Transformation Models: An Application and Insights

Andrés Vignaga

Department of Computer Science, Universidad de Chile
avignaga@dcc.uchile.cl

1 Introduction

The MDE approach to software development is centered on the notion of model and thus heavily relies on its main operator: *model transformation*. Model transformations can be rather complex software and therefore they need to be developed following a systematic approach. Such an approach is yet to be proposed and among the challenges raised by this issue the need for a method for model transformations specification, in terms of activities and artifacts, stands out for us.

A number of specification approaches [1, 4, 10, 16, 19] with varying nature, intent and properties have been proposed. In addition to those, *transformation models* [2] appeared as a proposal for an artifact specifying model transformations, which promises to deliver a number of good features, most notably, the ability to express what a model transformation does, abstracting away its technical realization details. An evaluation of such an approach can be conducted from two separate points of view. First, the approach can be evaluated by analyzing its rationale. Such an evaluation, though mainly focused on the advantages of the approach, can be found in [2]. Second, an evaluation can be conducted based on experiences in its application to concrete cases. We currently lack such an evaluation since the only reported application of transformation models is found in [5]. There, the context is the set of transformations between Entity-Relationship models (schema and states) and Relational models (schema and states). The syntactic transformation (i.e., between ER and RE schemas) is well-known, yet simple.

The purpose of this work is to report the application of transformation models to a more complex syntactic model transformation called CD2DCD. Such a transformation was inspired by an activity proposed for an object-oriented development process in [12]. It was introduced in [18], and it was specified using traditional object-oriented artifacts in [17]. The intent of CD2DCD is to produce the static part of a collaboration, in the form of a fully dressed UML class diagram, from the behavioral part of the collaboration, in the form of a set of UML communication diagrams, and another class diagram understood as a Domain Model.

We are aware that a single application of transformation models does not suffice for a thorough evaluation. However, we believe that this experiment is a step in that direction since it provides elements which enable a number of insights from both points of view mentioned above.

The rest of this report is structured as follows. In Sect. 2 we review background information on specification approaches and provide an overview of transformation models. Section 3 presents the transformation model we constructed for specifying CD2DCD. In Sect. 4 we describe the case study used for validating the specification. Finally, a discussion and insights drawn from our experience is found in Sect. 5.

2 Background

In this section we review some specific aspects of existing approaches to specifying model transformations which will be used in our discussion later on. An overview of transformation models is addressed next.

2.1 Aspects of Specification Approaches

Some specification approaches have been already informally applied in practice. A good place to start looking for them is existing and well-known catalogues of model transformations, such as Kevin Lano's [11] and ATL Transformations Zoo [1]. Transformations in the first catalogue are specified by a text expressed in natural language for their intention, a pair of source/target models expressed in the concrete syntax for an example of operation, and a set of conditions for their correct use expressed as combination of text and OCL-like constraints. In turn, transformations in the second catalogue are specified following a template associated to the catalogue. The template includes a name for the transformation (both a short and a full name), a short textual description, a graphical or textual specification of all source and target metamodels including their invariants expressed in text and optionally in OCL, pre- and postconditions again expressed in text and optionally in OCL, and finally any form of pseudocode. For this last section, concrete specifications usually include transformation rules expressed both in natural language and in ATL. Although it lacks the example section included in the first catalogue, transformations are specified similarly to regular operations: a signature including types for arguments and return value (i.e., metamodels), and pre- and postconditions.

Model transformations express a mapping among model elements. We identify two orthogonal dimensions for understanding mapping specifications. In the first dimension, we distinguish mappings at a meta-level from mappings at a model-level. The former are based on the abstract syntax of the involved languages, while the latter are based on their concrete syntax. Another dimension in which mappings can be understood refers to the means by which the mapping is expressed: in a "generic" form, or based on concrete examples. In the former, mappings are generically defined among language constructs, and in the latter mappings are defined on particular models. This latter form was recently introduced under the name of Model Transformation by Example (MTBE). To illustrate concrete enactments of these approaches, ATLAS Model Weaver (AMW) [4] follows a generic approach at the meta-level. In turn, a generic approach at the model-level is not apparent. MTBE approaches can be found at the meta-level [7, 16], and at the model-level [19]. As a special case, [10] suggests a design process for model transformations which combines two of these approaches; an initial version of a transformation based on MTBE at the model-level is refined to a generic version at the meta-level. A detailed discussion on these two dimensions goes well beyond the scope of this report, however they will be revisited in a later section for classifying the transformation models approach as the basis of a part of our discussion.

2.2 Overview of Transformation Models

Rationale. While model transformations focus on the process and means of going from a source model to a target model, transformation models characterize transformations focusing on properties of the source and target models. In terms of the UML Reference Manual [14],

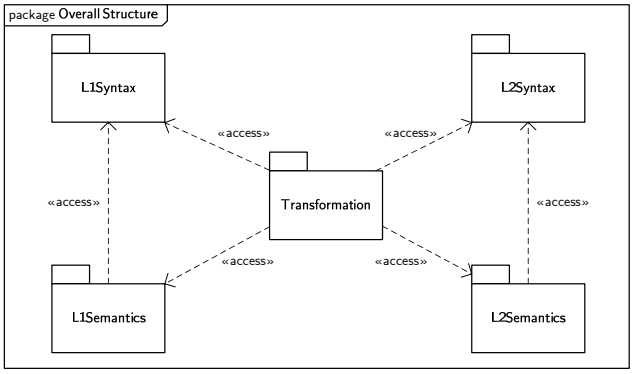

Fig. 1. Generic package structure of transformation models. Packages concerning the syntactic part include metaclasses defining the syntax of languages, packages concerning the semantic part include metaclasses defining an interpretation of those languages. The central package includes classes that define the transformation model, which are the context of the invariants that define the mapping(s) among all metaclasses.

a transformation model can be understood as a *specification*, a model transformation as an *implementation*, and their relationship as a *realization*[1]. A specification needs to be more abstract than an implementation in some concrete sense. In this case, a transformation model abstracts away the details of the technical realization of its related model transformations. This means that it expresses a mapping without relying on the constructs of particular transformation systems. Furthermore, as there may be many different ways to express the same idea within the same technology, and even across different ones, a transformation model concentrates the similarities of these many alternatives.

The specific way a transformation model defines a mapping is by expressing properties of the involved models. More concretely, the mapping is defined by a set of predicates on models, where only source models with their corresponding target models must satisfy such predicates. For example, provided that predicates $P_1, \ldots, P_k$ specify the mapping for a unary transformation T, then for every predicate: $P_i(m, n)$ iff $T(m) = n$. In other words, P_i are invariants of the transformation model that relate valid pairs of source and target models.

Overall Organization. Transformation models conform to MOF. Therefore, first, a transformation model can actually be regarded as a metamodel, and second, a transformation model is basically expressed in terms of classes, binary associations and OCL constraints. Figure 1 shows the overall structural organization of a transformation model which involves languages L1 and L2. Packages L1Syntax and L2Syntax contain the MOF-compliant metamodels for L1 and L2 respectively, in turn package Transformation contains the transformation model itself. Such a package contains:

[1] In the remainder of this document we will stick to these terms, however later on we will discuss some issues concerning their application in the context of model transformation development.

- *Static structure*: a set of classes, optionally associated among them, which are associated to metaclasses contained in L1Syntax and L2Syntax.
- *Constraints on the static structure*: a set of OCL invariants defined in the context of the previous classes that specify the mapping.

More specifically, a class contained in Transformation is typically associated to metaclasses contained in L1Syntax and L2Syntax, thus bridging both metamodels and motivating the dependencies between packages shown in Fig. 1. In this way, an invariant defined in the context of such a class may access both metamodels for expressing properties of source and target models.

Finally, providing extra metamodels for an interpretation of the languages involved in the transformation, their semantics can be included into the big picture as well. This enables the mapping to include not only syntactic aspects but also semantic ones. In Fig. 1 these additional metamodels are L1Semantics and L2Semantics. They naturally depend on the syntax of their corresponding languages, and their relationship with Transformation has the same nature of the dependencies just discussed above.

Features. Transformation models are based on invariants, therefore they are not directly executable. Many approaches represent transformations as operations; on the contrary, transformation models are represented as classes making the specification direction neutral. The general structure described before provides unrestricted access, for expressing the invariants, from the classes representing the transformation to every single metaclass used for defining the languages. Furthermore, as transformation models and metamodels are all MOF-compliant, invariants can be expressed in a uniform fashion. Finally, transformation models are models, and hence, they can be the subject of any model manipulation, most notably, model transformations.

3 A Transformation Model for CD2DCD

In this section we describe our application of transformation models to the specification of the CD2DCD model transformation. We start by giving an overview of the intent of the transformation, proceeding next to the formulation of the transformation model. Such a formulation includes here the static structure of the model only, and for reasons of brevity, the other main ingredient (i.e., the invariants) is fully presented in App. A.

3.1 Intent of CD2DCD

In the Rational Unified Process (RUP) [9], system behavior is captured in the form of use cases. Use cases express the way actors and the system interact in order to fulfill their goals. Larman [12] proposes to further express use cases as interactions where the system receives messages from actors, which trigger some special operations. These operations are called system operations. The design of system operations involves the definition of mechanisms inside the system which realize the expected behavior of each system operation. Such mechanisms are usually expressed as UML interaction diagrams, particularly communication diagrams. Communication diagrams define what objects participate in the mechanisms and what messages they send each other in order to make the mechanisms work. Participants are usually inspired by the concepts that are

present in the problem domain and their relations. An abstraction of the problem domain is captured in a UML class diagram called Domain Model.

For enabling the mechanisms depicted in the communication diagrams, a complete description of the structure of the participants is required. This description includes the definition of classes with their properties and relations, enabling a configuration of objects which can behave as expressed in the interactions. Such description takes the form of a UML class diagram, and is called a Design Class Diagram. The purpose of the transformation model presented next is the generation of a Design Class Diagram from a number of communication diagrams and a Domain Model.

Larman proposes a high level procedure, intended to be manually applied, for generating a Design Class Diagram. This motivated the idea of developing a model transformation which realizes such a procedure, enabling also its automatic execution. It is presented as a sequence of steps involving the population of an initially empty class diagram with design elements generated from information contained both in the interactions and in the Domain Model. The high level procedure (in its original form) is as follows:

1. *Identify software classes and illustrate them.* This involves scanning all communication diagrams and listing the classes mentioned, and then drawing a class diagram with these classes including the attributes previously identified in the Domain Model that are also used in the design.
2. *Add method names.* The methods of each class can be identified by analyzing the communication diagrams. In general, the set of all messages sent to instances of a class X across all interactions indicates most methods that class X must define. In turn, "create messages" and "accessing methods" are omitted. Additionally, messages to collections are assumed to be messages to container objects and are omitted as well.
3. *Add more type information.* The types of the attributes, method parameters and method return value may be shown.
4. *Add associations and navigability.* Associations are chosen based on a need-to-know criterion (i.e. the associations required to satisfy the visibility and ongoing memory needs indicated by the communication diagrams). Common situations suggesting a need to define an association with a navigability adornment from A to B are: A sends a message to B, A creates an instance of B, and A needs to maintain a connection to B.
5. *Add dependency relationships.* Dependencies are used to depict non-attribute visibility between classes; that is, parameter, global or locally declared visibility.

The proposal above seems concrete and enables a systematic generation of Design Class Diagrams. However, it omits some details which are necessary to perform certain steps. For example, concrete criteria for selecting attributes from the Domain Model, for identifying method names, and also for defining multiplicity values, are missing in steps 1, 2 and 4 respectively. Furthermore, to some extent, the procedure can be misleading. It is legal for an instance of class A, when having locally declared or parameter visibility on an instance of class B, to send a message to it. According to step 4, this would generate a navigable association from class A to class B, when a dependency would have been more appropriate. Finally, some additional aspects are left implicit to the developer, such as the need for refactorings on the resulting diagram. A refinement of the proposed procedure, based both on practical experience and compromise, is then required for developing an automatic model transformation.

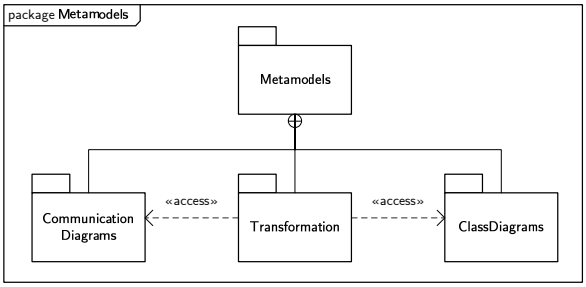

Fig. 2. Metamodel organization for the transformation model following the structure presented in Fig. 1. Packages CommunicationDiagrams and ClassDiagrams include the metamodels defining the syntax of source and target models. Package Transformation includes the transformation model.

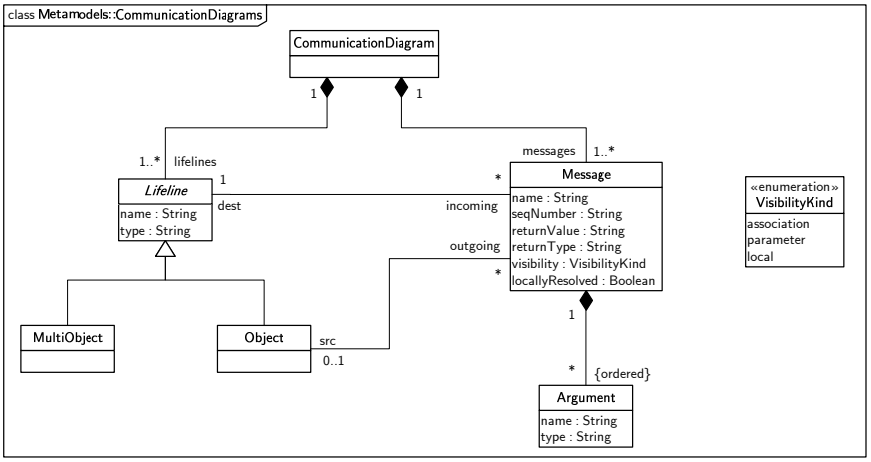

Fig. 3. Metamodel for communication diagrams

3.2 Structure of the Transformation Model

CD2DCD is an inherently syntactical model transformation; for that reason we dropped the semantic part of the general structure shown in Fig. 1. The concrete structure resulting from the adaptation to our specific case is shown in Fig. 2. Packages CommunicationDiagrams and ClassDiagrams include the UML-based metamodels and well-formedness rules for the communication diagrams, and for the Domain Model and Design Class Diagram, respectively. In turn, package Transformation includes the class representing the transformation model as well as the constraints that define it. The contents of each package is detailed next because Transformation includes the transformation model itself, and the contents of CommunicationDiagrams and ClassDiagrams are required for a proper understanding of the invariants within Transformation.

Figure 3 shows the metamodel for communication diagrams. A communication diagram may contain objects and multiobjects. Objects may receive and send messages to other participants.

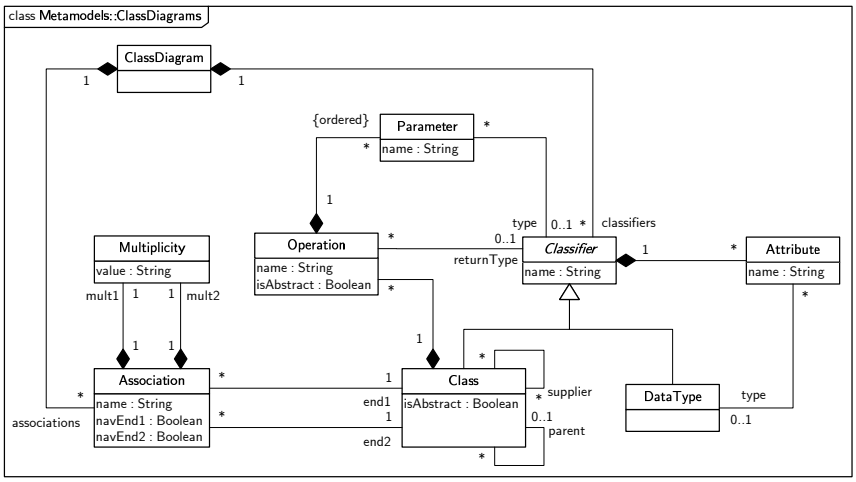

Fig. 4. Metamodel for class diagrams

Multiobjects are elements representing containers of objects and may receive messages only; the semantics of such messages, which are collection manipulation primitives, is assumed understood. Participants of interactions are typed. In turn, messages are sent from one participant to another, with the exception of the entry point of an interaction which has no source. A message has a sequence number and may return a typed value. A message may have typed arguments as well. The nature of the link between the source and the destination of a message may be specified in its visibility property. To that end, the VisibilityKind enumeration was defined. A message, via the locallyResolved meta-attribute, may be annotated with information specifying whether the destination object alone is capable of fulfilling its expected behavior. This is useful for deciding if a class provides a method for a given operation. A metaclass CommunicationDiagram owning a set of participants and a set of messages acts as a façade for providing a unique access point to actual elements.

Figure 4 shows the metamodel for class diagrams. A class diagram may contain classes and data types, both with typed attributes. Attribute types may be data types only. Classes may also contain operations. Both the return type and parameter types may be any type (class or data type). A class may be abstract and may have at most one superclass. Associations are binary and may relate classes only. A class may depend on an arbitrary number of suppliers. As a special case, a Domain Model is an analysis artifact, therefore design elements such as operations and dependencies are not used. Additionally, associations are bidirectional. Finally, a metaclass ClassDiagram owning a set of classifiers and a set of associations acts as a façade as before.

Figure 5 shows the static structure of our transformation model. In our case we defined a single class CD2DCD. An instance of that class may be associated to an arbitrarily number of communication diagrams, to exactly one class diagram playing the role of a Domain Model, and to exactly one class diagram playing the role of a Design Class Diagram. All constraints that

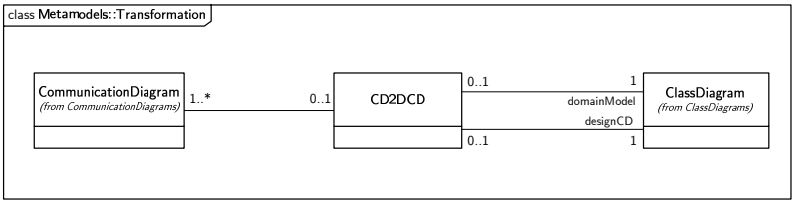

Fig. 5. Static structure of the transformation model

actually describe the transformation are OCL invariants in the context of class CD2DCD, and are fully detailed in App. A.

4 Case Study

The transformation model presented above was validated using concrete models obtained from a well known case study. The problem refers to a system for a Point-of-Sale register of a retail store. The complete specification of this case study can be found in [12]. The addressed use case is *Process Sale*, which deals with customers buying products at the store. The set of models includes:

- a Domain Model named ProcessSaleDM,
- a communication diagram for each system operation in the main success scenario of Process Sale, which will be enumerated next, and
- the resulting Design Class Diagram named ProcessSaleDCD, which we assume as correct as it was generated by our implementation of CD2DCD [17].

The interactions expressed in the communication diagrams constitute the behavioral part of the realization of Process Sale, while ProcessSaleDCD constitutes the static part. All these models are shown using concrete syntax in App. B.

For the main success scenario for the Process Sale use case the following system operations were identified:

- makeNewSale() creates a new sale and sets it as the current sale.
- enterItem(id : Integer, qty : Integer) adds qty pieces of the product identified by id to the current sale.
- endSale() : Float ends the current sale and returns its total amount.
- makeCashPayment(am : Float, cash : Float) records the cash payment for the current sale and registers the amount of cash paid by the customer.
- makeCheckPayment(am : Float, idN : String) records the check payment for the current sale and registers the number of the ID shown by the customer.

When the use case starts the system receives a makeNewSale message and gets ready for accepting and recording items. An arbitrary number of items is entered via multiple receptions of enterItem message. After the last item is entered the system receives an endSale message and stops accepting items for the current sale. The system may handle either cash or check payments.

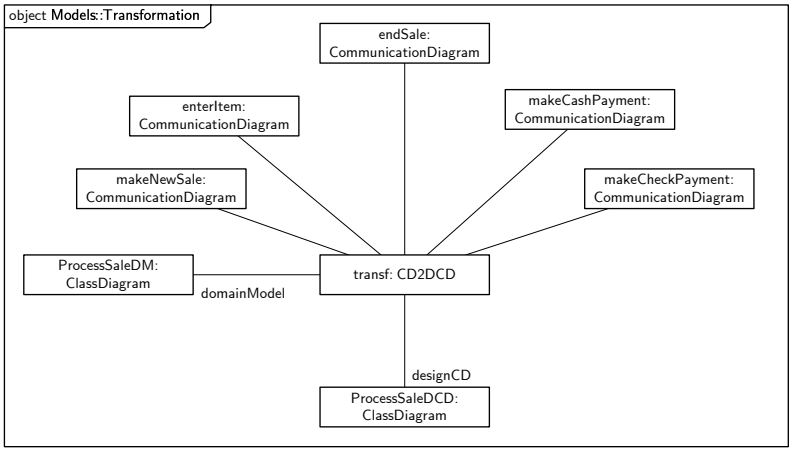

Fig. 6. Configuration of objects used for validating the transformation model (objects owned by instances of CommunicationDiagram and ClassDiagram are elided from the diagram for reasons of clarity).

The payment method is chosen by the customer. The system then receives a makeCashPayment or makeCheckPayment message respectively, and the use case is done.

Our validation was successful and consisted in establishing the configuration shown in Fig. 6, and evaluating on it all the constraints associated to class CD2DCD. For that purpose we used the USE [6] tool.

5 Discussion

We conclude this report with a discussion on the transformation model approach by means of a number of insights based on our experience on applying the approach to a large problem. While the syntactic part of the transformation model in [5] was specified using 3 invariants in 45 lines of OCL code with 3 auxiliary operations, our transformation model for CD2DCD required 21 invariants in 319 lines of OCL code with 8 auxiliary operations.

Insights reflect our current position on the transformation models proposal and they identify some key challenges as well. Insights refer to the classification of approaches discussed in Sect. 2.1, to the benefits presented in Sect. 2.2, and to the specification process and to validation and tool support as experienced by us. We conclude the discussion identifying some issues concerning the use of standard terminology to the context of model transformation development.

5.1 On the Transformation Models Approach

According to the terms discussed in Sect. 2.1 for classifying approaches to model transformation specification, transformation models specify model transformation in a "generic" fashion and at the meta-level. A generic specification enables a complete definition of the mapping, in contrast to MTBE-based approaches; however, it is rather hard to handle all possible cases at once. In turn, a mapping description at a meta-level enables independency from the concrete syntax

of any modeling language. However it is much closer to a final transformation implementation that deals with internal model representations [10], which tends to make the specification more complex. This is not a minor issue, because one would expect a specification to be simpler, or at least less detailed, than an implementation. To some extent, this is the price to pay with formal specifications, as is the case for OCL-based ones. An approach, as that in [10], which progressively increases the complexity could be a good trade-off.

5.2 On the Features of Transformation Models

Syntactic and Semantic Levels. As in CD2DCD, the semantic level is not necessarily mandatory for every model transformation. The way in which the general structure of transformation models is defined (see Fig. 1) enables the specifier to seamlessly ignore that part of the model.

Executability. Transformation models are non-executable. This is not necessarily a flaw. Model executability currently seems to be a major concern. When an artifact is executable, most notably implementation ones, there is a need to produce a model of it, which eventually "needs" to be turned into something executable, establishing a recursion. We argue that executability is a nice property but it should not be mandatory in every case. In this case, realization techniques are required though.

Realization Techniques. For transformation models, the definition of realization techniques is a good direction to take. Although we observed a resemblance between certain portions of the specification and the code that implements the transformation (in our case in Kermeta), producing an implementation from an OCL-based specification is a hard task [15]. On the one hand, although some patterns among the constraints of the transformation model may be detected, OCL seems to be too general and allows for many alternatives for expressing the same idea, and thus for establishing a general realization procedure. On the other hand, constraints may include too little information for producing an executable version (e.g. it is not always obvious how to enforce some equalities). A DSL with fewer constructs, which enables some form of annotation (somewhat similar to AMW) could be applicable. Though, the price to pay is to drift away from a standard.

Direction Freeness. Transformation models do not impose any specific directionality to the transformation. This is a very nice feature. However, some transformations are inherently unidirectional, as is the case of CD2DCD. Even in such a context, one can argue that bidirectionality can be useful at least for synchronization purposes. The discussion on the appropriateness of modifying a target model instead of applying the desired change in the adequate source model and propagating it through the transformation is in most cases rather philosophical and goes beyond the scope of this work. What is clear is that in unidirectional transformations, changes applied to the target model need to be at least restricted. Using CD2DCD as an example: a change in the name of a class in the Design Class Diagram could be resolved by updating the type of the corresponding objects of every interaction. But what would be the effect on the communication diagrams (and more precisely, exactly on which one of them) of the addition of a new operation on a given class within the Design Class Diagram? This question assumes an

established mechanism for distinguishing a simple update in the name of a class, from a more complicated case where a class is replaced with a similar one. CD2DCD can be regarded, in the backward direction, as "semidirectional".

Uniformity. Classes within the transformation model (e.g. CD2DCD class) have access to all other metaclasses (e.g. classes within CommunicationDiagrams and ClassDiagrams). This ensures that invariants can refer to any element of any model as required. Since this access is enabled by the façade metaclasses (e.g. CommunicationDiagram and ClassDiagram), invariants must obligatorily navigate through them to access elements, making OCL expressions even more verbose. Shorthand notation can be used to avoid this, but in that case expressions would not be manageable by standard tools anymore. As an alternative, more associations to appropriate metaclasses, instead of the façades, may be defined, however making the transformation model more complicated.

Transformation of Transformations. Transformation models can be the subject of model transformations. This is another nice property indeed. However, it is not clear the actual benefit of this possibility, since from a structural point of view transformation models are likely to be rather small and simple models (see Fig. 5). Furthermore, the definition of model transformations that operate on a MOF model which has associated a (potentially large) set of (potentially extremely complicated) OCL invariants appears as challenging.

Directionality of the Mapping. Finally, constraints within transformation models are unidirectional. Therefore the mapping can be an injection between elements or even a bijection, at the extra cost of including more constraints. Even though in some cases an injection is preferable, Kent *et al.* [8] argue that mappings should be a bijection, and moreover, they should be expressible in one single definition, thus avoiding the aforementioned extra cost and consistency concerns, but also ruling out the chance to defining injections.

5.3 On the Specification Process

The main challenge when it comes to elaborate a transformation model is the lack (to the best of our knowledge) of techniques for building a specification based on invariants. An ad hoc elaboration of such a specification is arduous. In fact, the time and effort we devoted to the transformation model reported in this work is comparable to the time and effort we devoted to the ad hoc implementation of CD2DCD [18].

For the formulation of the specification we inspired ourselves by the procedure described in Sect. 3.1, designing the complete set of invariants before detailing them. For defining the mapping in the forward direction we basically proceeded *source-oriented*, and *target-oriented* [3] for the backward direction. In particular, the original set of invariants was redefined twice as we dove into their details; we found that handling generalizations in the source metamodels was particularly tricky. Additionally, we believe there is much room for underspecification, especially when the developer does not have a deep understanding of the fine details of the transformation. Finally, a trade-off between the size and quantity of invariants is required. Too many small invariants make the specification globally complex to understand; too few large invariants make them individually hard to understand and debug.

The Transformation package shown in Fig. 5 is quite simple as it includes only one class: CD2DCD. An alternative structure which includes more classes could have been used though. However, how an interesting and more complex structure would look like, as well as its benefits, was not obvious for us.

5.4 On Validation and Tool Support

The elaboration of the transformation model was supported by the USE tool, both for type-checking and for evaluating constraints. First, as constraints are verbose and complex, the specification process is error-prone and USE's built-in type-checker helped in quickly identifying type errors. Though, a specialized OCL editor equipped with an auto-complete feature would have speeded up the writing process. Second, we fed the tool with the state depicted in Fig. 5 which included models that we knew beforehand were correct, meaning that all invariants would be satisfied. The validation process then was as follows. We wrote invariants in a one-at-a-time fashion, and once an invariant was ready we added it to the model and then loaded the state to check the result (this means that the state needed to be loaded every time an invariant was added). If the invariant was not satisfied we explored the evaluation tree to find the error (this was not a trivial task) and reworked the invariant appropriately. If the invariant was satisfied we modified the state in a number of specific ways for making the invariant to fail. This way we made sure to avoid false positives, just as we were testing a regular operation. After the last invariant was added and checked this way, we concluded the specification and the transformation model was considered validated. The case study we chose involves a representative number of constructs and therefore we consider the specification, although not thoroughly tested, reasonably validated.

It is worth noting that the USE tool, which handles UML class diagrams and OCL constraints only, may be applied to any transformation problem since transformation models are at the meta-level. Working at the model-level instead would require a tool supporting all the languages involved in the transformation.

USE does not support packaging facilities for elements, therefore all elements in the state were included in the same namespace. This implied that some cumbersome mechanisms for avoiding name clashes, such as using odd prefixes to the name of elements, were required. For example, both message 1 in the interaction of Fig. 8 and message 2.1 in the interaction of Fig. 9 are named create; our original strategy was to name the two objects of class Message representing them could not share a mnemonic name (i.e., create) even though they conceptually belong to separate models.

The state which represents the models of our case study was generated by a manually created script. With a total of 1059 commands, its creation was the source of a very large amount of errors. Although USE includes a limited state visualization facility, the complete state comprises 244 objects making the generated object diagram unmanageable, and the task of identifying errors very hard.

5.5 On Terminology

Throughout this document we dealt with the specification of model transformations, and we adhered to the terms of [14] which are standard in the modeling community. However, when analyzing some definitions we identified a number of issues concerning their application to the context of model transformation development which are, at least, not clear for us. First, a

specification is defined as "a declarative description of what something is or does." Second, an *implementation* is "a definition of how something is constructed or computed." Third, a *realization* is "the relationship between a specification and its implementation." Additionally, in the previous definition the term implementation is used as a noun, and a name collision arises when considering it also as a verb, since *to implement* is to "describe the functioning of a system in an executable medium." Specifications seem to be declarative, while implementation seem to have an executable, or at least, since its definition refers to the notion of computation, an operational flavor. The situation becomes intricate when considering the following statement:

> *There may be a chain of several specification-implementation relationships, in which each implementation defines the specifications for the next layer.* *[14, p. 22]*

We argue that the meaning of the referred layers conform different perspectives on artifacts, each of them having a particular purpose and which should be defined beforehand. As a typical example, a design model can be regarded as a specification of an implementation model, but at the same time as an implementation of an analysis model. Furthermore, what is actually being abstracted in those two realization relationships is different, and is part of the essence of the involved perspectives. Then, it could be also argued that an artifact is not a specification or an implementation *per se*; it is a specification or an implementation depending on the perspective from which it is considered. Therefore, the same artifact can be regarded as declarative and operational at the same time, but if so, this is because it is being considered from two very different perspectives.

When it comes to model transformation development perspectives are not clearly defined. A *transformation definition* expressed in languages such as QVT or ATL can be understood as a specification (it is declarative) and as an implementation (it is executable); no wonder they are usually referred to as "definition." Since we are dealing with specifications of model transformations, we believe that a deeper understanding is required about what we expect from such artifacts and what perspectives could be useful. For that reason, we believe that all these concepts, in the context of model transformation development, should be carefully reexamined.

6 Acknowledgements

This work was partially funded by Comisión Nacional de Investigación Científica y Tecnológica (CONICYT) Chile. We would like to thank María Cecilia Bastarrica and Daniel Perovich for reviewing early drafts of this document and for fruitful discussions.

References

1. ATL Transformations Zoo. Internet: `http://www.eclipse.org/m2m/atl/atlTransformations/`, 2007.
2. Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In Nierstrasz et al. [13], pages 440–453.
3. Krzysztof Czarnecki and Simon Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
4. Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, Erwan Breton, and Guillaume Gueltas. AMW: A Generic Model Weaver. In *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*, 2005.

5. Martin Gogolla. Tales of ER and RE Syntax and Semantics. In James R. Cordy, Ralf Lämmel, and Andreas Winter, editors, *Transformation Techniques in Software Engineering*, volume 05161 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.
6. Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.
7. Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG Document ptc/05-11-01, November 2005.
8. Stuart Kent and Robert Smith. The bidirectional mapping problem. *Electronic Notes in Theoretical Computer Science*, 82(7), 2003.
9. Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, third edition, December 2003.
10. Jochen M. Küster, Ksenia Ryndina, and Rainer Hauser. A Systematic Approach to Designing Model Transformations. Research Report RZ 3621, IBM, Zurich, July 2005.
11. Kevin Lano. A Catalogue of Model Transformations. Internet: `http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf`, 2007.
12. Craig Larman. *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, third edition, October 2004.
13. Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors. *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*, volume 4199 of *Lecture Notes in Computer Science*. Springer, 2006.
14. James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, 2005.
15. Dániel Varró. *Automated Model Transformations for the Analysis of IT Systems*. PhD thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, May 2004.
16. Dániel Varró. Model Transformation by Example. In Nierstrasz et al. [13], pages 410–424.
17. Andrés Vignaga. Generation of a Design Class Diagram in Kermeta, Internet: `http://www.kermeta.org/examples/mt_acm/`, 2006.
18. Andrés Vignaga and M. Cecilia Bastarrica. Transforming System Operations' Interactions into a Design Class Diagram. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *SAC*, pages 993–997. ACM, 2007.
19. Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. Towards Model Transformation Generation By-Example. In *HICSS*, page 285. IEEE Computer Society, 2007.

A Constraints of the Transformation Model

In this section we present the complete set of constraints that define the transformation model for CD2DCD, as well as every auxiliary operations used in the specification. All constraints are expressed in the context of CD2DCD class, and auxiliary operations are typically defined in the context of classes CommunicationDiagram or ClassDiagram. For the specification to be complete these classes require their own set of invariants. Such constraints would be well-formedness rules, and as we assume the semantics of both communication diagrams and class diagrams well understood we omit them here for reasons of brevity.

A.1 Invariants

In what follows we detail all OCL invariants which are the core of the transformation model. According to the terminology of [5], all constraints, with the exception of constraints 16 and 17,

are “exists constraints” which means that they require that for an element of a certain domain an element of another domain exists. For each constraint we provide an informal description in natural language followed by the OCL code. As invariants are unidirectional, we also annotate constraints with symbols (--→) or (←--) for denoting their direction: from source models to target model or from target model to source models respectively.

Constraint 1 (--→): For every possible type of lifeline in any interaction, a class in the target model must exist.

```
context CD2DCD
inv forLifelineExistsClass:
   self.communicationDiagram.lifelines.type->forAll(t |
      self.designCD.existsClass(t)
   )
```

Constraint 2 (--→): For every possible type of argument in any message of any interaction, a classifier in the target model must exist.

```
context CD2DCD
inv forTypedArgumentExistsClassifier:
   self.communicationDiagram.messages.argument->forAll(a |
      if self.communicationDiagram.lifelines.type->includes(a.type) then
         self.designCD.existsClass(a.type)
      else
         self.designCD.existsDataType(a.type)
      endif
   )
```

Constraint 3 (--→): For every possible type of return in any message of any interaction, a classifier in the target model must exist.

```
context CD2DCD
inv forTypedReturnExistsClassifier:
   self.communicationDiagram.messages->forAll(m |
      not m.returnType.isUndefined() implies
         if self.communicationDiagram.lifelines.type->includes(m.returnType) then
            self.designCD.existsClass(m.returnType)
         else
            self.designCD.existsDataType(m.returnType)
         endif
   )
```

Constraint 4 (--→): For every possible type of attribute in classifiers of the domain model which have a matching classifier in the target model, a data type in the target model must exist.

```
context CD2DCD
inv forTypedAttributeExistsDataType:
   self.domainModel.classifiers->forAll(cl |
      (self.designCD.existsClass(cl.name) or self.designCD.existsDataType(cl.name)) implies
         cl.attribs->forAll(a |
            self.designCD.existsDataType(a.type.name)
         )
   )
```

Constraint 5 (←--): For every possible data type in the target model, such a data type must be used for typing either an attribute in the domain model, an argument in any interaction, and the return value of any message.

```
context CD2DCD
inv forDataTypeExistTypedElement:
   self.designCD.classifiers->forAll(cl |
      cl.oclIsTypeOf(DataType) implies (
         self.domainModel.classifiers->exists(c |
            (self.designCD.existsClass(c.name) or self.designCD.existsDataType(c.name)) and
            c.attribs->exists(a | a.type.name = cl.name)
         ) or
         self.communicationDiagram->exists(cd |
            cd.messages->exists(m | m.returnType = cl.name)
         ) or
         self.communicationDiagram->exists(cd |
            cd.messages->exists(m | m.argument->exists(a | a.type = cl.name))
         )
      )
   )
```

Constraint 6 (←--): For every possible class in the target model, such a class must be used for typing either a lifeline, an argument, and the return value of a message in any interaction.

```
context CD2DCD
inv forClassExistTypedElement:
   self.designCD.classifiers->forAll(cl |
      cl.oclIsTypeOf(Class) implies (
         self.communicationDiagram->exists(cd |
            cd.lifelines->exists(l | self.designCD.isAncestor(cl.name,l.type))
         ) or
         self.communicationDiagram->exists(cd |
            cd.messages->exists(m | m.returnType = cl.name)
         ) or
         self.communicationDiagram->exists(cd |
            cd.messages->exists(m | m.argument->exists(a | a.type = cl.name))
```

```
         )
      )
   )
```

Constraint 7 (--→): For every attribute in the domain model, if the owner classifier has a match in the target model, the matched classifier must own a matching attribute.

```
context CD2DCD
inv forAttributeExistsAttribute:
   self.domainModel.classifiers->forAll(cl |
      self.designCD.existsClass(cl.name) or self.designCD.existsDataType(cl.name) implies
         cl.attribs->forAll(at |
            self.designCD.classifiers->exists(c |
               c.name = cl.name and
               c.attribs->one(a | a.name = at.name and a.type.name = at.type.name)
            )
         )
   )
```

Constraint 8 (--→): For every "getter" message in any interaction, an attribute must exist in the corresponding class.

```
context CD2DCD
inv forGetterExistsAttribute:
   self.communicationDiagram.messages->select(m |
      m.name.substring(1,3) = 'get' and m.locallyResolved
   )->forAll(m |
      self.designCD.classifiers->one(c |
         c.oclIsTypeOf(Class) and
         c.attribs->one(a |
            a.name = m.name.substring(4,m.name.size()).toLower() and
            a.type.name = m.returnType
         )
      )
   )
```

Constraint 9 (--→): For every "setter" message in any interaction, an attribute must exist in the corresponding class.

```
context CD2DCD
inv forSetterExistsAttribute:
   self.communicationDiagram->select(m |
      m.name.substring(1,3) = 'set' and m.locallyResolved
   )->forAll(m |
      self.designCD.classifiers->one(c |
```

```
         c.oclIsTypeOf(Class) and
         c.attribs->one(a |
            a.name = m.name.substring(4,m.name.size()).toLower() and
            a.type.name = m.argument->first().type
         )
      )
   )
```

Constraint 10 (←--): For every attribute in a class in the target model, there must either exist a matching attribute in a corresponding class, a "getter" or a "setter".

```
context CD2DCD
inv forClassAttributeExistsAttributeOrAccessor:
   self.designCD.classifiers->forAll(cl |
      cl.oclIsTypeOf(Class) implies
         cl.attribs->forAll(a |
            self.domainModel.classifiers->exists(c |
               c.oclIsTypeOf(Class) and
               c.name = cl.name and
               c.attribs->one(x | x.name = a.name and x.type.name = a.type.name)
            ) or
            self.communicationDiagram->exists(cd |
               cd.messages->exists(m |
                  m.name.substring(4,m.name.size()).toLower() = a.name and
                  if m.name.substring(1,3) = 'get' then
                     m.returnType = a.name
                  else
                     m.argument->first().type = a.name
                  endif
               )
            )
         )
   )
```

Constraint 11 (←--): For every attribute in a data type in the target model, there must exist a matching attribute in a corresponding data type.

```
context CD2DCD
inv forDataTypeAttributeExistsAttribute:
   self.designCD.classifiers->forAll(cl |
      cl.oclIsTypeOf(DataType) implies
         cl.attribs->forAll(a |
            self.domainModel.classifiers->exists(c |
               c.oclIsTypeOf(DataType) and
               c.name = cl.name and
```

```
            c.attribs->one(x | x.name = a.name and x.type.name = a.type.name)
         )
      )
   )
```

Constraint 12 (--→): For every message (which is not a "create" or an accessor or sent to a multiobject), there must be a corresponding operation in the class of the target object.

```
context CD2DCD
inv forMessageExistsOperation:
   self.communicationDiagram.messages->select(m |
      m.name <> 'create' and
      m.name.substring(1,3) <> 'set' and
      m.name.substring(1,3) <> 'get' and
      not m.dest.oclIsTypeOf(MultiObject)
   )->forAll(m |
      self.designCD.classifiers->exists(c |
         c.oclIsTypeOf(Class) and
         c.name = m.dest.type and
         c.oclAsType(Class).opers->one(o |
            o.name = m.name and
            o.returnType.name = m.returnType and
            Sequence1..(o.parameters->size())->forAll(i |
               o.parameters->at(i).name = m.argument->at(i).name and
               o.parameters->at(i).type.name = m.argument->at(i).type
            ) and
            if m.hasNested() or m.locallyResolved then
               o.isAbstract = false
            else
               o.isAbstract = true and
               c.oclAsType(Class).isAbstract = true
            endif
         )
      )
   )
```

Constraint 13 (←--): For every operation in the target model, the must be a corresponding message to an instance of the class which owns the operation.

```
context CD2DCD
inv forOperationExistsMessage:
   self.designCD.classifiers->select(cl | cl.oclIsTypeOf(Class))->forAll(cl |
      cl.oclAsType(Class).opers->forAll(o |
         self.communicationDiagram.messages->exists(m |
            m.name = o.name and
```

```
            m.returnType = o.returnType.name and
            Sequence1..o.parameter.size()->forAll(i |
               o.parameter.at(i).name = m.argument.at(i).name and
               o.parameter.at(i).type.name = m.argument.at(i).type
            ) and
            (not o.isAbstract implies (m.hasNested() or m.locallyResolved))
         )
      )
   )
```

Constraint 14 (--→): For every message with Association visibility, there must be an association (navigable in the direction of the message) between the classes of the sender and the receiver.

```
context CD2DCD
inv forMessageExistsAssociation:
   self.communicationDiagram.messages->select(m |
      m.visibilityKind = #Association
   )->forAll(m |
      self.designCD.associations->one(a |
         a.associates(m.src.type,m.dest.type) and
         if a.end1.name = m.src.type then
            a.end2Nav = true
         else
            a.end1Nav = true
         endif and
         if self.domainModel.getAssociation(m.src.type,m.dest.type).isUndefined() then
            a.name = m.src.type.concat('-').concat(m.dest.type)
         else
            a.name = self.domainModel.getAssociation(m.src.type,m.dest.type).name
         endif
      )
   )
```

Constraint 15 (←--): For every navigable association between two classes in the target model, there must exist at least one message in one interaction between instances of the participating classes and in the direction of the navigability.

```
context CD2DCD
inv forAssociationExistsMessage:
   self.designCD.associations->forAll(a |
      a.end1Nav implies self.communicationDiagram.messages->exists(m |
         m.src.type = a.end2.name and
         m.dest.type = a.end1.name and
         m.visibilityKind = #Association
      ) and
```

```
      a.end2Nav implies self.communicationDiagram.messages->exists(m |
         m.src.type = a.end1.name and
         m.dest.type = a.end2.name and
         m.visibilityKind = #Association
      )
   )
```

Constraint 16 (--→): The upper bound of a multiplicity is 1 unless a message exists from an instance of the class at its opposite end to an instance of the class at the multiplicity's end, which is also a mutliobject.

```
context CD2DCD
inv upperBoundMultiplicities:
   self.designCD.associations->forAll(a |
      a.end1Nav implies if self.communicationDiagram.messages->select(m |
         m.src.type = a.end2.name and
         m.dest.type = a.end1.name and
         m.dest.oclIsTypeOf(MultiObject))->isEmpty() then
            a.mult1.getUpper() = '1'
         else
            a.mult1.getUpper() = '*'
         endif and
      a.end2Nav implies if self.communicationDiagram.messages->select(m |
         m.src.type = a.end1.name and
         m.dest.type = a.end2.name and
         m.dest.oclIsTypeOf(MultiObject))->isEmpty() then
            a.mult2.getUpper() = '1'
         else
            a.mult2.getUpper() = '*'
         endif
   )
```

Constraint 17 (--→): In a navigable association from a source to a target class, if an instance of the target class was never created, received as an argument, or received as a result of a message, it suggests that a link between instances of the source and the target classes is mandatory, and therefore the lower bound of the multiplicity at the navigable end must be 1. Otherwise, a link is optional and that lower bound must be 0.

```
context CD2DCD
inv lowerBoundMultiplicities:
   self.designCD.associations->forAll(a |
      a.end1Nav implies
         if self.communicationDiagram.messages->exists(m |
            (m.src.type = a.end2.name and
```

```
            m.dest.type = a.end1.name and
            m.name = 'create') or
            (m.dest.type = a.end1.name and
            m.argument.type->includes(a.end2.name)) or
            (m.src.type = a.end1.name and
            m.returnType = a.end2.name)
         ) or self.communicationDiagram.messages->select(m |
            m.src.type = a.end2.name and
            m.dest.type = a.end1.name)->forAll(m |
               m.dest.oclIsTypeOf(MultiObject)) then
            a.mult1.getLower() = '0'
         else
            a.mult1.getLower() = '1'
         endif and
      a.end2Nav implies
         if self.communicationDiagram.messages->exists(m |
            (m.src.type = a.end1.name and
            m.dest.type = a.end2.name and
            m.name = 'create') or
            (m.dest.type = a.end2.name and
            m.argument.type->includes(a.end1.name)) or
            (m.src.type = a.end2.name and
            m.returnType = a.end1.name)
         ) or self.communicationDiagram.messages->select(m |
            m.src.type = a.end1.name and
            m.dest.type = a.end2.name)->forAll(m |
               m.dest.oclIsTypeOf(MultiObject)) then
            a.mult2.getLower() = '0'
         else
            a.mult2.getLower() = '1'
         endif
   )
```

Constraint 18 (--→): For every message with Parameter or Local visibility, there must be a dependency from the class of the sender to the class of the receiver.

```
context CD2DCD
inv forMessageExistsDependency:
   self.communicationDiagram.messages->select(m |
      m.visibilityKind = #Local or m.visibilityKind = #Parameter
   )->forAll(m |
      not self.designCD.associations->exists(a | a.associates(m.src.type,m.dest.type)) implies
         (self.designCD.getClass(m.dest.type).provider.name->includes(m.src.type) and
         self.designCD.getClass(m.src.type).supplier.name->includes(m.dest.type))
```

```
   )
```

Constraint 19 (--→): For every message argument which is an instance of a class, there must be dependency from the class of the object that receives the message and the class that types the argument.

```
context CD2DCD
inv forArgumentExistsDependency:
   self.communicationDiagram.messages->select(m |
      m.argument->exists(a | self.designCD.existsClass(a.type))
   )->forAll(m |
      m.argument->select(a | self.designCD.existsClass(a.type))->forAll(a |
         ((not self.designCD.associations->exists(as | as.associates(m.dest.type,a.type)))
         and
         m.dest.type <> a.type) implies
            (self.designCD.getClass(m.dest.type).supplier.name->includes(a.type) and
            self.designCD.getClass(a.type).provider.name->includes(m.dest.type))
      )
   )
```

Constraint 20 (--→): For every message that returns and object (not a data value), there must be a dependency from the class of the object that sends the message to the class of the returned object.

```
context CD2DCD
inv forReturnExistsDependency:
   self.communicationDiagram.messages->select(m |
      self.designCD.existsClass(m.returnType)
   )->forAll(m |
      not self.designCD.associations->exists(a | a.associates(m.returnType,m.src.type)) implies
         self.designCD.getClass(m.src.type).supplier.name->includes(m.returnType) and
         self.designCD.getClass(m.returnType).provider.name->includes(m.src.type)
   )
```

Constraint 21 (←--): For every dependency in the target model, there must be at least either: (a) one message in any interaction from an instance of the client class to an instance of the supplier class with Parameter or Local visibility, (b) one message received by an instance of the client class which has at least one argument which is an instance of the supplier class, or (c) one message sent by an instance of the client class which returns an instance of the supplier class.

```
context CD2DCD
inv forDependencyExistsMessArgRet:
   self.designCD.classifiers->select(c |
      c.oclIsTypeOf(Class) and not c.oclAsType(Class).supplier->isEmpty()
   )->forAll(c |
```

```
      c.oclAsType(Class).supplier->forAll(s |
         not self.designCD.associations->exists(a | a.associates(c.name,s.name)) and
         (self.communicationDiagram.messages->exists(m |
            m.src.type = c.name and
            m.dest.type = s.name and
            (m.visibilityKind = #Local or m.visibilityKind = #Parameter)
         ) or
         self.communicationDiagram.messages->exists(m |
            m.dest.type = c.name and m.argument.type->includes(s.name)
         ) or
         self.communicationDiagram.messages->exists(m |
            m.src.type = c.name and m.returnType = s.name
         )
         )
      )
   )
```

A.2 Auxiliary Operations

The operation `hasNested()` determines if a message has a nested subinteraction.

```
context Message::hasNested() : Boolean =
   if self.dest.oclIsTypeOf(Object) then
      self.dest.oclAsType(Object).outgoing->exists(m |
         m.seqNumber.substring(1,1) = self.seqNumber.substring(1,1)
      ) or
      self.src.isUndefined()
   else
      false
   endif
```

The operation `existsClass(...)` determines if a class diagram owns a class with a given name.

```
context ClassDiagram::existsClass(c : String) : Boolean =
   self.classifiers->one(cl |
      cl.oclIsTypeOf(Class) and cl.name = c
   )
```

The operation `existsDataType(...)` determines if a class diagram owns a data type with a given name.

```
context ClassDiagram::existsDataType(dt : String) : Boolean =
   self.classifiers->one(cl |
      cl.oclIsTypeOf(DataType) and cl.name = dt
```

```
   )
```

The operation `getAssociation(...)` finds an association in a class diagram which associates two given classes.

```
context ClassDiagram::getAssociation(c1 : String,c2 : String) : Association =
   self.associations->select(a |
      (a.end1.name = c1 and a.end2.name = c2) or (a.end1.name = c2 and a.end2.name = c1)
   )->asSequence()->first()
```

The operation `getClass(...)` finds a class with a given name in a class diagram.

```
context ClassDiagram::getClass(c : String) : Class =
   self.classifiers->select(cl |
      cl.oclIsTypeOf(Class) and cl.name = c
   )->asSequence()->first().oclAsType(Class)
```

The operation `associates(...)` determines if two given classes are associated in a class diagram.

```
context Association::associates(c1 : String,c2 : String) : Boolean =
   (self.end1.name = c1 and self.end2.name = c2) or
   (self.end1.name = c2 and self.end2.name = c1)
```

The operation getUpper() returns the upper bound of a multiplicity.

```
context Multiplicity::getUpper() : String =
   if self.value.substring(self.value.size(),self.value.size()) = '1' then
      '1'
   else
      '*'
   endif
```

The operation getLower() returns the lower bound of a multiplicity.

```
context Multiplicity::getLower() : String =
   if self.value.substring(1,1) = '1' then
      '1'
   else
      '0'
   endif
```

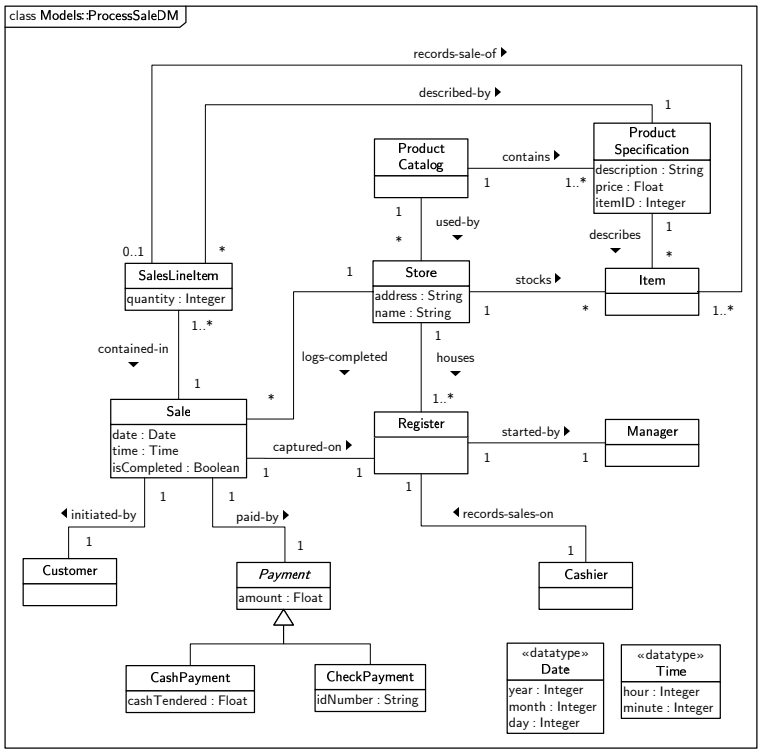

Fig. 7. Domain Model for Process Sale use case

B Sample Models for the Case Study

In this section we present the concrete models we used for validating the transformation model.

B.1 Source Models

The Domain Model illustrated in Fig. 7 shows the relevant concepts in the POS, restricted to the Process Sale use case. This version is almost identical to the original in [12], and includes the following extensions. The concept of a payment was specialized here for enabling both cash and check payments, and our model includes type information for every attribute. The former extension is motivated by the lack of generalization relationships in the original Domain Model; this version is more realistic. With the latter extension the transformation is feeded with a more detailed model.

The Store concept models the organization that owns the POS and registers all completed sales. The POS itself is modeled by Register. The current sale is related to a register by means of association captured-on. A sale has a SalesLineItem for each type of purchased product, for which the quantity is specified. Each of such entries has associated the information of the product

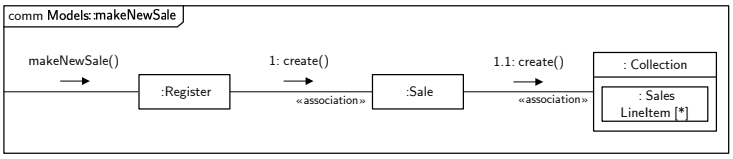

Fig. 8. Interaction for makeNewSale() system operation

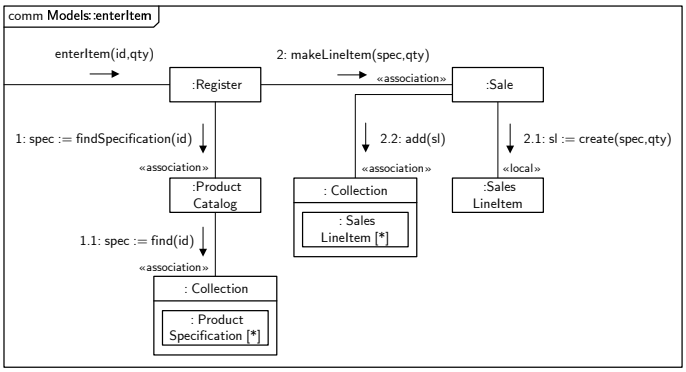

Fig. 9. Interaction for enterItem() system operation

(ProductSpecification), in particular, its price. A completed sale is associated with a payment, either by cash or by check.

When starting a new sale, the register creates a new instance of Sale, which in turn creates an empty collection of sales line items. This is shown in Fig. 8.

Every time an item is entered, the register looks for its specification and passes it to the sale in process, which creates a new line for it and records it in the line collection. Figure 9 shows this interaction.

After the last item is entered the register is asked to end the sale as shown in Fig. 10. The register firstr notifies the current sale that the purchase is complete and then asks its total for display. The sale iterates over its line items, collecting the subtotal of each, and returns the result.

The system supports two different paying methods, where different information is recorded. For finishing the use case, the register is told which paying method applies. The register passes the information to the sale, which creates the appropriate variant of payment. In either case, the register passes the sale to the store which is responsible for recording it. Figures 11 and 12 show the interactions for both variants of payment handling.

B.2 Target Model

When the transformation is executed on the source models presented above, the Design Class Diagram shown in Fig. 13 is generated.

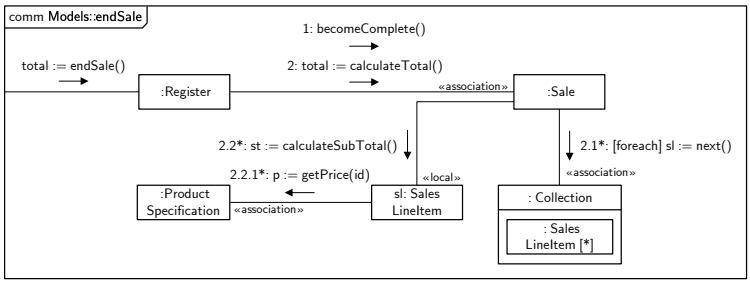

Fig. 10. Interaction for endSale() system operation

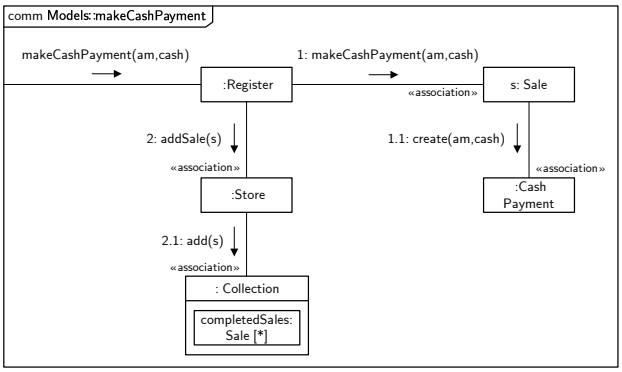

Fig. 11. Interaction for makeCashPayment() system operation

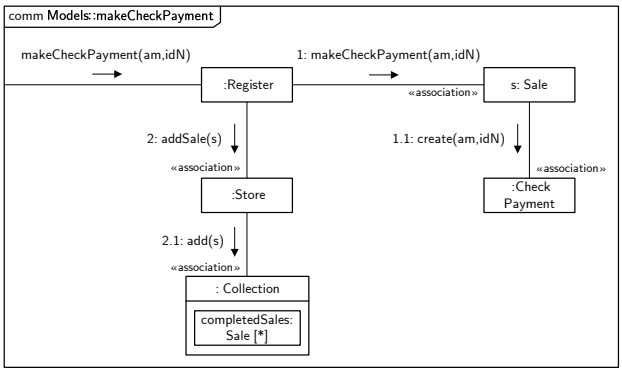

Fig. 12. Interaction for makeCheckPayment() system operation

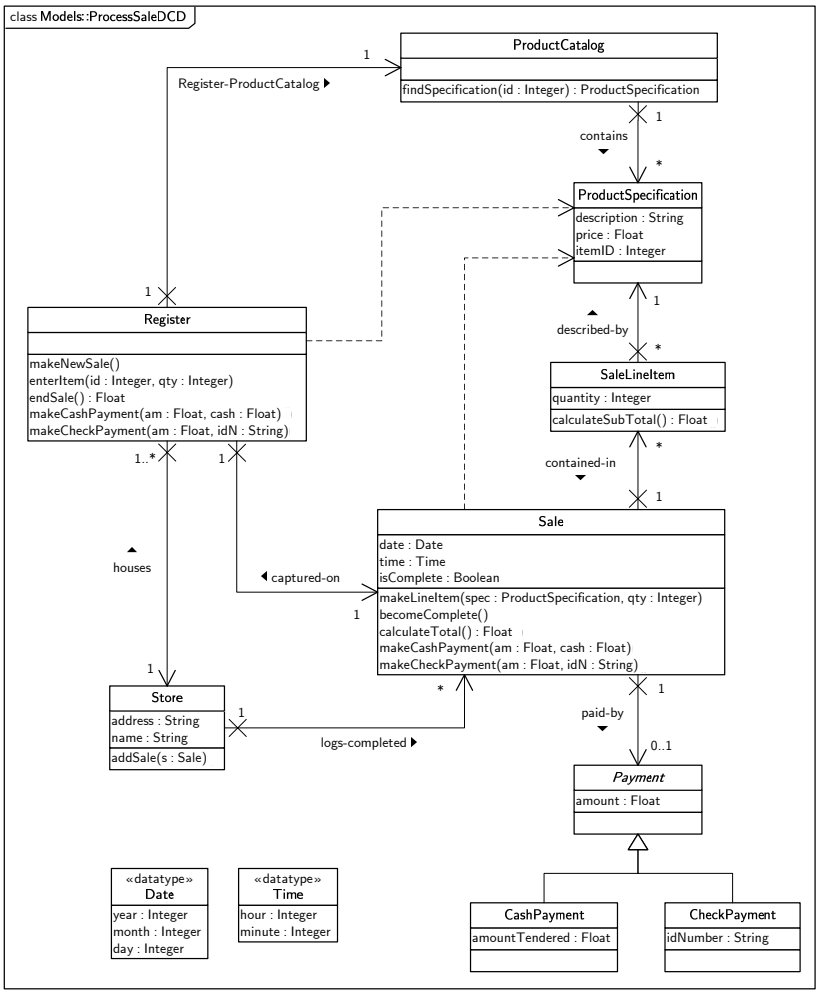

Fig. 13. Design Class Diagram for Process Sale use case