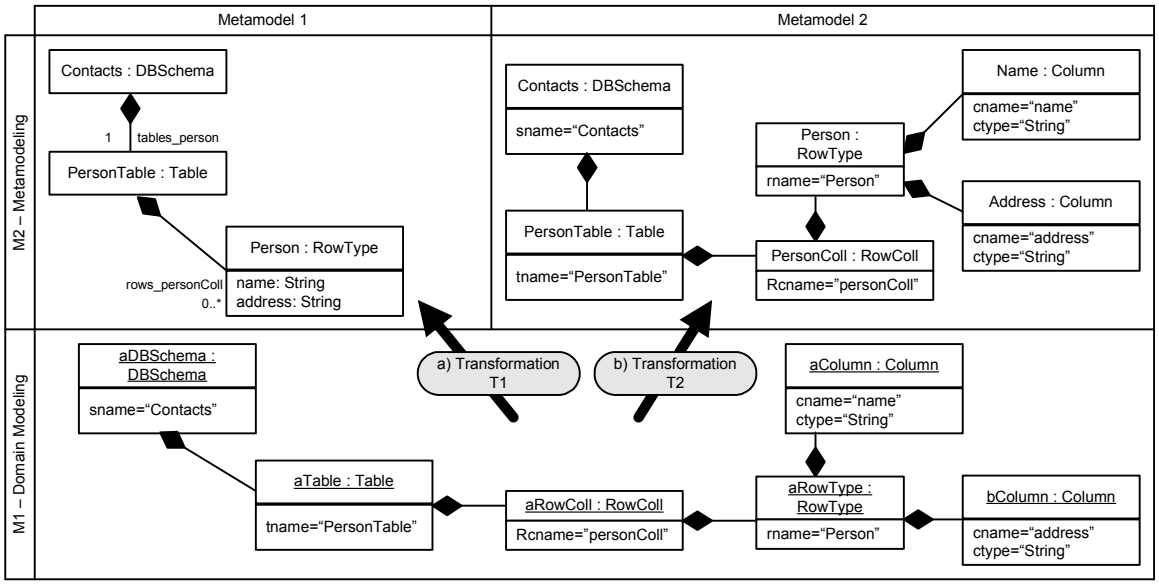

Fig. 4 Possible approaches for promoting a domain model into a metamodel.

In Figure 4, we show a simplified example of how a promotion approach could be applied in the case of our database schema modeling example. In the lower half of the figure we can see a domain model representing the schema for a contacts database. The idea behind the promotion approach is to transform the domain modeling elements with meta semantics to the corresponding metamodeling elements. For instance, the domain element *aTable* which models a table of persons gives origin to the class *PersonTable* at the meta level. We can say that *we are specializing the original metamodel by example*, since the source for its specialization is a concrete model that conforms to the metamodel we are specializing. There are, however, two perspectives in this kind of transformation: a) an *instantiation* perspective, where the goal of the resulting metamodel is to support instantiations of the modeled concepts (transformation T1 and metamodel 1 in Figure 4); b) a *specialization* perspective, where the goal of the resulting metamodel is to support further specializations of the modeled concepts (transformation T2 and metamodel 2 in Figure 4).

3. Multi-Staged Domain Modeling Approach

We will illustrate our approach to multi-stage modeling with an example based on a case study developed at I2S, a Portuguese software house specialized in the development of software for insurance companies. The software that the company develops is used in what is a typical scenario for multi-staged modeling. Insurance agreements, which represent agreements between insurers and their customers, are a core concept in insurance. These agreements are commonly known as insurance policies. The structure and rules that govern these agreements can be specialized at several stages, e.g., insurance company headquarters, division, branch, or agent. Figure 5 presents how our approach can be applied to support multi-stage modeling of insurance agreements.

The stages presented in Figure 5 represent players of an insurance business. In the figure we can see the insurance company, an insurance company division and an insurance company branch. Each of the stages runs the same domain-specific platform [5], in this case, an insurance information system platform. The domain-specific platform can be configured for a particular purpose through domain-specific modeling. Domain-specific modeling is done by a domain-expert. In this case, insurance agreements are modeled and used to configure the domain-specific platform. As Figure 5 shows, agreement models can be specialized in succeeding modeling stages. The concepts that are global and common to all the stages are represented by the domain model *Insurance.ecore*.

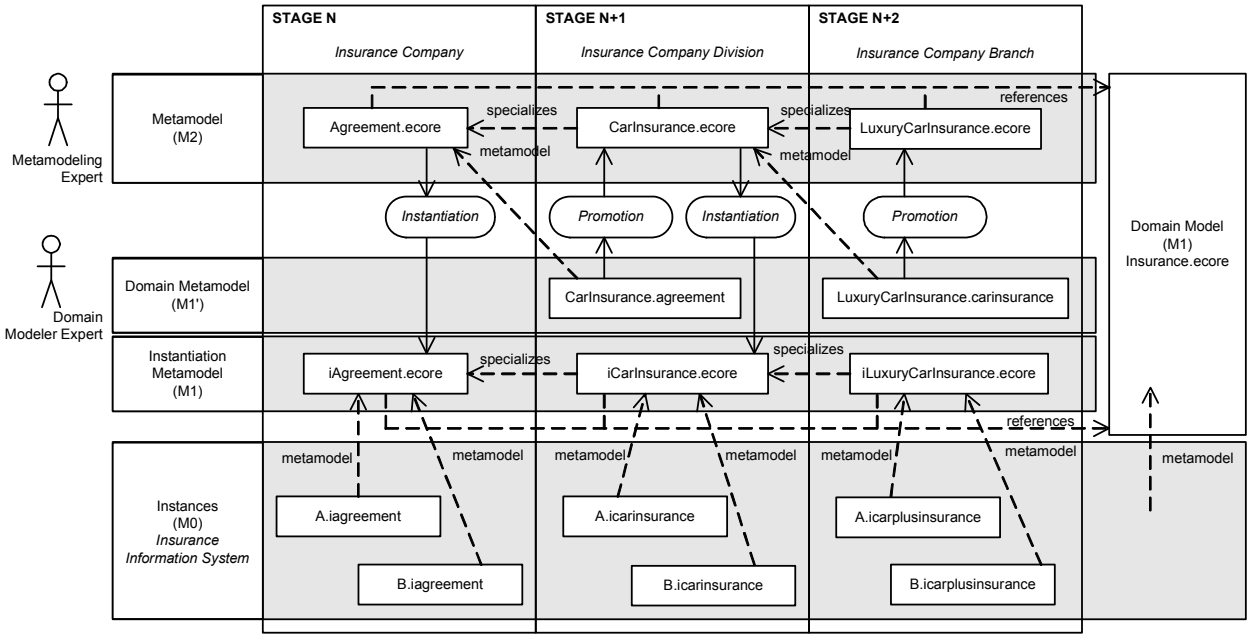

Fig. 5. Multi-staged modeling of insurance supply chain with EMF.

The first metamodel (*Agreement.ecore* in the M2 layer of stage N) is used to specify a specialized modeling environment, i.e., a domain-specific modeling environment. With this environment, the domain modeler at the M1 layer no longer needs to use *Ecore* abstractions, it can use specific abstractions of the domain. In this case, at the M1 layer, the domain modeler specifies insurance agreement models. We can see how this is achieved at stage N+1 and N+2 with *CarInsurance.agreement* and *LuxuryCarInsurance.carinsurance*. As we have discussed in the previous section, these domain models can be *promoted* to metamodels so that the generative capabilities of the metamodeling tool can be used to support two perspectives: the *specialization* of the domain models and the *instantiations* of the domain model.

To explain the multi-staged model driven scenario it is important to understand the involved roles. There are basically two human roles: metamodeler and domain modeler. The metamodeler uses the metamodeling framework directly. In the case of EMF, the metamodeler edits directly core models. A domain modeler is someone that edits domain-specific models, usually according to a metamodel that is specified by a metamodeler. Obviously, it is also necessary to have a platform to execute the modeled concepts. In Figure 5 this platform is the *Insurance Information System*, which is a domain-specific platform. In Figure 5, it is possible to observe the responsibility of these three roles for a multi-staged modeling approach: the metamodeler has the responsibilities at the M2 layer; the domain modeler has responsibilities at the M1 layer; and the domain-specific platform at the M0 layer. In fact, the M1 layer is divided into M1 and M1'. The M1' layer is where the domain modeling takes place. The domain model of a specific stage (at the M1' layer) is used to generate the metamodel of that stage by using the *promotion* transformation. For instance, the *CarInsurance.agreement* domain model is transformed into the *CarInsurance.ecore* metamodel. Following the discussion of the previous section, each metamodel has two perspectives: specialization and instantiation. The instantiation perspective is *generated* by the *instantiation* transformation, and the result is the *iCarInsurance.ecore* that is used as metamodel for instances at the corresponding stage. The original metamodel of the stage (in this case, *CarInsurance.ecore*) is used as metamodel to generate the modeling environment of the next stage. Details about these two perspectives and the transformations involved will be given in the next section.

Since the approach is based on domain-specific models and those require a domain-specific modeling environment, the process *bootstrapping* is done by the metamodeler. The first metamodel (core model) is used to introduce the domain-specific modeling concepts that will be used by domain-experts in all the stages to create or specialize domain-specific models (in this case, insurance agreements). Such metamodel will provide domain modelers their modeling concepts in a way similar to the concepts that EMF provides to the

metamodeler. As a *bootstrapping* metamodel, it must also integrate with the concepts of the domain-specific platform (which are modeled in the domain model *Insurance.ecore*). Figure 6 depicts the models at the *bootstrapping* of the multi-staged model driven approach for the software insurance supply-chain case study: *Agreement.ecore* which is the *specialization* metamodel; *Insurance.ecore* which is the domain model; and *iInsurance.ecore* which is the *instantiation* metamodel. The *bootstapping* process corresponds to the first stage (*Stage N*) depicted in Figure 5. Figure 6 serves also as a good example to illustrate the *specialization* and *instantiation* perspectives that we can obtain from a metamodel. Some elements of the specialization metamodel are annotated (the annotations are depicted in the diagram in a similar way to UML stereotypes). If we take a closer look at the specialization metamodel of Figure 6, we see that only *AgreementRoot* and *Agreement* are not annotated and that one instance of the former must contain exactly only one instance of the latter. In our approach, we use annotations to mark the semantic of the elements at the domain metamodeling level (M1' in Figure 5). For instance, according to the annotations of the *Action* element in Figure 6, the inclusion of such element by a domain expert in a domain model corresponds to adding the semantics of an *EOperation* at the meta level. This means that in the resulting instantiation model an *operation* should be generated. The use of annotations to mark metamodel elements enables the association of meta semantics to metamodels without requiring an intrusive modification on them. Therefore, the approach can be non-intrusively applied to whatever metamodel. Only non-annotated elements and annotated elements contained by references with *refines* or *subsets* annotations are included in the instantiation model. We will further detail this transformation in the next section.

In Figure 5 and Figure 6, it is possible to observe the possible relationships between the models that result from the modeling activities and the domain model. This domain model represents the concepts that support the domain specific platform (in our example, the *Insurance Information System*). These concepts are available at every stage of the supply chain and, to a certain extent, represent the commonality in the product line.

At each stage, the *instantiation* activity generates the instantiation model from the metamodel. The metamodels are also used to generate the domain modeling environments of the next stages. These environments are used by the domain modelers and the resulting domain models are *promoted* to metamodels that are specializations of the previous ones. This process can be repeated to support subsequent modeling stages.

4. Transformation Patterns

In this section we present our approach to multi-stage model driven software development. Since a multi-stage model driven approach can be applied in several scenarios (being the insurance software supply chain only one of them) we explain our approach as a set of model driven development patterns. We follow the spirit of the original description of design patterns and we describe here the problem and the proposed solution of each model driven pattern. The consequences of the patterns are discussed on Section 5. Each presented pattern is a part of a more large-scale pattern that we call *Multi-Stage Domain Specific Modeling*. If we continue to make the analogy with traditional development patterns we could say that this is an architectural style pattern [6].

The Problem
How to support a multi-staged domain-specific modeling approach with model specialization using current metamodeling tools and in the context of a domain-specific platform.

The Solution
The idea behind the proposed approach is that the domain models will be used in two perspectives: to support instances of modeled concepts at any given stage (*instantiation* perspective) and to support the specialization of concepts at the next stage (*specialization* perspective). The proposed solution adopts *off-the-shelf* metamodeling tools. By this we mean that the solution is essentially based on existing generative and transformational support of publicly available metamodeling tools. Eclipse EMF is one example of such a metamodeling tool. To support the multi-stage model driven approach we propose that the models of the native metamodel format be annotated in a manner that marks their elements as being *instance* elements (*instantiation* perspective) or *meta* elements (*specialization* perspective). Such annotations can then be used to guide two transformation activities: the *instantiation* transformation and the *promotion* transformation.

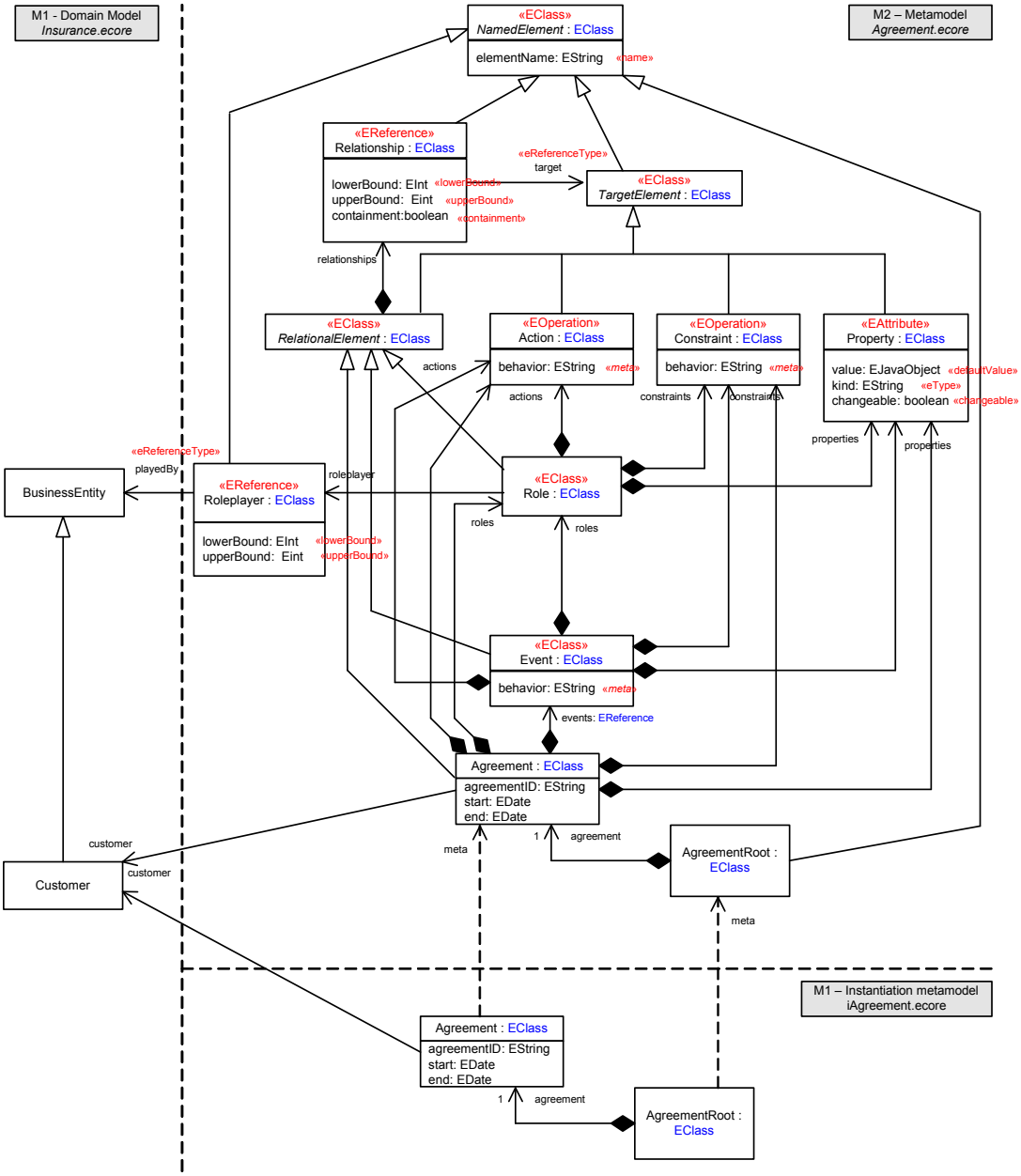

Fig. 6. Metamodel, instantiation metamodel and domain model.

Figure 5 presents an illustration of this pattern for an insurance supply chain. The instantiation transformation uses the elements of the metamodel that are part of the instantiation perspective to generate another metamodel adapted to support instances of the modeled concepts. Details of this transformation will be given next. The promotion transformation *interprets* the domain metamodel as a *specialization by example* of the metamodel of the previous stage. Using the annotations of the previous metamodel, it promotes the domain metamodel into a native metamodel that specializes the previous metamodel. Therefore, specialization is achieved by the domain metamodel. The process can be repeated to support further stages: the resulting metamodel can be used to generate the instantiation model of the current stage as well as supporting the generation of the domain specific modeling environment for the next stage. Therefore, the metamodeling tool is reused across all stages and each stage has a generated domain-specific modeling environment.

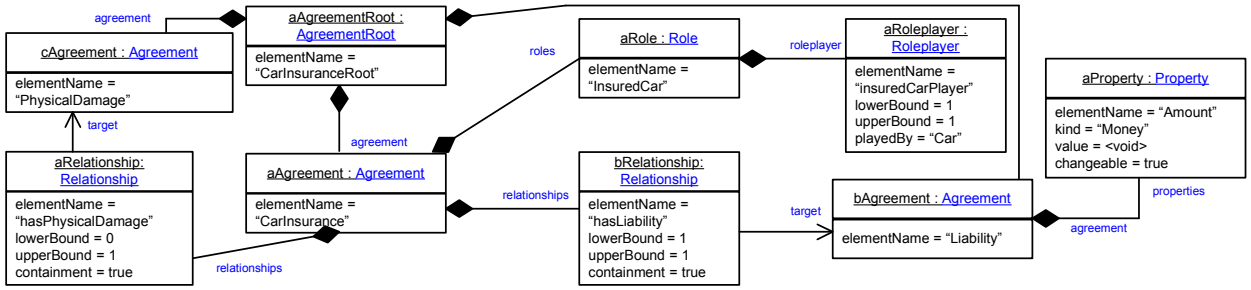

Fig. 7. Domain metamodel for a car insurance agreement (*CarInsurance.agreement*).

4.1 Promotion Transformation Pattern

Problem

How to support the specialization of domain models that are not native metamodels of the metamodeling tool and, therefore, do not have the native support for specialization.

Solution

We propose a solution that is based on the promotion of the domain model to a native metamodel of the metamoldeling tool in a way that preserves the semantics of the domain model. Therefore, the domain model can be seen as a *domain metamodel*. We call this transformation a promotion because we are transforming a model into a metamodel, i.e., we are *promoting* a model into a metamodel. Our solution is proposed in the context of the multi-stage model driven pattern and therefore, in conformance with the other patterns involved we use annotations to guide the transformation process.

Figure 7 presents an example of a domain model (in fact it is *acting* as a metamodel) which metamodel is the one presented in Figure 6 (*M2 metamodel*). The result of applying the *Promotion* transformation to the model of Figure 7 results in the native metamodel of Figure 8. Basically, each object of the domain model becomes a *Class* (*EClass*) in the native metamodel. Each reference instance (or *link*) becomes a *Reference* (*EReference*) in the native metamodel. As such, we can say that the *Promotion* transformation is composed of mainly two sub-transformations: *Object* to *Class* and *Reference Instance* to *Reference*.

The goal is to transform domain models into their equivalent native metamodels. A domain model is an instance of a metamodel, and as such is composed of *objects* and *links* or reference instances between objects. The *objects* are instances of *Class* elements (*EClass*) of the metamodel. The *links* are instances of *Reference* elements (*EReference*) of the metamodel. When the domain modeler creates an instance of a *Class* he/she is making a *specialization* of the *Class*. As such, in the *Object* to *Class* transformation, an object is transformed into a *Class* that must specialize (become a subtype of) the meta-class of the source object. For instance, the *aRole* object of Figure 7 becomes the *InsuredCar EClass* in Figure 8. The *InsuredCar EClass* is a specialization of the *Role EClass*, which is the meta-class of the *aRole* object. This is the basic principle regarding the promotion of objects to classes. However, some details must be taken care. For instance, what is the name for the new generated classes or what to do with the values of the objects fields? Once again, we adopt annotations to solve these issues. If we take a look at the original metamodel from Figure 6, we see that the *elementName* field of the *NamedElement* class is annotated with «name». The *Promotion* transformation uses this annotation to select the value it will use for the name of the generated classes. Regarding the values of the objects fields, they are used as default values in the new generated classes (see Figure 8).

Links also follow an approach similar to that of the objects. As we have mentioned, they become references in the resulting metamodel. But, because they are instances of references, they are annotated as *subsets* or *refines* of the original reference. For instance, the *roles* link of Figure 7 that links *aAgreement* and *aRole* becomes the *insuredCar* reference between *CarInsurance* and *InsuredCar* target elements (see Figure 8). This reference is annotated as being a *subset* of the *roles* reference of the metamodel of the previous stage. We use the terms *subsets* and *refines* with similar semantics as the ones used in the UML language. Since the metamodeling tool (in this case EMF) is not aware of these annotations, it is necessary to extend/adapt it so that it will generate code according to the annotations in the metamodel. In the case of EMF, because of its extensible architecture, it is simple a matter of developing JET [7] templates that add the necessary extensions to the generated code. Because this is relatively straightforward we do not detail any further this necessary activity.

4.2 Instantiation Transformation Pattern

Problem
How to support instantiation from domain metamodels that are not native metamodels of the metamodeling tool and, therefore, are not directly usable by metamodeling tools as sources for generating instantiation supporting code.

Solution
The solution presented here is part of the multi-stage model driven pattern. Therefore, it depends on the transformation presented in the previous section that enables the promotion of a model to the metamodel level. The sequence of transformations is depicted in Figure 5. In the figure it is possible to observe that the instantiation model is obtained from the metamodel that results from the *Promotion* transformation (except for the *bootstrap* stage). In the instantiation metamodel the goal is to have only the elements that resulted from the elements with *meta semantics* of the previous stage that were specialized. For instance, in Figure 6 we see that the instantiation model only contains elements that do not have annotations denoting their *meta semantics*. In Figure 8, we see that the metamodel that resulted from the *Promotion* has several *refines* and *subsets* annotations. These indicate elements that were specialized. Therefore, these elements are included in the instantiation metamodel of the same modeling stage (see Figure 9).

Figure 8 presents the specialization metamodel at stage N+1 of the insurance case study. This specialization metamodel was obtained from the domain metamodel depicted in Figure 7. Figure 9 presents the output of the instantiation transformation, when the source metamodel is the one depicted in Figure 8. As it is possible to observe from both figures, the resulting instantiation metamodel not only contains elements that are not annotated as meta elements in the source metamodel but also contains the annotated elements that resulted from the specialization process. These elements are those that *subset* or *refine* meta annotated elements of the previous stage. For instance, in Figure 8 we can see that the *hasLiability* relationship between *CarInsurance* and the *HasLiability EClass* subsets the *relationships* relationship of the previous stage. Therefore, *HasLiability* is included in the resulting instantiation metamodel. Since this element has an annotation stating that it has the meta semantics of an *EReference*, it becomes an *EReference* element in the resulting metamodel.

The solution proposed for this pattern is straightforward if we consider it only in the context of single-stage development. When we consider it in a multi-stage approach we have to take into account the refinements (specializations) made in the previous stage. The annotations in the source elements regarding such refinements as well as their meta semantics can guide the creation of the instantiation model. As it is possible to observe in the previous examples, such annotations are done using the names of the elements of the meta-metamodel of the modeling tool (or the *native* metamodel). In the presented examples we use ecore, the meta-metamodel of EMF. From Figure 6, we see the original *intention* of the metamodeler regarding the elements and their meta semantics at the domain metamodeling level: the non-abstract elements *Event* and *Role* should have a meta semantic of an *EClass*; the non-abstract elements *Action* and *Constraint* should have a meta semantic of an *EOperation*; the non-abstract elements *Roleplayer* and *Relationship* should have a meta semantic of an *EReference*; and the non-abstract element *Property* should have a meta semantic of an *EAttribute*. These examples represent the four most typical instantiation transformations: *Class* to *Attribute*; *Class* to *Operation*; *Class* to *Reference* and *Class* to *Class*.

In Figure 9 we can see the result of applying these transformations to the source metamodel of Figure 8. For instance, if we take the example of the source element *Amount*, we see that this element is annotated as having the meta semantics of an *EAttribute*. As such, if transformed, it must become an *EAttribute* element in the target metamodel. Other possible annotations in the source element may be used to further specify the value of target element attributes. For instance, regarding *Amount*, we see that in its ancestor element *Property* the field *kind* is annotated as *eType* (see Figure 6). Therefore, the value of this field is used in the instantiation transformation as the value of the field *eType* for the *EAttribute* that resulted from *Amount*.

Similarly to what was said regarding the promotion transformation, the instantiation transformation also requires adaptations to the generative infrastructure of the metamodeling tool. This is required so that the generated code supports the semantics of the proposed annotations. This support can be added in the same way as presented in the previous section.

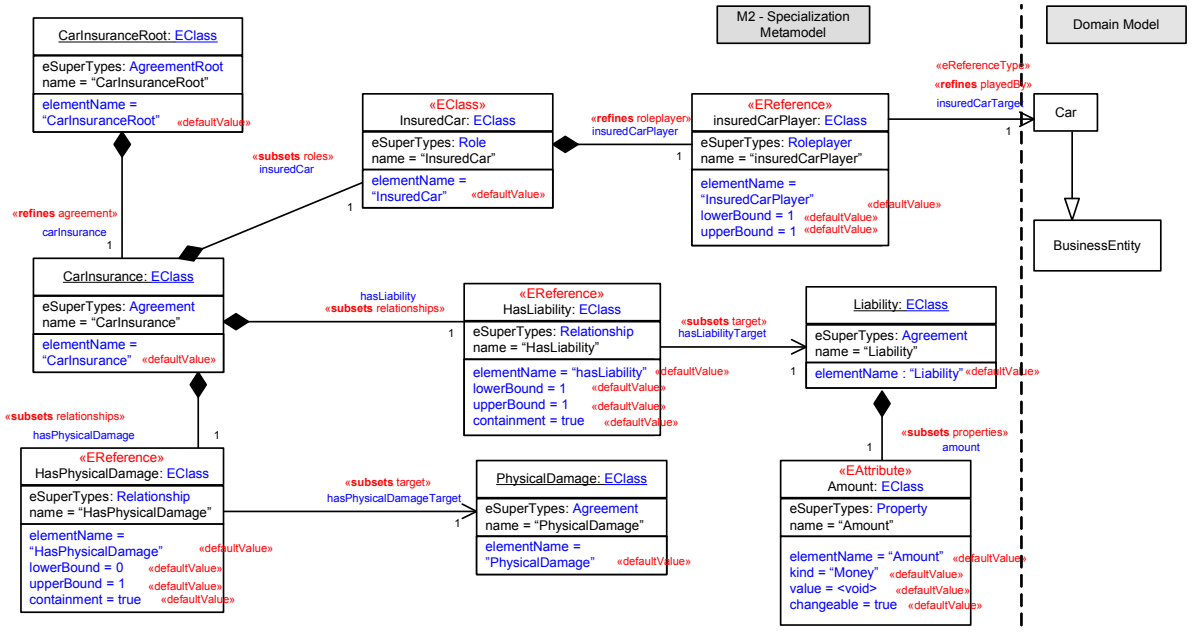

Fig. 8. Native metamodel for a car insurance agreement (*CarInsurance.ecore*).

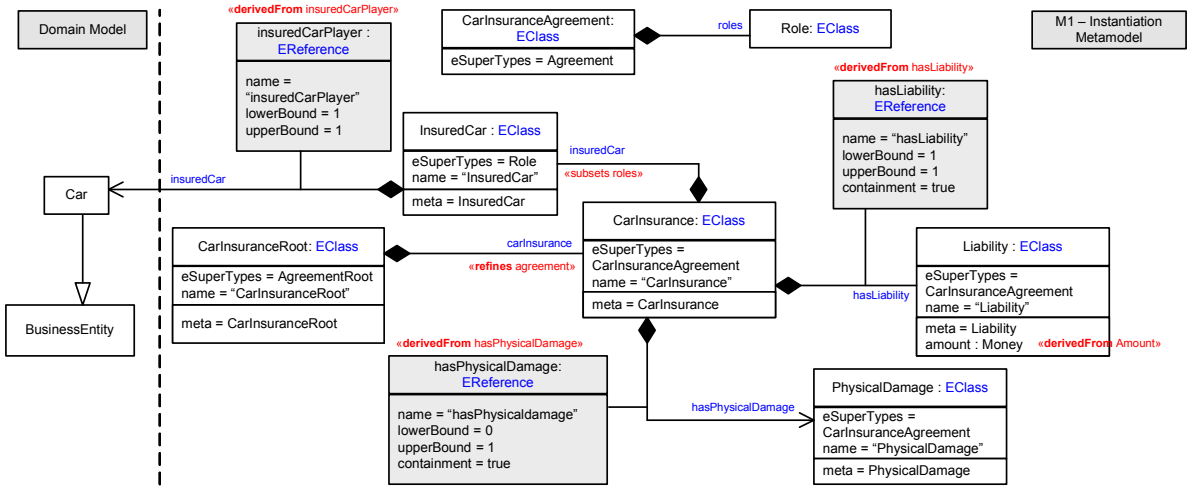

Fig. 9. Instantiation metamodel for a car insurance agreement (*ICarInsurance.ecore*).

5. Discussion

The case study we have presented was based on experimental development made at the I2S company. The actual solution running in the company is not truly multi-staged in the sense we have presented here. It is based on a domain modeling tool that was generated using EMF (and GMF for the graphical part). However, these are the only similarities with the solution proposed in this paper: the code that supports instantiation from the domain metamodel does not reuse the EMF generative capabilities and domain metamodels are not truly specialized between stages. Since the domain modeling tool is the same for all stages, model *templates* are used in each stage as a starting point for modeling. In the approach presented in this paper each stage has its own domain specific modeling environment. The solution that is actually running in the company does not escalate well since the natural constraints that result from the specialization of the metamodels in the approach presented in this paper have to be hard coded into the domain modeling tool. Since the results from the experimental developments with our approach were

considered by the company as very promising a prototype is been developed. In this prototype we will also tackle details that were not discussed in this paper like the integration of GMF and the support for the specialization of model constructs such as OCL expressions. For the moment, model transformations have been implemented with SmartQVT [8].

Multi-staged software development approaches are not a new concept, particularly in the context of product lines and software factories [9]. Czarnecki *et al.* have discussed the subject and presented approaches to support multi-staged modeling [10]. However, the focus was on feature modeling and, therefore, their approach is not as generic as the one presented in this paper. The core of the approach we have presented in this paper is based on promoting models to metamodels. We had already started to tackle this topic in a previous work [11]. In that work, we presented how a feature model could be promoted to an EMF metamodel in order to support feature configurations. The approach was, nevertheless, limited to feature models. However, that work and a discussion in the EMF newsgroup [12] have inspired us to further investigate the *model promotion* idea.

Promoting models to metamodels is an approach already in use. For instance, in KerMeta, an action metamodel is composed with EMOF [2] to obtain an executable EMOF metamodel at the M2 level [13]. This metamodel is then promoted to the M3 meta-meta level. The UML2 project [14] also adopts a *promotion* like approach to transform UML class diagrams to ecore models. Our approach extends these ones since we propose a generic support for multi-staged modeling. The *specialization promotion* and *instantiation* transformations that support our approach can be generically applied to whatever metamodels since the approach is not intrusive, it does not require modification of the metamodels. Regarding this aspect, our approach is based on annotating the metamodels. These annotations *guide* how and where the transformation patterns are applied. We can say that the approach has some similarities with the way EMF uses *genmodels* to support code generation.

6. Conclusions

In this paper, we have presented and discussed a problem that is common in software product lines and software factories: multi-staged model driven development. We have described our approach to multi-staged model driven development following the spirit of architectural and design patterns. The patterns presented in this paper show how domain models can be specialized at several stages by reusing as much as possible the functionalities of actual publicly available metamodeling tools and their generative capabilities. We have illustrated the problem and the patterns using EMF concepts and a case study from the insurance domain. However, as discussed in the paper, our approach can be generically applied to any metamodeling tool that supports annotations..

We believe that, as metamodeling tools mature and become more widely adopted also model driven patterns, like the one discussed in this paper, will become widely adopted and eventually incorporated into the metamodeling tools.

7. References

[1]MDA, "Model Driven Architecture Guide Version 1.0.1," vol. 2007: OMG, 2007, http://www.omg.org.

[2]MOF, "Meta Object Facility (MOF) 2.0 Core Specification (formal/06-01-01)," vol. 2006: OMG, 2006, http://www.omg.org.

[3]E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[4]EMF, "Eclipse Modeling Framework," vol. 2007: Eclipse Foundation, 2007, http://www.eclipse.org/emf/.

[5]K. Czarnecki, M. Antkiewicz, and C. H. P. Kim, "Multi-level Customization in Application Engineering," *Communications of the ACM*, vol. 49, 2006.

[6]M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*: Prentice Hall Publishing, 1996.

[7]JET, "Eclipse JET - Java Emitter Templates," vol. 2007: Eclipse Foundation, 2007, http://www.eclipse.org/emft/projects/jet/.

[8]SmartQVT, "SmartQVT - Open Source Transformation Tool Implementing the MOF 2.0 QVT-Operational Language," vol. 2007: France Telecom, 2007, http://smartqvt.elibel.tm.fr/.

[9]J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*: Wiley, 2004.

[10]K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models," *Software Process Improvement and Practice, special issue on "Software Variability: Process and Management"*, vol. 10, pp. 143-169, 2005.

[11]A. Braganca and R. J. Machado, "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines," SPLC 2007, Kyoto, Japan, 2007.

[12]E. Merks and *Others*, "Making EMF models valid Ecore models for a two-level code generation," eclipse.tools.emf newsgroup thread, 2006, http://dev.eclipse.org/newslists/news.eclipse.tools.emf/msg20713.html.

[13]P.-A. Muller, F. Fleurey, and J.-M. Jezequel, "Weaving Executability into Object-Oriented Meta-Languages," Models2005, Jamaica, 2005.

[14]UML2, "UML2 - Model Development Tools (MDT)," Eclipse Foundation, 2007, http://www.eclipse.org/modeling/mdt/?project=uml2.