On Extracting a Design out of Software Contracts using Model Transformations

Andrés Vignaga, Daniel Perovich, and María Cecilia Bastarrica

Universidad de Chile, FCFM, Departamento de Ciencias de la Computación,
Av. Blanco Encalada 2120, 3^{er} Piso, 837-0459 Santiago, Chile
{avignaga,dperovic,cecilia}@dcc.uchile.cl

Abstract. A software contract specifies the behavior of an operation, focusing on its effect on the system state, and can be used as a basis to design a realization of such behavior. In object-oriented software development, software contracts are often used during analysis for specifying system level operations. Also in this context, object interactions are used to represent the behavioral design. Software contracts suggest a set of system responsibilities, which in turn are assigned to objects during the design. This step in the development process is currently performed manually, requiring some extent of creativity. Although general guidelines for assigning responsibilities exist, no systematic approach to extract a design from software contracts has been proposed. In this work we explore the elaboration of such an approach. To this end, we propose techniques based on model transformations and other Model Driven Engineering concepts for automatically extracting object interactions, expressed in terms of communication diagrams, out of software contracts. Additionally, a toolset supporting these techniques was developed and we show their applicability to a well known case study.

1 Introduction

A development process organizes software development into a series of activities, each of them accepting input artifacts and producing output artifacts. A concrete processes proposes methods and techniques, which prescribe, with varying levels of detail, how to perform many of those activities. Currently, there exist a number of tools which assist or even automate the realization of some activities. However, many of them are still carried out manually by developers.

Model Driven Engineering (MDE) promotes an approach where artifacts are considered as models and the activities which manipulate them as transformations. Such an approach requires the formalization of models and the systematization of activities. In object-oriented development, a number of activities are already in this category. For example, in implementation activities, partial source code generation from class diagrams is widely supported by CASE tools. Also, design class diagrams can be automatically generated from a set of object interactions and a domain model [19]. Tools typically automate activities which involve translation and where artifacts are well understood. Examples of such

activities are those comprised in design, implementation and even testing. Activities involving creativity, particularly at early stages of development, usually lack tool support. An example is the transition from analysis to design, most notably, the activity of extracting a design out of a system level behavioral specification.

In this work we explore the elaboration of such an approach in the context of the Rational Unified Process [9] and the refinement proposed by C. Larman [10] for the specification of system behavior. There, system behavior is specified by means of Software Contracts, and the key aspects of design are object interactions which realize such specifications, expressed as communication diagrams. In these terms, the problem addressed is the elaboration of an approach for extracting object interactions out of software contracts of system level operations.

This activity is comprised in the Analysis and Design discipline and is currently considered as mainly creative. In practice, the activity is carried out manually. Methodological approaches do not provide concrete guidance in how to design object interactions from software contracts, thus no tool assistance is available. This activity involves two main steps: to identify fine grained responsibilities from system level responsibilities expressed in the software contracts, and to assign such fine grained responsibilities to objects. The first step is realized by a thorough analysis of software contracts, and usually, based on the developers' understanding of the system responsibilities. Specifying complete software contracts is not trivial; underspecified contracts provide little information for responsibility identification. Since in many practical situations they currently are not cost effective, their specification is usually omitted. Expertise of object-oriented developers lead to techniques, such as CRC [2] and GRASP [10], which provides criteria for responsibility assignment and metrics for structure design. However, these techniques provide no support to fine grained responsibility identification and little assistance on the steps to be followed for using such criteria. The assignment of fine grained responsibilities is usually performed ad-hoc and not documented explicitly. Interaction design is related to programming. Deriving interaction design from a set of responsibilities is therefore related to program derivation. The general case for program derivation seems unfeasible, or at least hard to tackle. Existing approaches from the formal methods area [8] focus on algorithms like searching or sorting, but there are none tackling program derivation for large object-oriented systems. This work is not intended to cope with the general case, and neither to be applied to all large systems. It is restricted to object-oriented systems whose state is based on information models, and where system operations manipulate such state.

The approach we propose tackles the two main steps mentioned above separately. The first step is based on the existing resemblance between a software contract for a system level operation and a relational model transformation which exhibits the same behavior of such system level operation. If we understand software contracts as equivalent to relational model transformations, then an engine which is capable of executing such transformation on a given input system state could be extended to additionally deliver the set of actions on that input state which are necessary to produce the output system state. Such actions conform

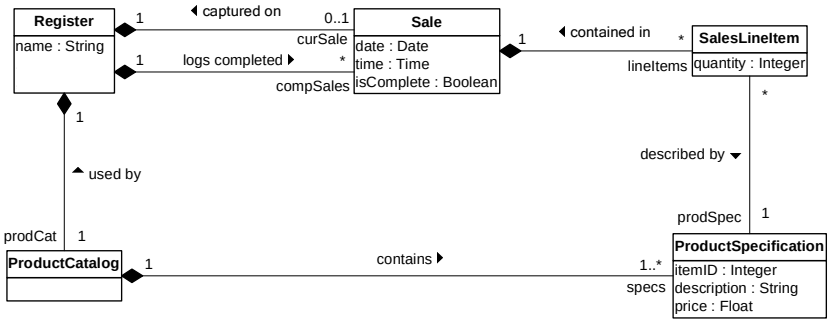


Fig. 1. Domain Model for the POS system.

the set of fine grained responsibilities. We use QVT [12] for specifying software contracts and build a QVT Engine for processing the corresponding relational transformation that produces these responsibilities as its output. The second step consists of a mechanical transformation which generates an object interaction, based on the information model, the set of responsibilities and a set of design decisions in terms of GRASP. For this purpose, we built a Kermeta weaver that takes the execution trace generated by the QVT Engine and the design decisions, and builds the communication diagrams that realize the original software contracts.

The rest of this paper is organized as follows. Section 2 introduces background information on software contracts, object-oriented design, relational model transformations, and the driving case study used to illustrate our approach. In Section 3 the key aspects of the approach are presented, and the tools developed for validating its feasibility are discussed. Section 4 details the complete application of our approach to a system operation within the case study, and demonstrates the operation of the toolset. Section 5 concludes.

2 Background

2.1 POS

The case study we work on along this paper is the NextGen point-of-sale (POS) system introduced by C. Larman in [10], where it is used to explain the application of the Rational Unified Process. It deals with different kinds of requirements such as functionality, fault-tolerance, client-server communication, flexibility and customization. In this work we exclusively focus on functional requirements.

A POS system is a computerized application used to record sales and handle payments in a retail store, which is deployed on each register. Figure 1 presents the Domain Model for the application. This model is variant of that presented in [10] intended to simplify the exposition of our proposal. In the POS domain

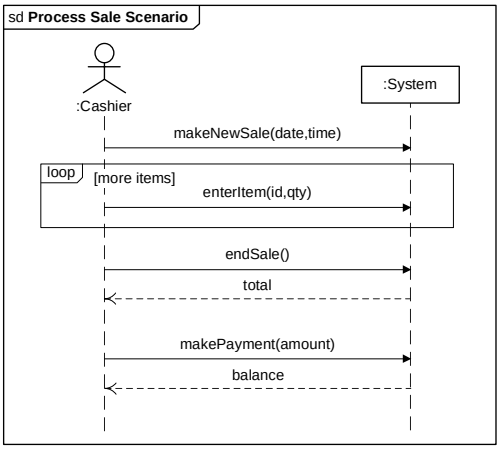


Fig. 2. System Sequence Diagram for the main success scenario of the Process Sale use case.

there is a single register which has a product catalog containing product specifications. In addition, the register logs all completed sales and may be working on a current sale at a particular point in time. Each sale preserves the list of items sold. The main use case for this application is the Process Sale, in which the cashier processes a new sale. Figure 2 presents the System Sequence Diagram for the main success scenario. In this scenario, the cashier asks the system to make a new sale. Each product being purchased is recorded by the system. When no more products are left, the cashier finishes the sale and the system presents the sale total. Finally, the cashier indicates the amount payed by the customer and the system presents the balance.

2.2 Software Contracts

Software contracts have their roots in work on formal specification and verification of software programs, and mainly on Hoare logic [7]. B. Meyer ported the notion of program specification to the object-oriented paradigm, presenting in [11] a object-oriented design methodology based on software contracts. There, class level operations are axiomatically specified by means of pre- and postconditions.

In the Rational Unified Process, and particularly in Larman [10], software contracts are applied in an alternative scenario. Here, software contracts are Analysis artifacts which describe the semantics of a system level operation. In this scenario, the whole system is understood as an object, instance of a fictitious class System. The Domain Model of the system specifies the internal structure or state of such object, while the System Sequence Diagrams specify the operations that must be provided by the class System. Software contracts express the semantics of these operations by describing their effect on the system state. Different

Operation:	enterItem(id:Integer, qty:Integer)
Preconditions:	· There is a current Sale in the Register. · There exists a ProductSpecification with itemID equals to id in the ProductCatalog of the Register. · The value of qty is greater than 0.
Postconditions:	· The current Sale has a new SalesLineItem. · This line item has quantity equals to qty and corresponds to the ProductSpecification which itemID is id.

Fig. 3. Software contract for `enterItem()` in natural language.

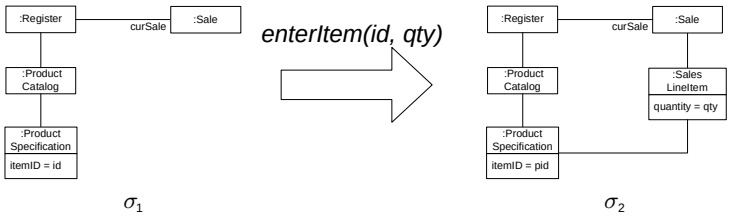


Fig. 4. Snapshots exemplifying the effect of `enterItem()`.

techniques and languages may be used for expressing a software contract. Figure 3 presents the software contract for the `enterItem()` system level operation in natural language. Snapshots illustrating the system state before and after the execution of this operation are shown in Figure 4. A formal version of this contract expressed in OCL is presented in Figure 7. Informal contracts are easier to define than formal contracts, however, the formal ones are more unambiguous the informal versions.

Software contracts are usually specified in terms of conditions (predicates) on states. System designers must extract fine grained responsibilities out from contracts and then design a solution that realizes them. In [10], software contracts are not expressed in terms of conditions, rather they directly list the actions that need to be performed. Figure 5 shows a software contract of this style. This kind of contracts go a step further into design than those based on conditions. This work is based on this latter kind of contracts as they focus on specification rather than on realization.

2.3 Object-Oriented Design

Object-oriented design involves the development of a Design Model which fully realizes all system responsibilities. From a dynamic perspective, this model is formed by a net of interacting objects that, altogether, fulfils the system's goal. This perspective is stated by means of communication diagrams. From a static point of view, the model is conformed by a set of related classes that determine the system internal structure which provide support for the object interactions.

Operation:	enterItem(id:Integer, qty:Integer)
Cross References:	Use Case Process Sale
Preconditions:	· There is a sale underway.
Postconditions:	· A SalesLineItem instance sli was created (instance creation). · sli was associated with the current Sale (association formed). · sli.quantity became qty (attribute modification). · sli was associated with a ProductSpecification, based on id match (association formed).

Fig. 5. Software contract for `enterItem()` proposed by C. Larman in [10].

Based on the specification of the system responsibilities of each system level operation, developers create an object interaction to realize it. To this end, the system responsibility is distributed among the objects; this approach is known as responsibility-driven design [10]. This activity is currently performed manually, requiring some extent of creativity. Although general guidelines exists, no systematic approach to extract a design from the specification of system responsibilities has been proposed. The General Responsibility Assignment Software Principles (GRASP) provide guidelines for achieving this goal. GRASP defines basic object-oriented principles or building blocks in design [10]. They aid developers in structuring the object interactions in order to obtain quality designs. No general methodology for applying these building blocks is available. Based on the fine grained responsibilities for a system level operation, developers decide which criteria to follow.

At system design, for each system level operation developers build a communication diagram describing the object interaction. This interaction assumes the preconditions of the operation because reaching such system state is responsibility of the operation caller. However, object interactions are in charge of fulfilling the postconditions. To this end, GRASP are applied, although seldom documented. Figure 6 presents the communication diagram for the `enterItem()` system level operation. Then, a class diagram is derived from the communication diagrams. It sums up the internal static structure of the system. This activity can be developed systematically, and also automatic approaches have been proposed [19]. Both communication and class diagrams provide a graphical representation of the Design Model.

2.4 Relational Model Transformations

Model Driven Engineering is a general approach based on domain specific languages and model transformations for aiding software development. Model transformations translate between source and target models. As indicated in [5], different techniques provide such functionality: direct model manipulation, graph-based, structure-driven, and relational.

In this work, direct model manipulation and relational approaches are used. In the first technique, models are instance of an object-oriented meta-model and transformations directly operate on source and target models by means of an

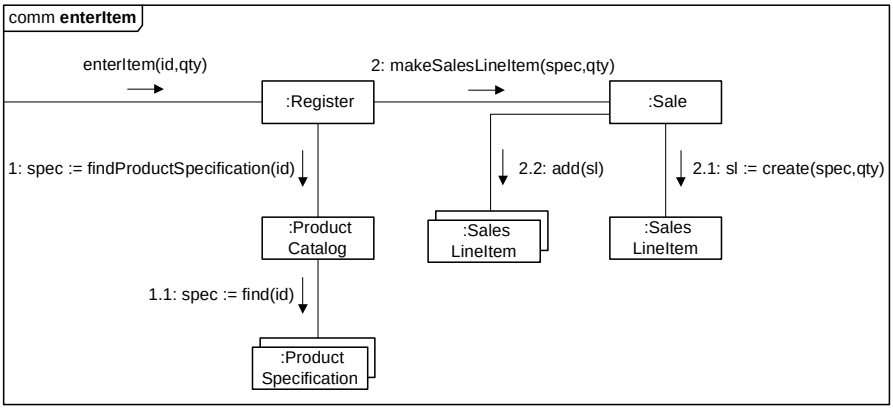


Fig. 6. Behavioral design for `enterItem()`.

API. An engine for such technique directly executes the code of the transformation; KERMETA is an example of such engine. In the relational technique, a transformation between candidate models is specified as a set of relations that must hold on the models for the transformation to be successful. QVT Relations is a relational transformation language, part of the QVT [12] specification. Relations in such language can be checked or enforced; an engine capable of executing these relational transformations can then modify the target model so as to satisfy the relationships.

3 Solution

For achieving our goal, we propose an approach based on four key ideas that strongly rely on Model Driven Engineering techniques: domain specific languages and model transformations. We also provide tool support for fully automating these techniques.

3.1 Approach

Extracting the system behavioral design out from software contracts of the system level operations is a hard task to accomplish in one single step. Several artifacts at different levels of abstraction are involved, and their semantics is not easily mapped. So building a transformation for such a task requires some sophistication. Several input models are weaved into a resulting model making use of some user assistance. This last issue prevents from achieving full automation; however, full automation can be obtained by early specifying some design decisions. There are four ideas that are key to our approach: divide and conquer, software contracts are model transformations, an extended transformation engine, and explicitly specified design decisions.

A software contract for system level operation specifies the behavioral effect of an operation on the system state. It states the system level responsibilities that must be carried out by any implementation for the corresponding operation. In object-oriented design, these responsibilities are distributed among the interacting objects in the system. We propose to divide this gap in two separate stages. The first stage consists of extracting the set of system level responsibilities out from software contracts. These responsibilities are expressed in terms of state modification primitives like create, set, link and unlink, as suggested by [13, 18]. Once this is done, the second stage is to assign them to the interacting objects that conform the system state, so as to obtain such interactions in terms of communication diagrams.

System level operations can be either system state queries or updates. Software contracts are of major interest when used for specifying the latter case. Here, a system level operation alters the system state, modifying the initial state s_1 to get a final state s_2. Then, a system level operation can be seen as a function or transformation from state to state. A model of the system state can be expressed in terms of a UML object diagram as the one in Figure 4. Let σ_1 and σ_2 be the models of the system states s_1 and s_2. A system level operation that takes the system from s_1 to s_2 can be seen as a model transformation of σ_1 into σ_2. The software contract specifies the effect that the operation has on the system state, and accordingly the transformation to be performed on the source instance model in order to obtain the target instance model.

Given that a system level operation is a function, two consecutive states of a system can be related by means of the operation that generates one from the other. The semantics of the modeling/programming language generally defines such relation. Another way of relating two consecutive states is by means of the software contract of the operation. Then, the initial state must satisfy the preconditions, the final state must satisfy the postconditions, and only what is specified can change. If two consecutive states satisfy such a relation, then there exist a program realizing the contract that transforms the initial state into the final state.

Our key idea is that a software contract can be itself a model transformation. There are several kinds of model transformations. Relational model transformations state a relationship between the source and target models, and thus they are quite similar to the specification of software contracts. Hence, to specify a software contract is equivalent to defining a relational model transformation from a model of the initial state into a model of the final state. This interpretation of software contracts as relational model transformations is aligned to some ideas explored in [4]. Also, following the classification of model transformation approaches proposed in [5], a software contract can be cataloged as a model transformation that in the Source-Target Relationship dimension corresponds to Existing Target, as the input system state is duplicated and considered as the initial output state. As this copy of the state is modified it is an Update transformation, which in particular is Destructive as certain elements can be possibly eliminated (e.g. links).

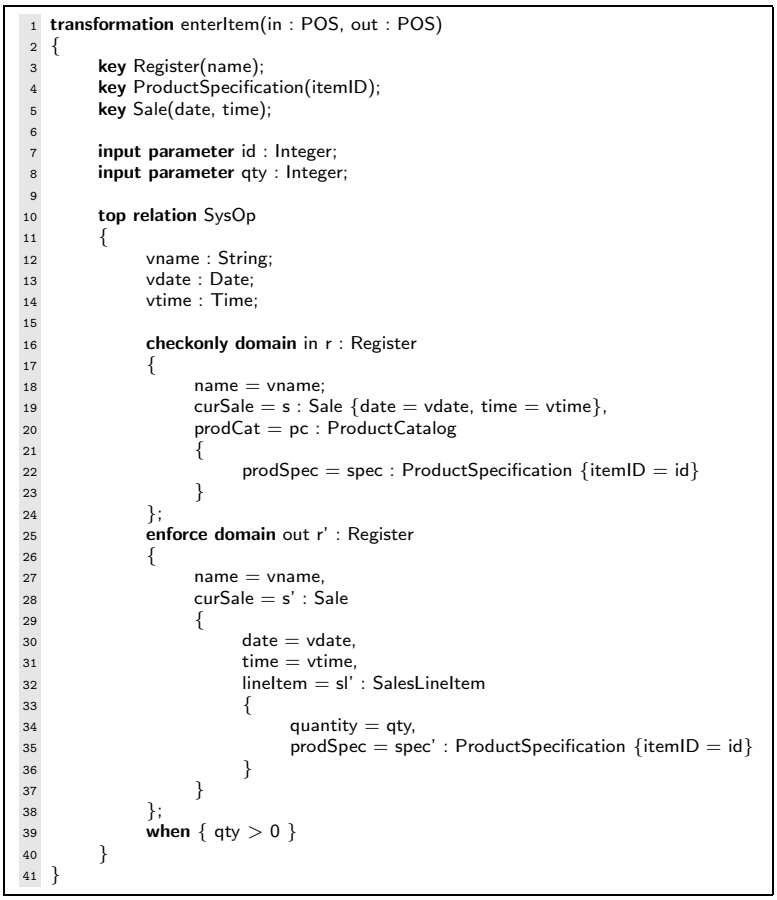
```
context System::enterItem(id:Integer, qty:Integer)
  pre:  not self.register.curSale.oclIsUndefined()
  pre:  self.register.prodCat.prodSpec→exists(itemID = id)
  pre:  qty > 0
  post: self.register.curSale.lineItem→one(x | x.oclIsNew())
  post: let ps = self.register.prodCat.prodSpec→any(itemID = id);
            s  = self.register.curSale;
            sli = s.lineItem→any(x:SalesLineItem | x.oclIsNew())
        in
            sli.prodSpec = ps and sli.quantity = qty
```

Fig. 7. Software contract for `enterItem()` in OCL.

To illustrate this idea, Figure 7 presents a formal version of the software contract for the `enterItem()` operation in OCL. In turn, Figure 8 presents the software contract as a relational transformation in QVT. Figures 7 and 8 present different versions of the same software contract: one in terms of OCL and other in terms of QVT. Given that a QVT relational transformation establishes the predicates that must hold in the source and target model, the transformation establishes what must be done and not how it must be done. An engine tool able to execute such kind of transformations should manage to make these predicates hold.

A relational transformation engine must be able to identify the set of actions that must be performed on the initial target model in order to obtain the final target model that satisfies the relations. For example in Figure 4, which shows the transformation from a initial target model σ_1 into a final target model σ_2 for the relational transformation of Figure 8, the engine must decide that a new SalesLineItem has to be created, that it must be linked to the current Sale and to the corresponding ProductSpecification, and that its attribute must be set. These actions correspond to the fine grained responsibilities that can be extracted from the software contract; these responsibilities are listed as a postcondition in the C. Larman's version of this contract in Figure 5.

For identifying fine grained responsibilities, given a software contract in terms of a relational transformation, we are more interested in the set of actions that needs to be performed than the resulting target model itself. Current implementations of QVT engines have still limited functionality, such as [15] and [16]. However, even a fully-compliant QVT engine would not produce as output a key element to our approach, that is, the set of performed actions. Our approach requires an extended engine for executing relational transformations. Such extension makes the engine to record information about the initial state of the models and also log the set of actions that needs to be performed in order to obtain the final model. This information is the execution trace of an operation. As noticed in [1], the output of the execution of a model transformation can be of different kinds: the resulting model itself, a list of atomic changes, or a special model reflecting the difference between the initial and the final model.



```
1  transformation enterItem(in : POS, out : POS)
2  {
3      key Register(name);
4      key ProductSpecification(itemID);
5      key Sale(date, time);
6
7      input parameter id : Integer;
8      input parameter qty : Integer;
9
10     top relation SysOp
11     {
12         vname : String;
13         vdate : Date;
14         vtime : Time;
15
16         checkonly domain in r : Register
17         {
18             name = vname;
19             curSale = s : Sale {date = vdate, time = vtime},
20             prodCat = pc : ProductCatalog
21             {
22                 prodSpec = spec : ProductSpecification {itemID = id}
23             }
24         };
25         enforce domain out r' : Register
26         {
27             name = vname,
28             curSale = s' : Sale
29             {
30                 date = vdate,
31                 time = vtime,
32                 lineItem = sl' : SalesLineItem
33                 {
34                     quantity = qty,
35                     prodSpec = spec' : ProductSpecification {itemID = id}
36                 }
37             }
38         };
39         when { qty > 0 }
40     }
41 }
```

Fig. 8. Relational transformation for `enterItem()` in QVT.

The second kind of output, named the execution trace in this work, is what is expected from the QVT engine.

Continuing with the example for the `enterItem()` system level operation, the execution trace obtained from the contract in Figure 8 is presented in Figure 9. The execution trace of Figure 9 is formed by three parts: the assumptions on lines 3 to 17, the actions on lines 19 to 24, and the design decisions on lines 26 to 28. Assumptions and actions can be actually extracted from the software contract while design decisions could only be suggested.

The second stage of our approach consists of assigning the fine grained responsibilities to system objects, generating the communication diagrams. To this end, the design decisions that need to be made must be specified so as to generate the desired artifact. GRASP provide a well-understood and broadly-accepted vocabulary for referring to responsibility assignment in object-oriented design.

```
1  xtrace enterItem
2  {
3      domain model POS;
4
5      input parameter qty : Integer;
6      input parameter id : Integer;
7
8      search key id of ProductSpecification;
9
10     object r : Register;
11     object s : Sale;
12     object pc : ProductCatalog;
13     object spec : ProductSpecification;
14
15     areLinked r, s, capturedOn;
16     areLinked r, pc, usedBy;
17     areLinked pc, spec, contains;
18
19     createAction sl : SalesLineItem
20     {
21         linkAction s, sl, containedIn;
22         setAction sl, quantity, qty;
23         linkAction sl, spec, describedBy;
24     }
25
26     creator Register of Sale;
27     creator Sale of SalesLineItem;
28     controller r;
29 }
```

Fig. 9. Resulting xTrace for `enterItem()`.

Our approach relies on the specification of which GRASP must be applied during design. Some basic GRASP as Creator, Expert and Controller might be extracted from the software contract and the Domain Model. However, tieing design decision to the system specification may be inappropriate in the general case. By separately specifying these decisions they get explicitly documented and the approach is more flexible as the generated artifacts can accommodate to different scenarios. To this end, we extend the execution trace resulting from the previous stage in order to include these decisions. They are expressed in terms of the application of GRASP. For example, Figure 9 shows these decisions in lines 26 to 28. The extended execution trace states that the Register class is the creator of Sale class (line 26) which in turn is the creator of SalesLineItem (line 27). In addition, line 28 states that the Register instance r is the controller of the system operation.

Overall structure of the solution Figure 10 presents the overall structure of the solution to the problem being address.

In the context of object-oriented development processes, requirement gathering is approached by building a Domain Model and writing down use-cases for functional requirements, which in turn are used for identifying system level operations. Each use-case scenario is analyzed and, as a result, a System Se-

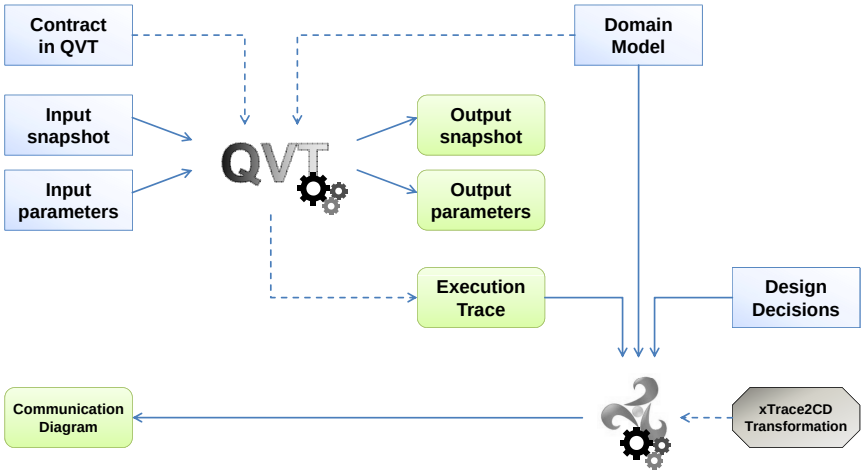


Fig. 10. Solution overall structure.

quence Diagram is build for each of them. For the POS System, the Domain Model is presented in Figure 1 and the System Sequence Diagram for its single use-case is shown in Figure 2. For each system level operation a software contract is built. Different strategies and languages can be used to this end. Our approach is based on specifying them as QVT relational transformations. Figure 8 presents the corresponding software contract for the `enterItem()` system level operation. Software contracts can be accompanied by snapshots in order to depict the effect of the operation (see Figure 4). Our approach requires the elaboration of an input snapshot satisfying the preconditions.

Once this artifacts are built, the extended QVT engine is used to process them and generate the expected results. This tool executes the QVT relational transformation as indicated by the software contract taking into account the system Domain Model. The execution takes as input the snapshot for the initial state of the system and the values for the transformation input parameters. The execution produces an updated system state, the values for the transformation output parameters, and the execution trace followed for manipulating the system state. The top part of Figure 10 depicts this stage. In our example, the outputted snapshot is shown in Figure 4 and the execution trace is presented in Figure 9. The execution trace lists the assumptions on the system state and the fine grained responsibilities of the system level operation.

The following step is to generate a communication diagram that carries out these responsibilities, and by this means, realizes the software contract. To this end, the obtained execution trace is extended with the design decisions in terms of GRASP. Then, Model Driven Engineering techniques are again applied. A model transformation processes the extended execution trace and produces the communication diagram. Accordingly to [3], this second stage is actually not a

transformation but a model weaving because there are two input models and there is no complete automation. We here provide a particular solution to the problem of integrating user guidance: the chosen design decisions are appended to the execution trace, as shown in Figure 9. In our example, the outputted diagram is shown in Figure 6. The produced diagram is not necessarily the best design that can be achieved, but it is a behavioral design that correctly and completely realizes the software contract of the operation and that follows the specified design decisions. This step requires an additional artifact not outputted by the previous step. By early specifying design decisions in a separate artifact, fully automation can be achieved. However, developers may prefer to try different design decisions. The proposed approach accommodates to both automatic and interactive toolset scenarios.

3.2 Companion Toolset

The proposed approach could be followed manually. However, to get the best out of the approach, a set of companion tools was developed so as to aid in the process. Two tools compose the toolset: a QVT Engine for executing software contracts expressed as QVT relational transformations, and a xTrace2CD model transformation to produce the communication diagrams. Both tools are illustrated in Figure 10. In the Figure, squared entities correspond to input artifacts created by the developer team and rounded entries represent artifacts generated by applying the approach and its companion toolset. The two tools composing the toolset are also depicted in the Figure: the QVT Engine and the xTrace2CD transformation. In what follows, we explain the overall architecture of these tools.

QVT Engine. The first step of the approach relies on the execution of QVT relational transformations. Our approach requires a nonstandard engine as additional output is expected from it. As a consequence, we developed a tool called QVT Engine which executes QVT relational transformations and outputs the expected artifacts. This tool is not a fully-complaint QVT engine; rather, it includes the additional capability while coping only with those constructs of QVT that were mandatory for implementing the case study. The QVT Engine is implemented in Prolog [17]. The Prolog execution model, based on unification and backtracking, favors the processing of relational transformations. This assessment is shared with [5], where the logic programming paradigm is suggested as the best fit for implementing relational model transformations.

Inputs and outputs of the tool are Prolog terms. Parsers and pretty-printers are planned for future work but they are not implemented yet. The QVT Engine receives several arguments for its execution. A relational transformation must be provided. QVT relational transformations were extended as we need to handle input and output parameters for the transformations; in other words, we developed a domain specific language, inspired in QVT Relations, for expressing software contracts of system level operations. Figure 8 is an example of usage

of such language. Additionally, a domain model must be provided; it consists of a PROLOG term indicating the set of classes and their properties. Associations are not first class citizens in our models as we follow the same approach as KERMETA where associations are defined as related pairs of properties in the associated classes. Also, the input consists of an initial state listing all existing objects and the values of their properties. Each object has a unique object identifier which is used for object referencing. The state preserves the next free object identifier which is used and updated when new objects are incorporated to the state. Finally, the values for the input parameters of the relational transformation must be provided. These input artifacts are the representation in PROLOG of those shown in Figure 10.

The execution of the tool proceeds in the following way. The initial state is duplicated so as to consider this copy as the first version of the final state. This final state is updated during the tool execution. Variables are allocated as soon as they are declared, and their value is set as soon as available. First, the QVT ENGINE processes the check condition identifying which combination of objects satisfy it. For each combination the enforce condition is processed; this procedure forces the condition to hold. To this end, the engine creates objects, sets their properties, and links and unlinks pairs of objects as required. Which action must be taken depends on the conditions in the enforce clause and the system state. After this condition is enforced, the when condition is processed to check whether it holds for the combination of objects identified in the execution of the check and enforce conditions. Whether the combination is unsuccessful, the engine backtracks so as to try with the subsequent combination. Later, the where condition is enforced, which generally sets the values for the output parameters. After finishing the execution, the final state, the values of the output parameters and the execution trace are outputted.

The architecture of the QVT ENGINE tool is based on two kinds of modules. On one hand, abstract data types were defined for each manipulated artifact and the execution state. PROLOG predicates were implemented to abstract away the inner structure of the PROLOG terms that represent the domain model, transformation, system state, execution trace and variable values. On the other hand, particular PROLOG modules were developed to process transformations and the involved conditions, and to evaluate OCL expressions. The condition processor consists of four predicates, each for processing conditions in the context of each kind of clause, namely check, enforce, when and where. Once a new kind of condition needs to be used, only these predicates need to be updated so as to handle it. The OCL expression evaluator was developed analogously. The implementation of all these predicates is based only on the predicates defined for the abstract data types modules.

Several challenges were confronted in the development of this tool, and two of them deserve further comment. A methodological challenge we faced up was the identification of which actions on the state must be performed so as to enforce a given condition. A general predicate expressed as an OCL boolean expression may be difficult to tackle. In the context of QVT, however, conditions are struc-

tured and each kind of them can be analyzed separately. Enforceable conditions as OCL boolean expression can only be present in the where clause, and this usage is strongly restricted in the current implementation. For each kind of condition, we analyze all possible system states, and based on this information, we decide which actions must be performed to get the condition to hold. A technological challenge we confronted was the backtracking nature of PROLOG. Several combinations of objects may apply to the transformation, and the backtracking mechanism explore them all. However, when a successful branch is reached, the state must be preserved so as to process the following branches. This was achieve be means of the backtracking-independent global variable assignment of PROLOG.

XTrace2CD. The second step of our approach relies on the execution of a model transformation or weaving named XTRACE2CD. Such transformation was implemented in KERMETA [6] and is responsible of producing a communication diagram from an extended trace. In this section we provide an overview of the transformation, a detailed example of its operation is presented in Section 4.2.

Concrete inputs for XTRACE2CD are the Domain Model and the extended trace. The extended trace includes the execution trace produced by QVT ENGINE and the design decisions in terms of GRASP as illustrated in Figure 10. Models manipulated by this transformation are expressed in XMI and are instances of ECore metamodels: ClassDiagrams and XTrace metamodels for source models, and CommunicationDiagrams metamodel for the target model.

XTRACE2CD operates as follows. The target model can be understood as a directed multigraph of objects, where a particular edge, called the *entry point* of the diagram, represents the message which starts the interaction. Such message corresponds to the system operation, and by convention it has no source object and it is handled by the operation controller. The first stage of the transformation consists of generating the entry point. In the second stage, the sequence of actions included in the extended trace is iteratively processed in order. For every single action a set of messages is produced for resolving the corresponding sub-interaction. It involves a number of messages for the action itself, and since the controller is ultimately the source of every action in the interaction, a path of messages from the object performing the action back to the controller. Arguments required for performing the action, such as input parameters or objects, are collected along such path.

Control is centralized in a class representing the transformation, which is also responsible of generating the entry point. Once the main loop is entered, action processing is delegated to specific classes specialized in processing each kind of action. We keep track of the system state by updating an internal structure as an action is processed. This state reflects the evolution of the interaction as it occurs, and is used to enforce action's preconditions. For example, an object cannot be linked to another before it was created. The initial configuration of such state satisfied the precondition of the system operation, and is taken from the execution trace.

Among the main challenges faced in the development of the transformation, two of them stand out. Given that the path of messages from the controller to the object that performs an action is constructed backward, the numeration of messages is not trivial. This is because other paths may have already been generated in the target model and that when a message is created its invocation context has not been decided yet. Second, the naming of the generated messages was not straightforward in many cases. In the complete case study four communication diagrams were generated, totalizing nineteen messages. Except for the message that asks for the subtotal of a sale line item when the sale total is calculated, names for the remaining eighteen messages were properly derived from the context, resulting similar to those chosen in the original case study in [10].

4 Case Study

The proposed approach and its companion toolset were validated on the case study introduced in Section 2.1. Currently, following this approach and applying the toolset, all system operations were successfully processed and all communication diagrams obtained. In order to illustrate the applicability of the approach and the way the toolset operates, in this section we show step by step how the resulting artifacts are generated. We circumscribe the presentation to the `enterItem()` system level operation of the case study.

First, we show how the QVT Engine extracts the fine grained responsibilities out of the software contract for the operation, also querying the Domain Model of the system shown in Figure 1. These responsibilities are expressed in terms of an execution trace. Second, this trace is extended so as to include the design decisions in terms of GRASP; this extended trace is shown in Figure 9. Thus, we show how the xTrace2CD transformation generates the corresponding communication diagram, presented in Figure 6.

4.1 Execution Trace Generation

In this section we explain how the QVT Engine proceeds in order to achieve the expected output by processing the input artifacts. Although both inputs and outputs must be expressed as Prolog terms, we develop the explanation in terms of the textual and graphical artifacts.

Four artifacts are the inputs for the QVT Engine: the Domain Model of POS system, the software contract expressed as a QVT relational model transformation shown in Figure 8, the initial state of the system expressed in terms of an snapshot as σ_1 in Figure 4, and a value for each transformation parameters. The outputs consist of the resulting system state σ_2 shown in Figure 4 and the execution trace presented in Figure 9. As the QVT Engine does not support in-place transformation, the initial state σ_1 is provided twice to the transformation: one received as `in` and the other as `out` in the transformation. The tool applies

the transformation updating the out system state, which initially corresponds to σ_1 and conforms the final state σ_2 when the tool finishes the execution.

The QVT Engine takes the QVT transformation from the input and processes it construct by construct. Let us follow the transformation line by line, analyzing the actions performed by the tool for executing them.

```
1 transformation enterItem(in : POS, out : POS)
```

This line declares the transformation's name and the participant models. The tool records this information in the execution state. Also, it records that in and out are attached to the corresponding copy of the input value σ_1. Although in the general case the input and output instances may differ in the model, our application of QVT transformation always relies on a model common to both instances. The execution trace is initialized as shown in lines 1 and 3 in Figure 9.

```
3 key Register(name);
4 key ProductSpecification(itemID);
5 key Sale(date, time);
```

From these lines the tool records which attributes are used for identifying instances of a given class. For example, line 3 states that Register instances are identified by its name attribute. This information is used later when we need to look for a particular instance in the model.

```
7 input parameter id : Integer;
8 input parameter qty : Integer;
```

These declarations imply the allocation of id and qty as constants attaching them to the values from the input. Output parameters may also be declared; this is the case of the `endSale()` operation. Output parameters are allocated as variables and their value is initialized to null. Also, parameters are recorded in the execution trace; see lines 5 and 6 in Figure 9.

```
10 top relation SystemOperation
```

Then, the single relation is processed. Such relation consists of variable declarations and check, enforce, when and where conditions. The execution of the relation proceeds as follows.

```
12 vname : String;
13 vdate : Date;
14 vtime : Time;
```

A variable is allocated for each declaration and its values is initialized as null. The actual variables' values are obtained when the check condition is processed.

```
16 checkonly domain in r : Register
```

The check condition declaration indicates on which model instance it operates on (in) and which kind of objects is going to be checked (Register). All instances of Register are attached to the variable r, one at a time, and the execution of the relation proceeds for each of them. As in σ_1 there is only one Register instance, the relation is processed for this instance only. Line 10 is recorded in the execution trace; see Figure 9. The check clause also states the condition that must hold for r, processed as follows.

```
18  name = vname;
```

This condition is checked on the name attribute of the object attached to r. As vname is an uninitialized variable, the tool assigns the property's value to it.

```
19  curSale = s : Sale {date = vdate, time = vtime},
```

curSale is the opposite association end of the capturedOn association between Register and Sale. The right-hand side of this condition is an object template construct which indicates that there must be a Sale instance as the value of the curSale that must satisfy the inner conditions. The tool checks whether there exists such an instance and allocates and initializes the variable s. Also, lines 11 and 15 are recorded in the execution trace. The inner conditions are checked by comparing an object property with an uninitialized variable which makes the variables' values to be updated.

```
20  prodCat = pc : ProductCatalog
21  {
22      prodSpec = spec : ProductSpecification {itemID = id}
23  }
```

Analogously, an instance for pc is located, recorded in the execution trace (lines 12 and 16), and the inner condition checked. The prodSpec property is a set of objects of class ProductSpecification. In this case, the object template condition behaves differently. The collection of objects is traversed and each object is checked with the inner condition. The objects that satisfy it are used, one at a time, in the remaining execution of the relation. Provided that there is a unique instance whose itemID matches the value of the input parameter id, just one object is to be considered. Variable spec is allocated and initialized, and lines 13 and 17 are recorded in Figure 9). Also, it is recorded that id is used as a search key for ProductSpecification instances; see line 8.

```
25  enforce domain out r' : Register
26  {
27      name = vname,
```

Analogous to the check clause, the enforce condition declares the model instance to work with (out), the class of objects to check (Register), and the variable to which each of these objects is allocated to (r'). Declared variables in the check and enforce conditions must be different. Their values reside on different model instances; a Register instance attached to r cannot be the same as the one attached to r'. Provided that the name attribute of class Register is declared as key (see line 3), the tool looks up a Register instance that matches this key condition. As the variable vname was initialized with the name of r, the object found is a Register instance in out whose name coincides with the name of r. Such object is allocated to r'. Both r and r' correspond to the very same object, r in the initial state and r' in the final state. Provided this relation between the initial and the final state, objects and links are not appended to the execution trace while processing the enforce condition. The actions performed to update the system state are expressed in terms of the related objects in the initial state, already declared while processing the check condition. The next condition must be enforced for r' as follows.

```
28 curSale = s' : Sale
29 {
30       date = vdate,
31       time = vtime,
```

The curSale property of r' is already attached to a Sale instance. The condition requires that this property must be attached to a Sale instance that satisfies the inner conditions. As date and time are keys of the class Sale (see line 5), that actually attached instance is checked to satisfy this key condition. If it holds, no action is taken. If it does not, the unique Sale instance that satisfies this condition would be looked up for and linked. If no such instance exists, a new one would be created and linked. We find this scenario later. Thus, keys are used to identify whether the already present objects are the ones expected and also, to look them up in the set of all instances of a given class. As the first case holds, s' is allocated and initialized and the inner condition is enforced as follows.

```
32 lineItem = sl' : SalesLineItem
33 {
34       quantity = qty,
```

s' may have zero or more SalesLineItem instances in its collections of lineItems; in our case no instance is present. No key is indicated for this class and then there is no mechanism to compare its instances. So, a new instance of SalesLineItem is created (as it cannot be looked up elsewhere) and appended to the collection of line items of s'. Variable sl' is allocated and initialized with this new instance. Additionally, a createAction is registered in the execution trace. Given that the inner conditions apply to the newly created instance, the resulting actions of their processing are declared as inner actions of the createAction in the execution trace. A linkAction is also registered in the execution trace indicating that s' and sl' have to be linked. To enforce the first inner condition, the quantity attribute of sl' is set to the value of the input parameter qty. The tool updates the final state out and records this fact in line 22 in Figure 9.

```
35 prodSpec = spec' : ProductSpecification {itemID = id}
```

As prodSpec has multiplicity at most 1, this condition requires to attach the corresponding instance of ProductSpecification by looking it up in the set of all instances of this class. Then, the link is created and recorded in the execution trace (line 23).

```
39 when { qty > 0 }
```

The when condition consisting of an OCL boolean expression is evaluated. As no where condition was specified and no other instances satisfy the check condition, the execution finishes.

Let us comment on a final remark. The when clause states a property that must hold on both instances of the input and output model instances. The usage of variables reduces the need of when conditions, which is used only for checking general preconditions on input parameters. The where clause is mainly used for stating the expected value of output parameters. This clause is used in the `endSale()` and `makePayment()` system level operations for indicating how the total and balance are obtained. The supported construct for this kind of clause is the

comparison of a uninitialized variable to an OCL expression. The expression is evaluated as assigned to the variable. Additionally, an assignAction is registered in the execution trace indicating the structure of the OCL expression.

4.2 Communication Diagram Generation

In this section we illustrate the operation of xTrace2CD transformation by reviewing how the inputs are handled for the `enterItem()` system level operation in order to produce the output.

The primary input for xTrace2CD is the extended trace shown in its textual form in Figure 9, which includes both the execution trace as generated by QVT Engine and the design decisions. The transformation also accepts the Domain Model depicted in Figure 1. The output of the transformation is the communication diagram shown in figure Figure 6.

The transformation executes as follows. First, it loads its input models. Then it initializes the state of the object interaction and creates an empty communication diagram. Finally, the entry point is created and the actions are processed. In what follows we detail these activities.

```
3 domain model POS;
```

This part of the extended trace contains information about the domain model that needs to be loaded, and is the first entry to be processed.

```
10 object r : Register;
11 object s : Sale;
12 object pc : ProductCatalog;
13 object spec : ProductSpecification;
14
15 areLinked r, s, capturedOn;
16 areLinked r, pc, usedBy;
17 areLinked pc, spec, contains;
```

Lines 10 through 17 are used for initializing the state of the object interaction. It consists of a graph where the nodes correspond to the declared objects and the edges to the links declared by areLinked. Note that object sl does not exist at the beginning of the object interaction and thus it is not present in the initial state.

```
1 xtrace enterItem;
```

The entry point of the communication diagram is a message. Its name is generated using the name of the trace itself, in this case enterItem.

```
5 input parameter qty : Integer;
6 input parameter id : Integer;
```

Arguments of the entry point are generated from the input parameters. If an output parameter was specified, then it would have been used to generate the return value of the message.

```
28 controller r;
```

By convention, the entry point has no source and its destination is the controller

that was selected for the operation. In this way, the entry point is fully generated. The remainder of the transformation consists of processing the sequence of actions included in the extended trace.

```
19 createAction sl : SalesLineItem
20 {
21     linkAction s, sl, containedIn;
22     setAction sl, quantity, qty;
23     linkAction sl, spec, describedBy;
24 }
```

In this example a single action is specified; the creation of object sl. As a particular case, a create action may have nested subactions. In [14] the creation of an object is considered in one single step, however for clarity, a separation of such action into object allocation and object initialization is proposed. In this work we follow the former approach, and therefore we allow nested actions within a single create action. These nested actions correspond to the initialization of the new object.

Line 19 causes the generation of an object named sl of class SalesLineItem in the communication diagram, and the update of the state. Also, a message named create is generated and such object is set as its destination. The source of the message is determined by knowing the creator of instances of class SalesLineItem. Line 27 of the extended trace indicates that such class is class Sale. Since as part of sl's initialization it must be linked to the object s in line 21, which is an instance of the creator class, the transformation concludes that s should be the source of the create message. Additionally, the Domain Model expresses that a Sale is composed of many line items. This motivates that such record needs to be reflected in the diagram. This causes the generation of a multiobject of class SalesLineItem representing the recorded line items. A message from s to such collection carrying sl as an argument next to the create message achieves this effect. The case of the action in line 23 is treated differently. The Domain Model indicates that a line item is to be connected to a single ProductSpecification. Therefore, no collection is needed and for sl to receive spec as an argument suffices.

The rest of the nested actions in lines 22 and 23 are used to generate the arguments of the create message, that is variable qty and object spec. Both elements must be accessible by s in order to be passed as arguments to sl. On the one hand, qty is an input parameter and therefore it is not directly accessible by s; on the other hand, a path from s to spec cannot be found in the stored state. For these reasons, both elements must be received as arguments by s before it sends the create message. In this way, message number 2 in Figure 6 is generated. Since it is an explicit order for object s for creating the line item, the name of such message is makeSalesLineItem. The source of that message is taken as the next to s in the shortest path from sl to r, which is the controller. In this case, such element is r itself. The process described above is repeated; the transformation checks if the arguments are accessible to r. Since qty is an input parameter, it is in fact accessible from r. Additionally, in the stored state there exists a path from r to spec, via the ProductCatalog pc. In that way, before the order to create

the line item is sent, spec must be reached. To that end, r retrieves spec from pc using id as a key by virtue of line 8, and the message findProductSpecification is generated. Finally, since a ProductCatalog has many specifications, in order to return spec, pc sends a find message to a collection of specifications, again using id as a key. This search needs to be performed by r before the order for creating sl is issued. For this reason, the sequence number in the search has 1 as a prefix, while the creation has number 2. This completes the generation of the communication diagram shown in Figure 6.

5 Conclusion

In this work we proposed an approach that aims at systematizing the activity of producing an object interaction from a software contract. The approach is supported by a toolset and was applied to a complete case study, showing the feasibility of the ideas presented here. Software contracts can be regarded as a valuable artifact, since important information can be extracted from them, and also such information can be used for producing other artifacts. This provides a compelling reason to specify software contracts formally.

This proposal is a practical application of several aspects of MDE. First, software contracts are now considered as models. Second, a non traditional application of relational model transformations is used for producing the execution trace; it is not the outcome of the transformation what is most important, and the transformation specification can be understood as an input for generating the trace. Third, a domain specific language was developed for expressing the extended execution trace, which in turn is considered itself as a model. Fourth, an imperative model transformation was defined for producing the expected communication diagrams.

The approach also provides a direct mechanism for testing software contracts. When a relational transformation T is executed on a given system state model σ_1 for producing the execution trace, a resulting model σ_2 is residually generated. Such model can be examined to check if the transformation produced the expected result, and therefore, in conjunction with σ_1, can be used to validate the software contract associated to T.

Software contracts expressed as QVT relational transformations can be considered as executable specifications. Although the QVT Engine tool currently works on object models, it can be extended to manipulate the system state residing outside the engine, e.g. as a persistent database. Such an approach allows developers to obtain a running system straight from its specification, bypassing design and implementation. This is useful for early validation and verification, and also, in certain development scenarios, such running system can be treated as the final system.

Both the proposed approach and the toolset present limitations. Although the POS System case study was completely and accurately solved, we identified particular scenarios which cannot be tackled yet by the current solution.

The proposed approach is strongly dependent on particular system states as input in order to obtain the resulting artifacts. Although this can be seen as a lack of generality, this is not the case. The generated design is only based on one generic initial state but several may be required. In other words, in the general case, it is not possible to define a system state (a snapshot) which satisfies a generic predicate of the form $A \vee B$ if A and B do not hold at the same time. This kind of scenarios leads to branching in the design, which is not currently covered. To overcome this limitation, several initial states can be processed for each system level operation, obtaining for each of them one communication diagram that solves each particular case. A merging mechanism must be defined so as to unify all these cases. To accomplish this idea and to study its automation is suggested as future work.

From the toolset perspective, the current implementation faces several limitations. The QVT Engine tool has partial support for primitive types, OCL expressions and relation conditions. Also, only create, set, link and unlink state modification primitives are considered, not considering object deletion. Nowadays deletion in object-oriented systems is not explicit; usually a companion garbage collector is in charge of freeing unneeded (unreachable) objects. However, the decision of deleting an object may be extracted out of software contracts in an analogous way as the other primitives. To explore this possibility, together with the development of a garbage collector, is also suggested as future work. Finally, as we are considering only one relation by relational transformation, we cannot define a contract which operates over two sets of objects; the check clause works on the set of instances of a given class and only objects accessed from them can be manipulated. To incorporate this feature is simple as it just implies an additional level of the backtracking tree: to try all relations. The xTrace2CD has limited support of GRASP. Currently, commonly used criteria as Pure Fabrication and the introduction of derived attributes and new associations is not covered, techniques which generally improve the design. To incorporate them is suggested as further work. As we mentioned above, we have no extensive applications of the proposed toolset so as to obtain significant data of design metrics like coupling and cohesion. By solving more case studies we would be able to collect this information, which in turn, provides feedback to improve the current implementation. It is important to notice that, in some way, the resulting design also depends on the design decisions specified by the developer in the extended execution trace. As a final limitation, neither tool considers the generalization relationship between classes. The impact of including this feature mainly falls on the xTrace2CD tool as decisions about overriding must be taken into account. Two possible ways to overcome this issue are to compare how each class in the hierarchy react to the same message so as to check if it can be generalized, and to incorporate this information as additional design decisions to be taken into account by the transformation.

References

1. Alanen, M., Lundkvist, T., Porres, I.: Reconciling Diagrams After Executing Model Transformations. In proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Model Transformation track, Dijon, France, 2006.
2. Beck, K.: A Laboratory for Teaching Object-Oriented Thinking. OOPSLA 1989 Conference Proceedings, New Orleans, Louisiana, USA, October 1-6, 1989.
3. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In proceedings of the European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, LNCS 3599. Uwe Aßmann, Mehmet Aksit, Arend Rensink editors, Springer Verlag, 2005.
4. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the Specification of Model Transformation Contracts. OCL and Model Driven Engineering Workshop, part of the Seventh International Conference on the Unified Modeling Language UML 2004, Lisbon, Portugal, 2004.
5. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. OOPSLA 2003 Workshop on Generative Techniques in the Context of MDA, Anaheim, CA, USA, 2003.
6. Fleurey, F., Drey, Z., Vojtisek, D., Faucher, C.: Kermeta Language, Reference Manual. Internet: `http://www.kermeta.org/docs/kermeta-manual.pdf`, 2006.
7. Hoare, C.A.R.: An Axiomatic Basis of Computer Programming. Communications of the ACM, Volume 12, 1969, 576–580.
8. Kaldewaij, A.: Programming: The Derivation of Algorithms. Prentice Hall, 1990.
9. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley, third edition, 2003.
10. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall, third edition, 2004.
11. Meyer, B.: Object-Oriented Software Construction. Prentice Hall, second edition, 1997.
12. OMG: Revised Submission for MOF 2.0 Query/View/Transformations RFP (ad/2002-04-10). OMG Document ad/2005-07-01, 2005.
13. Richters, M.: A Precise Approach to Validating UML Models and Constraints. Logos Verlag, Berlin, BISS Monographs, number 14, 2002.
14. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, Addison Wesley, second edition, 2005.
15. SmartQVT: An open source model transformation tool implementing the MOF 2.0 QVT-Operational language, Internet: `http://smartqvt.elibel.tm.fr/`, 2006.
16. Sriplakich, P.: Techniques des transformations de modèles basées sur la métamodélisation. D.E.A. report, Internet: `http://modfact.lip6.fr/qvtP.html`, 2003.
17. Sterling, S., Shapiro, E.: The art of Prolog: advanced programming techniques. MIT Press, second edition, 1994.
18. Vignaga, A.: A Formal Semantics of State Modification Primitives of Object-Oriented Systems. Master's thesis, Pedeciba and Universidad de la República, 2004.
19. Vignaga, A., Bastarrica, C.: Transforming System Operations' Interactions into a Design Class Diagram. To appear in proceedings of 22nd Annual ACM Symposium on Applied Computing (SAC2007), Model Transformations track. Seoul, Korea, 2007.