Transforming System Operations’ Interactions into a Design Class Diagram

Andrés Vignaga
Departamento de Ciencias de la Computación
FCFM, Universidad de Chile
Blanco Encalada 2120, Santiago, Chile
avignaga@dcc.uchile.cl

María Cecilia Bastarrica
Departamento de Ciencias de la Computación
FCFM, Universidad de Chile
Blanco Encalada 2120, Santiago, Chile
cecilia@dcc.uchile.cl

ABSTRACT
This work reports the results of the development of a model transformation realized in Kermeta. This transformation accepts a Domain Model and a set of Communication Diagrams containing the design of system operations corresponding to the same use-case. The result is a Design Class Diagram expressing the class structure that part of a system should logically exhibit in order to enable the occurrence of the interactions specified in the Communication Diagrams. To that end, along with specific meta-models for input models and for the output model, an executable model for the transformation was defined. This paper reviews the elements involved in the development of this transformation, and describes the design of its structure and behavior. The transformation was applied to a case study taken from the bibliography and the results are presented and discussed.

Categories and Subject Descriptors
D.3.2 [Language Classifications]: Specialized Applications Languages—model transformation languages; K.6.3 [Management of Computing and Information Systems]: Software Management—software development, software process

General Terms
Languages, Design

Keywords
Model Transformation, MDE, Kermeta, Metamodeling

1. INTRODUCTION
Model Driven Engineering (MDE) is an approach to software development where the notion of model is promoted to a first class construct, and constitutes a primary artifact at all stages of software development. Model management is thus becoming a core discipline. In this context, model transformations play a central role because information contained in models can be effectively managed and traced throughout the development process, enabling models to fulfill their purpose. A development process can be understood as a partially ordered set of model transformations [4]. Transformations aim at generating new models in an ideally automated fashion, where an otherwise manual approach could result complex and error prone. Concrete transformation applications address the question of how good model transformations are in software engineering practice, and they are useful not only in actual development processes but also they constitute a valuable asset for the improvement of both the techniques for transformation development and the tools that support them [9].

The Unified Process [3] is a widely accepted development process in which the notion of model is centric, and it is use-case driven. Particularly, design artifacts are contained in the Design Model, which is organized according to collaborations (use-case realizations). A collaboration of this kind is a static structure consisting of a set of instances that participate in the use-case realization. The behavior of the collaboration is specified as a set of interactions between those instances, typically showing the flow of messages needed to fulfill the functionality expected for every system operation within the use-case. In terms of Larman [8], interactions should be expressed by means of Communication Diagrams. In addition, a Design Class Diagram is a class model from which instances in the collaboration are typed, and is also the main input for the code generation stage. Larman’s proposal for collaboration design provides concrete techniques for UP’s Analysis & Design discipline, and suggests defining the behavior of the collaboration first (i.e. the interactions), possibly using a Domain Model as an inspiration, and then the Design Class Diagram.

In this paper we propose an implementation for this particular model transformation. It accepts as input a set of interactions in the form of Communication Diagrams, each of them defining the behavior of a system operation within a use-case, and a class diagram specifying the Domain Model used in their definition. Its output is a class diagram specifying the classes, and their relationships, that enables the interactions used as input. This artifact is an important part of the Design Model. In [8], its generation is intended to be manual, where only a high level description of the process and some hints are provided. We believe that the problem tackled by this transformation is not trivial.
The remainder of this document is organized as follows. Section 2 introduces the meta-modeling approach for model transformation, upon which our proposal is based; presents Kermeta which is the environment where the transformation was developed, and discusses the assets involved in the implementation. In Section 3, the details of the transformation are summarized and analyzed. The results of its application to a case study are reported in Section 4. Section 5 concludes.

2. APPROACH AND TOOLS

Model transformations can be classified in a number of categories depending on the techniques and tools used in their development [5]. Examples of these include general purpose programming languages, such as Java or C#, and transformation specific languages, such as QVT [10] or ATL [7]. Particular languages can be further classified in imperative, declarative or hybrid [1]. Another category of model transformations is characterized by the use of meta-modeling tools.

2.1 Transformations based on Meta-Modeling

In this approach, a transformation is a model of an object-oriented program that transforms models. The application of the transformation is, therefore, the execution of such model. The languages for expressing the transformation and the meta-models of the models to be transformed are the same. Elements in models and objects in the transformation are at the same level, and thus manipulated homogeneously. This approach has the benefit of making no distinction between transformation development and software development [6], so best practices of object-oriented development can be applied to transformations as well.

2.2 Kermeta

Kermeta [2] is a domain specific language (DSL) for metamodeling engineering, which adheres to the approach described above. It includes features from different functional domains of current model-driven technologies. Kermeta supports the definition of meta-data by being an extension of Essential MOF (EMOF). Its support for OCL allows for navigating and querying models. It provides an imperative and object-oriented executable action language for specifying methods in meta-models and expressing their operational semantics. Kermeta does not provide specific constructs for model transformations. However, its full compatibility with ECore, a variant of EMOF introduced by the Eclipse Modeling Framework, allows manipulating models from repositories, making Kermeta suitable for modeling transformations of MOF-based models, and particularly UML models. Kermeta is supported by a set of tools distributed as an Eclipse plug-in, which include an editor, a parser, a type-checker, an interpreter and a debugger.

2.3 Elements of the Transformation

The framework proposed by Kermeta for transformation definition can be understood in three levels. The language for meta-model and transformation specification is found at the top level. At the intermediate level, the transformation and meta-models for its input and output are specified. Source models, an instance of the transformation (its execution), and the result are placed at the bottom level. In our case, Kermeta and ECore are the languages at the top level.

Figure 1: Static structure of the transformation

A customization of the UML meta-model and our transformation complete the intermediate level. Finally, sample models and the corresponding result, all expressed in XMI, are comprised in the bottom level.

In our transformation two versions of the meta-models coexist, one expressed in ECore and the other in Kermeta. The Kermeta version specifies meta-models’ structure and behavior. Meta-model behavior enables reusable model and model element manipulation capabilities, including queries, modifications, and even more complex logic. The ECore version can only specify structure and is redundant with the Kermeta version; it is necessary however for model serialization purposes.

UML meta-model includes more detail and complexity than required for our purposes, hence a tailored version with the usual features and some minor extensions was used instead. Class Diagrams include support for classes and data types with typed attributes (such types are data types only); classes can also contain typed operations. Available relationships are generalizations, dependencies, and binary associations with multiplicities and navigabilities. Design Class Diagrams and Domain Models conform to this meta-model. Communication Diagrams include named objects (both single and multiobjects), and named and typed messages between objects, with a sequence number. We believe that these constructs would suffice for most practical applications. Additionally, messages can be annotated with visibility and resolution information, which is discussed later. We refer to [11] for a detailed description of the meta-models.

3. TRANSFORMATION

In this section the details of the implemented transformation are summarized and analyzed. The complete source code is available in [11].

3.1 A Model for the Transformation

From a logical point of view, the transformation is organized in two packages containing Kermeta classes, Transformations and Metamodels. Transformations package embodies the logic of the transformation and consists of two
3.2 Transformation Steps

Following the algorithm proposed in [8], the transformation generates a Design Class Diagram by sequentially adding new elements, or modifying existing ones, on an initially empty class diagram. The concrete steps performed by the transformation are described next. Each step assumes that previous steps were successfully performed. As a general procedure, the target model is navigated before the addition of an element to prevent duplicates. Criteria for element equality are examples of behavior actually included in the meta-models. They will be discussed when not apparent.

3.2.1 Adding Data Types

Data types are assumed to be fully specified in the Domain Model, therefore they are copied from that source. Since they are used for typing attributes, including those in data types, the copy is deep.

3.2.2 Adding Classes

A class is created for every possible type of object in any interaction. For example, the occurrence of an object of type `Sale` generates a class with that name. When a class has a corresponding element in the Domain Model, instantiability is derived from that element, and the parent class (if applicable) is recursively created. Collections motivated by multiobjects are not explicitly represented.

3.2.3 Adding Attributes

Class attributes can be derived from the state of objects of that class. For example from an object of class `Sale` with state `[date = 1/31/2001, time = 2:43pm and isComplete = false]`, it can be derived that such class needs attributes `date`, `time` and `isComplete`. Since in most practical situations it is not likely that object state is specified, it is not implemented in this version of the transformation.

Interaction diagrams lack a mechanism for specifying the attributes used for resolving a message, therefore they need to be inferred. Messages accessing the value of an attribute usually follow a pattern in their names (prepended with `get` or `set`). If such message is locally resolved by the target object, known via the message's resolution property, it suggests an attribute for the object's class. Otherwise, a derivable attribute, or no attribute at all, is created. The type of such attribute is obtained from the return type of the `get` message or from the type of the argument of the `set` message, in both cases, if present. Additionally, if a class has a corresponding element in the Domain Model, attributes of that element can be copied to the class.

3.2.4 Adding Operations

Class operations can be derived from the messages that any instance of that class receives. For example, a sale receiving a message `makeLineItem(spec, qty)`, will generate an operation named `makeLineItem` with two parameters in class `Sale`: `spec` of class `ProductSpecification` and `qty` of type `Integer`. Parameter types are obtained from the type of the message's arguments, if present. Return type is got from the message's information as well. Two operations are considered equal not only when their name match, but also when the parameters' number and type match in order.

If a message is locally resolved, or has nested messages (known via the sequence numbers), then the corresponding operation will be concrete; otherwise it will be abstract. A class with an abstract operation is specified as abstract. Messages for object creation and attribute access are not included in the result. Additionally, since collections are not represented, messages to multiobjects are ignored and assumed as part of the class that implements the collection.

3.2.5 Adding Associations

An association between two classes is created when there exists at least one message flowing between instances of those classes, and the message's visibility is `association`. The association is at least navigable in the direction of the message. If the association has a corresponding element in the Domain Model, the name and multiplicities can be borrowed from it. Two associations are considered equal when their name match, and when the classes they relate (in any order) also do. Optionality in association names requires that at most one association between two classes could be defined, or that role names must be present.

If an association does not have a corresponding element in the Domain Model, its name is given by the concatenation of the names of the classes it relates. The default value of a multiplicity is `'0..1'`; if needed, the minimum can be set to `1` and the maximum to `*`. In fact, if the target object is a multiobject, then the final value at the navigable end is `'0..*'`. This criterion leads in some cases to a fixed set of possible multiplicity values; we believe, however, that it still provides flexibility in practice.
3.2.6 Refining Multiplicities

Part of the refinement of the default value of multiplicities, as described above, was performed in the previous step for the maximum, based on information local to one message at a time. This step refines all minimums which are 0; any of them could be set to 1 or left unchanged. Such change is carried out when a mandatory link is detected. If the corresponding association end is not navigable, then the multiplicity is irrelevant and should be omitted; otherwise, at least one message flows in that direction. If an object of the same class as the target was never created, received as an argument, or received as a result of a message, it suggests that the link between the source and the target is mandatory, and therefore the multiplicity needs to be updated.

3.2.7 Adding Dependencies

A dependency from a class to another is created when visibility needs to be specified, and the first class is not already related to the second by a navigable association or a generalization, which are stronger forms of dependencies. A dependency is suggested typically when an instance of the client class sends a message to an instance of the supplier class, and the visibility is local or parameter. This includes the cases when the target instance is created by the source object, or when the target is received as an argument, or even when it is the result of a message. This step deeply relies on visibility annotations. In many cases, such information cannot be derived. Since an interaction diagram cannot tell whether an object keeps a reference to another after a message is finished or not, the stability of the link may not be decided. The exception is, as discussed above, the case when the target object just “appears” in the scope of the source of the message; the link must have been established before, and thus it is stable.

3.2.8 Factorizing

At this point of the transformation, the Design Class Diagram under construction is complete; however, properties of classes may be factorized. This is specially the case when a class (most notably an abstract class) is derived and the same properties are detected in all its subclasses. Properties that can be factorized are: attributes, operations, and participation in associations. Hence, when a class for which all its subclasses own the same attribute is detected, the attribute is removed from the subclasses and added to the superclass. The case of an operation is slightly different. The operation is removed from those subclasses which do not have a method for it, it is added to the superclass and marked as concrete if it was removed at least from one subclass. In the case of an association, all subclasses must be associated to the same separate class. Those associations need to be replaced by a single association between that class and the base class. Multiplicities and navigabilities for the new association are got from the conjunction of the corresponding information in the old associations. In most of the cases that information trivially coincide; when the information diverges instead, the weakest values should be used in order to make any valid state in the old configuration still valid in the new one.

3.3 Analysis

The results produced by the transformation are syntactically correct since they constitute valid instances of the ClassDiagrams meta-model. Additionally, actions on the resulting class model are statically verified by Kermeta’s typechecker. The semantic correctness of the Design Class Diagram naturally relies on the logic of the transformation, but also on the quality and completeness of the input models.

The transformation was designed to provide all the necessary classes and relations that enable the complete occurrence of any of the interactions. However, the mechanism described in 3.2.3 requires the Domain Model to be thoroughly specified and refined. Unfortunately even in that case, the set of generated attributes for a class may not be accurate; attributes could be missing, and unnecessary ones could be included. This can be avoided by modifying the interactions’ meta-model to support object state. This implies an extra cost in the interaction specification stage, since more information needs to be included in the models.

The impact on the result of errors in the input models goes beyond this discussion, however, the impact of missing information is indeed relevant. In a Domain Model, class instantiation and type information about its attributes may be missing. The consequences of this is the same loss of information in the resulting class diagram. For an association, the name and multiplicities may be missing. As discussed, a name for an association is generated, and multiplicities are derived.

In a Communication Diagram, the name of an object is not mandatory in most circumstances; that has no impact on the class diagram. For a message, type information could be missing. The impact is on the corresponding operation parameters and the return type, which would be missing too. However, since attributes may be generated from some messages, type information for those attributes would not be generated. Moreover, operation comparison may fail, extending its impact to the factorization step. Missing type information, could also yield a loss of dependencies. As discussed above, in many cases it is not possible to correctly infer information about visibility. When missing, this could cause a loss of dependencies, and more dangerously, the situation where a dependency is created when an association was applicable, or viceversa. Finally, without information about message resolution, abstract and concrete operations would not be distinguishable.

4. CASE STUDY

The transformation just described was applied to an extended version of the case study used in [8], which deals with the use-case of processing a sale in a POS system of a retail store. The extension is such that in our case study the system handles both cash and check payments for a sale. A deeper description of the case study and the XMI files for the sample models is available in [11]. The documentation includes an overview of the Process Sale use-case, and shows the System Sequence Diagram for the main success scenario, the Domain Model and the Communication Diagrams for every system operation in that scenario.

The Design Class Diagram in the original case study and the outcome of our transformation, resulted remarkably similar. However, a close comparison showed only three differences between them. With respect to the model generated by our transformation, in the original model there is an unnecessary association, the navigability of a different association is inverted, and specific payments are missing.
This last difference naturally occurred because in the original case study our extension regarding handling different kinds of paying methods was not considered.

5. CONCLUSIONS

This paper presented a concrete implementation of a well known and core procedure of a widely accepted development process, which was never implemented as a transformation before. It is also a concrete example of the applicability of model transformations to software engineering practice. The transformation behaved as expected when applied to the case study, and allowed the detection of inaccuracies in the crafted version of the Design Class Diagram presented in [8]. We conclude that a proof-of-concept was satisfactorily developed.

The transformation formalizes the procedure proposed by Larman, and a refined description of it can be extracted from our results. The design strictly follows such procedure and is straightforward, however requiring multiple iterations over source models. The application of design pattern Visitor is not straightforward to decide which operations are general and the source-model definition and manipulation from repositories. Model navigation, and particularly target model navigation, resulted a natural mechanism for avoiding duplicate elements in the target model. Languages such as ATL do not allow target model navigation, thus sometimes cumbersome mechanisms should be implemented to achieve the same results. Navigation based on OCL expressions provides a high degree of flexibility and leads to more compact code, however, when intensively applied to an imperative context it has the effect of reducing the flow of messages between objects, as observed from a comparison with the C# version, which is characteristic of object orientation.

The correctness of the result heavily relies on input models. As expected, an effort in the previous stages leads to a more accurate and complete, and thus useful, result. The information in the input models that was identified as critical is usually available when such models are created, and sooner or later needs to be incorporated into models. In this context, providing such information to the transformation should be cost effective.

The transformation can be improved in many ways. Handling the specification of object state would enable a more accurate attribute identification. Allowing classes to participate in interactions introduce class level properties. Additionally, decisions that deserve further revision could be logged. Based on the role of the problem tackled in this work, we expect that a refined version of this transformation could be effectively applied in real projects.

6. ACKNOWLEDGMENTS

The work of Andrés Vignaga was partly funded by Comisión Nacional de Investigación Científica y Tecnológica (CONICYT) Chile.

7. REFERENCES