A Case Study for Evaluating Bidirectional Transformations in QVT Relations

Bernhard Westfechtel
Applied Computer Science I, University of Bayreuth, D-95440 Bayreuth, Germany

Keywords: Model-driven Software Engineering, Bidirectional Model Transformations, QVT Relational.

Abstract: In model-driven software engineering, high-level models of a software system are eventually transformed into executable code. Model transformations constitute a key technology for model-driven software engineering. QVT Relations (QVT-R) is a language for the declarative specification of model transformations which was defined in an OMG (Object Management Group) standard. In addition to unidirectional transformations, QVT-R supports bidirectional transformations: Rather than writing two unidirectional transformations separately, the user may provide a single relational specification which may be executed in both directions. In this way, the specification of a bidirectional transformation may be simplified considerably — which is crucial e.g. for round-trip engineering. This paper investigates a case study for evaluating QVT-R's capabilities for specifying bidirectional transformations. Even in this rather simple case study, development of a bidirectional transformation turns out to be more complex than expected. Motivated by the case study, we propose extensions to QVT-R which facilitate the specification of bidirectional transformations considerably.

1 INTRODUCTION

In *model-driven software engineering* (Schmidt, 2006), software development is driven by the construction of high-level models which are transformed over multiple stages into executable code. For model-driven software engineering, key enabling technologies are required for defining modeling languages as well as defining and executing model transformations.

In object-oriented modeling, the abstract syntax of a modeling language is defined by a *metamodel* (a structural class model). To this end, the Object Management Group (OMG) provides the *Meta Object Facility* standard (*MOF* (Object Management Group, 2013)), whose subset EMOF (Essential MOF) has been implemented in the Eclipse Modeling Framework (*EMF*) (Steinberg et al., 2009). (E)MOF is used widely for metamodeling.

In contrast, a wide variety of languages and tools for *model transformations* has been developed (Jakumeit et al., 2014). Model transformation languages are based on different computational paradigms (procedural, functional, rule-based, object-oriented), application and scheduling strategies, directionality (unidirectional vs. bidirectional), support for in-place or out-place, batch or incremental transformations, etc. (Czarnecki and Helsen, 2006).

In the light of this diversity, the OMG issued the *QVT* standard (*Queries, Views, and Transformations*), which defines a family of model transformation languages at different levels of abstraction (Object Management Group, 2014). The most high-level language is *QVT Relations* (*QVT-R*), which supports the declarative specification of transformations between MOF-based models. Furthermore, QVT-R reuses the *Object Constraint Language* (OCL), an expression language for MOF models (Object Management Group, 2012). QVT-R addresses a wide spectrum of model transformation scenarios, including enforcing and checking, unidirectional and bidirectional, batch and incremental, and n:1 transformations.

While QVT-R provides many interesting and promising features, the language has not been adopted widely. This stands in sharp contrast to OMG standards for defining modeling languages such as MOF and OCL. Thus, on the one hand, model transformations constitute a key enabling technology for model-driven software engineering. On the other hand, the QVT-R language, which is promoted as an OMG standard, has not gained significant acceptance so far.

These observations motivated our research into the *evaluation* of the QVT-R language. In particular, we investigate the following features of QVT-R: (1) *expressiveness* (what kinds of transformation prob-

Westfechtel B..
A Case Study for Evaluating Bidirectional Transformations in QVT Relations.
DOI: 10.5220/0005466801410155
In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2015), pages 141-155
ISBN: 978-989-758-100-7

lems may be solved?), (2) *conciseness* (may transformations be written concisely, at a high level of abstraction?), (3) *readability* (how easy is it to understand a transformation written in QVT-R?), and (4) *semantic soundness* (are the semantics of transformations defined clearly and in a unique way?).

This paper focuses on a particularly interesting feature of QVT-R: the declarative specification of *bidirectional transformations*. Rather than writing two unidirectional transformations separately, a transformation developer may provide a single relational specification which may be executed in both directions. This approach does not only save specification effort. In addition, it ensures the consistency of forward and backward transformations.

Bidirectional transformations are required in a variety of use cases of industrial relevance. For example, *round-trip engineering* integrates forward and reverse engineering into a coherent process which needs to be supported by bidirectional transformation tools. Furthermore, integration of heterogeneous tools calls for *data converters* which ideally perform lossless transformations in all directions.

The property of *bidirectionality* merely states that a given transformation definition may be executed in both directions. In addition, it is highly desirable that forward and backward transformations behave consistently. The strongest form of mutual consistency is *mutual invertibility*: By transforming back and forth in either direction, the original model is always reproduced. Unfortunately, mutual invertibility may not be achievable in the case of heterogeneous metamodels due to loss of information. Even then, however, bidirectional transformations with weaker consistency properties may still be achievable.

In this paper, we *evaluate* QVT-R's support for bidirectional transformations through a *case study* from the domain of *project management* (Kerzner, 1998). The case study deals with transformations between different kinds of activity networks for project planning: While *Gantt diagrams* are composed of activities related by dependencies of different types (end-start, start-start, etc.), *CPM networks* consist of events which are connected by activities. Their underlying metamodels are similar, yet heterogeneous. Any Gantt diagram may be transformed into an equivalent CPM network. The corresponding backward transformation may reconstruct the original Gantt diagram, but it works only for those kinds of CPM networks which may be generated by the forward transformation. This restriction is inherent to the transformation problem at hand and applies no matter which transformation language is used. Full mutual invertibility cannot be achieved here. The backward transformation inverts the forward transformation, but this statement does not always hold for the opposite direction: If the backward transformation is applied to some "inappropriate" CPM network, the forward transformation may not reconstruct the original network.

The case study is small enough to be presented in a single, self-contained paper. Though being small, it does provide some important insights into *potentials* and *limitations* of *QVT-R* with respect to the specification of bidirectional transformations. While we succeeded in solving the transformation problem at hand with a concise specification, development of the bidirectional transformation turned out to be more complex than expected. In particular, the QVT-R user has to consider the execution semantics of QVT-R in detail to ensure that the transformation is actually executable in both directions. Furthermore, the resulting specification is hard to read and understand. Finally, the case study reveals several issues concerning the definition of the dynamic semantics.

Motivated by the case study, we propose *extensions* to *QVT-R* which facilitate the specification of bidirectional transformations considerably. These extensions make specifications of bidirectional transformations easier to construct and understand. Furthermore, they remove some problems regarding the semantics definition of QVT-R. The proposed extensions may contribute to a wider adoption of QVT-R.

The rest of this paper is structured as follows: Section 2 introduces basic notions from the domain of model transformations. Section 3 briefly describes QVT-R. Section 4, the core part of this paper, presents the case study, including an evaluation and a proposal for revising QVT-R. Section 5 compares related work. Section 6 concludes the paper by summarizing and discussing our findings.

2 MODEL TRANSFORMATIONS

A *model* is an abstraction of a system allowing predictions or inferences to be made (Kühne, 2006). A model is expressed in a *modeling language*. A *metamodel* defines the abstract syntax of a modeling language, usually with the help of class diagrams and additional constraints (written e.g. in OCL). A model obeying the rules defined in a metamodel is called an *instance* of that metamodel.

A *model transformation* operates on a set of models, each of which may be either read, created, or updated. A *single-source-single-target transformation* takes a single *source model* as input and produces a single *target model* as output. A transformation is called *exogenous* (*endogenous*) if the source and tar-

get metamodels are different (the same). The case study presented in this paper deals with an exogenous single-source-single-target transformation.

A transformation may be modeled as a *function* from a source model to a target model. A transformation is *total* if it may be applied to any instance of the source metamodel, *deterministic* if it returns a unique target model for a given source model, *injective* if it returns different target models for different source models, *surjective* if it can generate any instance of the target metamodel, and *bijective* if it is both injective and surjective.

As defined so far, a transformation is *unidirectional*: It can generate an instance of the target metamodel from an instance of the source metamodel, but not vice versa. A *bidirectional transformation* is a pair of opposite unidirectional transformations (called *forward* and *backward transformation*).

In the context of round-trip engineering and bidirectional data converters, it is highly desirable that both forward and backward transformation are *total* and *mutually inverse*. Totality ensures that the transformations may be applied to all instances of the respective metamodels. Mutual inversion ensures consistent behavior: Applying the backward transformation after the forward transformation returns the original model; likewise for the opposite direction.

However, in practical applications these requirements may at best be approximated. If the forward and the backward transformation are both total and mutually inverse, there is a bijective (1:1) mapping between source models and target models. Such a mapping does not exist in the case of *heterogeneous metamodels*. This is illustrated in Figure 1, where c_{ij} denote concepts defined in metamodels (in terms of classes, attributes, or references): c_{11} cannot be mapped at all, c_{12} and c_{13} are mapped to the same concept c_{22}, c_{14} may be mapped nondeterministically to c_{23} or c_{24}, and c_{15} has to be simulated by the set of concepts $\{c_{25}, c_{26}\}$.

Thus, total and mutually inverse forward and backward transformations may be provided only in degenerate cases such as e.g. an endogenous copy transformation. In the case of exogenous transformations, neither the metamodels nor their instances may be mapped 1:1 onto each other; if the metamodels were equivalent, one of them could be simply discarded. However, weaker notions of mutual consistency may still be achieved. For example, the bidirectional transformation may be *forward invertible*: The forward transformation is total, and the backward transformation inverts the forward transformation. But applying the transformations in the opposite direction does not guarantee the reconstruction of the original model; rather, this property holds only for models generated by the forward transformation. The property of forward invertibility will be demonstrated by the case study presented in Section 4.

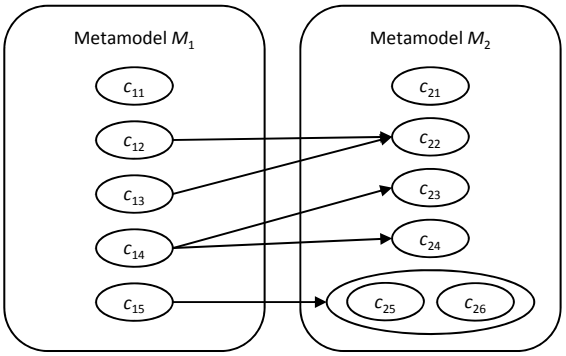

Figure 1: Mappings between heterogeneous metamodels.

3 QVT RELATIONS (QVT-R)

QVT-R is a declarative language for specifying both uni- and bidirectional model transformations as well as consistency checks. This section gives a brief overview of the concepts underlying QVT-R; see (Object Management Group, 2014) for a comprehensive description and (Reddy et al., 2006) for a tutorial.

In QVT-R, a *transformation* may be defined on $n \geq 2$ models; here, we assume $n = 2$. In this case, a transformation defines a relation $T \subseteq M_1 \times M_2$, where M_1 and M_2 denote sets of models.

A transformation is specified in terms of rules, each of which defines a *relation* between source and target patterns. A relation consists of *domains*, each of which defines a pattern in one model. A domain has a unique root object and is marked by a *domain qualifier* (checkonly or enforce) which controls how the domain may be used in a transformation (see below). Furthermore, a relation may have a when *clause* which serves as a precondition for applying the relation. A relation may also comprise a where *clause* which essentially acts as a postcondition. Finally, *variables* may be declared in a relation which are used in domains, the when and the where clause.

QVT-R offers two modes of execution. In the *checking mode*, the transformation is executed on two models $m_1 \in M_1$ and $m_2 \in M_2$. For each relation and each instance of a source pattern in m_1, a corresponding target pattern instance is searched in m_2 and vice versa. If all of these checks succeed, $T(m_1, m_2)$ holds. Domain qualifiers are immaterial in the checking mode.

In the *enforcing mode*, the transformation is exe-

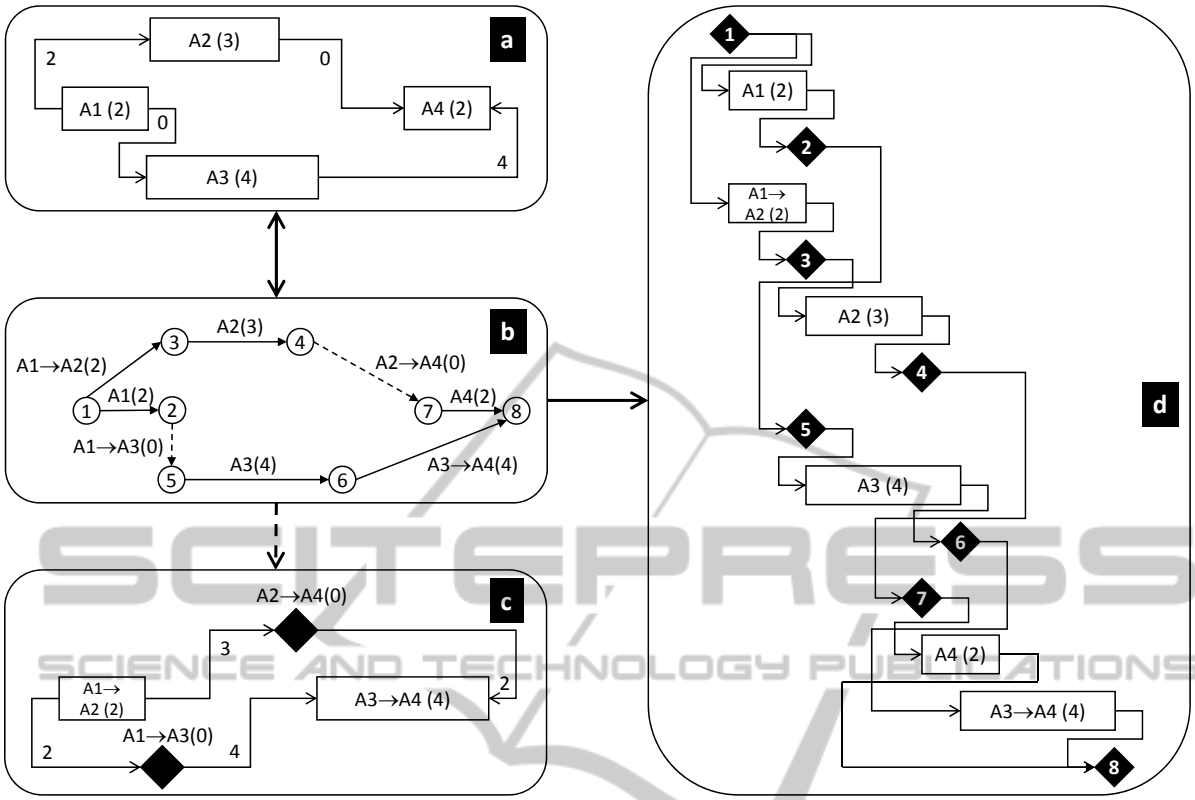

Figure 2: Gantt diagrams and CPM networks.

cuted either in forward or backward direction. If the transformation is executed on a given pair of models m_1 and m_2 in forward direction, m_2 is updated to establish consistency (and m_1 remains unchanged); likewise for the opposite direction. A *batch transformation* is treated as a special case (m_2 is empty).

To establish consistency, it is checked for each relation and each source pattern instance whether a corresponding instance of the target pattern already exists. If there is no such instance, it is created if the target pattern is qualified by enforce; otherwise, an inconsistency is reported. Furthermore, the check is performed also in the opposite direction. If there is no matching source pattern instance, the target pattern instance is deleted if its domain is qualified by enforce; otherwise, an inconsistency is reported.

Ideally, a transformation may be executed both in the checking mode and the enforcing mode in either direction. Thus, a *bidirectional transformation* may be defined by a single specification. In this case, all domains should be qualified by enforce. For a *unidirectional transformation*, source and target domains should be qualified by checkonly and enforce, respectively. Transformations which are run only in checking mode should mark all domains by checkonly.

4 CASE STUDY

To evaluate QVT-R's capabilities for specifying bidirectional transformations, we performed a case study in the *project management* domain (Kerzner, 1998). The case study is kept deliberately small so that it may be presented in a self-contained paper. It deals with *bidirectional data conversion* between different kinds of *activity networks* for project scheduling, namely *Gantt diagrams* and *CPM networks*.

The bidirectional transformation is developed in multiple steps. First, we develop a forward transformation from Gantt diagrams to CPM networks. Second, we construct a corresponding backward transformation. In the third step, we synthesize the unidirectional transformations into a single bidirectional transformation. After that, we evaluate the case study with respect to the criteria mentioned in the introduction and conclude with some proposals for revising QVT-R.

4.1 Activity Networks

A *Gantt diagram* is an acyclic graph whose nodes and edges represent *activities* and *dependencies*, respec-

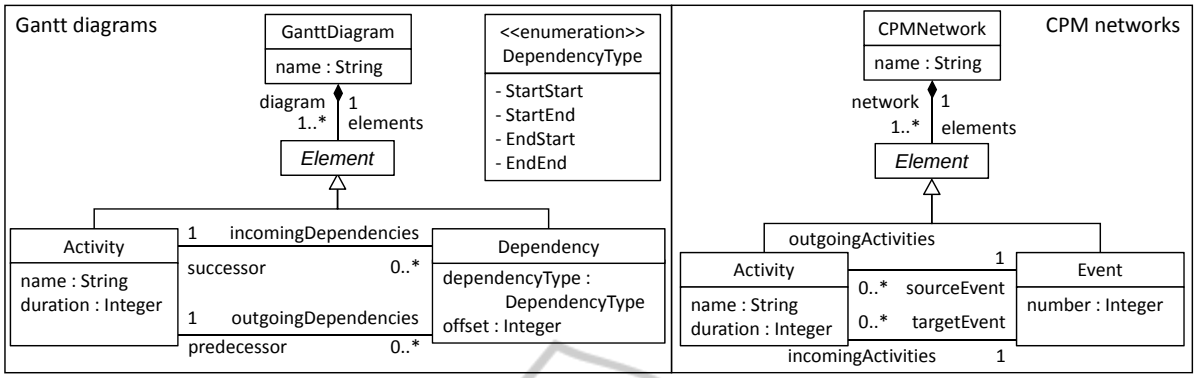

Figure 3: Metamodels for Gantt diagrams and CPM networks.

tively (Figure 2a,c,d). For each activity (time-scaled bar), a name and a duration is specified. An activity with duration 0 is a *milestone* (black diamond). A dependency from activity a_1 to a_2 defines an ordering constraint: a_2 can be started/finished only after a_1 has been started/finished. Dependencies are classified into four types: *start-start*, *start-end*, *end-start*, and *end-end*. Each dependency has an *offset*[1], which describes the delay between the start/end of the predecessor and the successor activity, respectively.

In contrast to a Gantt diagram, a *CPM network* (Critical Path Method) is not time-scaled (Figure 2b). CPM networks differ from PERT charts only in the treatment of durations (which are deterministic in a CPM network and stochastic in a PERT chart); this difference is immaterial in the context of this paper.

A CPM network is an acyclic graph whose nodes and edges represent *events* and *activities*, respectively. An event may occur after its incoming activities have been finished. Conversely, an activity may start after its source event has occurred. Events do not consume time. An activity has a name and a duration. Events are illustrated as circles which are labeled with unique numbers. An activity is shown as an arrow labeled with its name and duration. A dashed arrow indicates a *pseudo activity* with duration 0.

Both kinds of activity networks may be employed for *scheduling*. From the data provided in a Gantt diagram, earliest/latest start/finish times of activities may be calculated automatically. Activities whose earliest and latest start times coincide are located on a *critical path*. Similar calculations may be performed in CPM networks (e.g., earliest/latest times of events).

Figure 3 displays the *metamodels* for Gantt diagrams and CPM networks on which our QVT-R transformations are based. The metamodels are defined as class diagrams in Ecore, EMF's implementation of EMOF. In a Gantt diagram, both activities and dependencies are represented as objects; likewise for events and activities of CPM networks. Our Ecore models do not include derived attributes for calculated times because the model transformations deal only with intrinsic data.

The metamodels are closely related since Gantt diagrams and CPM networks are essentially used for the same purpose. However, the metamodels are still heterogeneous; concepts may not be mapped 1:1 to each other. First, a Gantt activity is mapped not only to a CPM activity, but also to a source and a target event. Second, the different types of Gantt dependencies have to be simulated by CPM activities connecting appropriate events. Third, event numbers have no correspondence in Gantt diagrams.

4.2 Forward Transformation

A Gantt diagram is transformed into an equivalent CPM network as follows: Each Gantt activity is mapped to a source event, a target event, and a connecting CPM activity with the same name and duration. The events are numbered in a unique way. Each Gantt dependency is mapped to a CPM activity, as well. The type of the Gantt dependency determines the source and target events; for example, in the case of a start-start dependency the source event of the predecessor activity is connected to the source event of the successor activity. The name of the CPM activity is composed from the predecessor and successor names, and the duration is copied from the dependency's offset. With these rules, the diagram of Figure 2a is transformed into the network of Figure 2b.

Figure 4 shows the *QVT-R specification* of the *forward transformation*. The transformation operates on a diagram and a network (line 1). The direction of the transformation is not specified explicitly, but can be inferred from the domain qualifiers of the relations.

The relation Diagram2Network (lines 2–6) maps a

[1]To simplify matters, we assume non-negative offsets.

```
1  transformation forward(diagram : gantt, network : cpm) {
2      top relation Diagram2Network {
3          name : String;
4          checkonly domain diagram gd_diagram : gantt::GanttDiagram {name = name};
5          enforce domain network cn_network : cpm::CPMNetwork {name = name};
6      }
7      top relation Activity2Activity {
8          name : String; duration : Integer;
9          sourceEventNumber, targetEventNumber : Integer;
10         checkonly domain diagram gd_activity : gantt::Activity {
11             name = name, duration = duration,
12             diagram = gd_diagram : gantt::GanttDiagram {}
13         };
14         enforce domain network cn_activity : cpm::Activity  {
15             name = name, duration = duration,
16             network = cn_network : cpm::CPMNetwork {},
17             sourceEvent = cn_sourceEvent : cpm::Event {
18                 network = cn_network, number = sourceEventNumber
19             },
20             targetEvent = cn_targetEvent : cpm::Event {
21                 network = cn_network, number = targetEventNumber
22             }
23         };
24         when {
25             Diagram2Network(gd_diagram, cn_network);
26         }
27         where {
28             sourceEventNumber = 2*noOfActivity(gd_activity) - 1;
29             targetEventNumber = 2*noOfActivity(gd_activity);
30         }
31     }
32     top relation Dependency2Activity {
33         cn_predActivity, cn_succActivity : cpm::Activity;
34         gd_dependencyType : gantt::DependencyType;
35         offset : Integer; name : String;
36         checkonly domain diagram gd_dependency : gantt::Dependency {
37             offset = offset,
38             diagram = gd_diagram : gantt::GanttDiagram {},
39             predecessor = gd_predActivity : gantt::Activity {},
40             successor = gd_succActivity : gantt::Activity {},
41             dependencyType = gd_dependencyType
42         };
43         enforce domain network cn_activity : cpm::Activity {
44             duration = offset, name = name,
45             network = cn_network : cpm::CPMNetwork {},
46             sourceEvent = cn_sourceEvent : cpm::Event {},
47             targetEvent = cn_targetEvent : cpm::Event {}
48         };
49         when {
50             Diagram2Network(gd_diagram, cn_network);
51             Activity2Activity (gd_predActivity, cn_predActivity);
52             Activity2Activity (gd_succActivity, cn_succActivity);
53         }
54         where {
55             cn_sourceEvent = sourceEvent(cn_predActivity, gd_dependencyType);
56             cn_targetEvent = targetEvent(cn_succActivity, gd_dependencyType);
57             name = gd_predActivity.name + '->' + gd_succActivity.name;
58         }
59     }
60 }
```

Figure 4: Forward transformation.

diagram to a network of the same name. The relation Activity2Activity (7–31) transforms a Gantt activity to a CPM activity. Its source pattern consists of a Gantt activity; pattern matching binds variables for the name, the duration, and the enclosing diagram. Its when clause ensures that the relation is applied only after the diagram has been transformed into a network. Furthermore, the relation call is used to retrieve the corresponding network object. The target pattern consists of a CPM activity whose name and duration are copied from the Gantt activity. Furthermore, the activity is inserted into the network which has been retrieved via the relation call in the when clause. Source and target events are created along with the activity; their numbers are calculated in the where clause. To this end, the number of the Gantt activity — say n — is determined from the source model (its index in the list for the containment reference from the diagram). Source and target event are assigned the unique numbers $2n-1$ and $2n$.

noOfActivity (lines 28–29) is a *query*, i.e., a function whose body is specified by an OCL expression. Queries may be defined as part of a QVT-R specification; due to the lack of space, all query definitions are omitted in this paper.

The relation Dependency2Activity (32–59) maps a Gantt dependency to a CPM activity. The source pattern is composed of a dependency as well as its predecessor and successor activities. The when clause ensures that the enclosing diagram and the predecessor and successor activities have already been transformed. The target pattern consists of the activity to be created and the (already existing) events to which it is to be connected. The variables for the source event and the target event are bound in the where clause, taking the dependency type into account. Furthermore, the where clause composes the name of the activity for the dependency from the names of the predecessor and successor activities. Finally, the duration of the activity is copied from the offset of the dependency.

The forward transformation performs a straightforward *syntax-based translation* of a Gantt diagram into a CPM network: The source model is transformed in terms of syntactic elements — the diagram and its composed activities and dependencies — into the target model. The transformation is total, deterministic and injective, but it is not surjective: For example, the CPM network $1 \to A1 \to 2 \to A2 \to 3$ cannot be generated by the forward transformation because $A1$ and $A2$ share the common event 2.

4.3 Backward Transformation

The *backward transformation* may be performed in different ways, meeting different requirements. A syntax-based transformation which may be applied to any network (including e.g. $1 \to A_1 \to 2 \to A_2 \to 3$) maps each event to a milestone, each real activity to an activity, and each pseudo activity to a dependency. Figure 2d demonstrates that this backward transformation does not invert the forward transformation. In general, the resulting Gantt diagram contains at least twice as many activities as the original Gantt diagram. Thus, an iterative application would generate a sequence of monotonically growing Gantt diagrams.

Instead, we develop a *pattern-based backward transformation* which inverts the forward transformation (but is partial rather than total). In the backward transformation, it has to be decided whether a network activity is transformed to an activity or a dependency in the diagram. This decision cannot be performed uniquely using only structural information. For example, we may attempt to calculate a *maximal matching*, i.e., a maximal set of edges such that no node is incident to more than one edge in the set, and transform the edges in the matching back to activities. For this purpose, the algorithm of Edmonds may be used (see (Jungnickel, 2008), Section 13.4). However, the maximal matching is not unique, as demonstrated in Figure 2b,c.

To avoid non-deterministic behavior, the backward transformation relies on *coding conventions* applied in the forward transformation. For example, a CPM activity which was generated from a Gantt activity may be recognized with the help of event numbers: The number n of its source event is odd, and the number of its target event is $n+1$.

In the specification of the backward transformation (Figure 5), the relation Activity2Activity maps a CPM activity to a Gantt activity of the same name and duration when the query MapToActivity, which exploits the coding convention explained above, returns true (line 15). Similarly, the relation Activity2Dependency is applicable only when the complementary query MapToDependency evaluates to true (line 37). From the source and the target event, the respective owning activities are retrieved, again relying on coding conventions (38–39). From these CPM activities, the corresponding activities in the Gantt diagram are retrieved by relation calls (40–41). The type of the dependency is determined in the where clause (44).

4.4 Bidirectional Transformation

The bidirectional transformation is synthesized from the unidirectional transformations by merging corresponding relations. All domains are qualified by enforce. Domains, when and where clauses are merged

```
1  transformation backward(network : cpm, diagram : gantt) {
2      top relation Network2Diagram {...}
3      top relation Activity2Activity {
4          name : String; duration : Integer;
5          checkonly domain network cn_activity : cpm::Activity  {
6              name = name, duration = duration,
7              network = cn_network : cpm::CPMNetwork {}
8          };
9          enforce domain diagram gd_activity : gantt::Activity {
10             name = name, duration = duration,
11             diagram = gd_diagram : gantt::GanttDiagram {}
12         };
13         when {
14             Network2Diagram(cn_network, gd_diagram);
15             mapToActivity(cn_activity);
16         }
17     }
18     top relation Activity2Dependency {
19         cn_predActivity, cn_succActivity : cpm::Activity;
20         gd_dependencyType : gantt::DependencyType;
21         offset : Integer;
22         checkonly domain network cn_activity : cpm::Activity {
23             duration = offset,
24             network = cn_network : cpm::CPMNetwork {},
25             sourceEvent = cn_sourceEvent : cpm::Event {},
26             targetEvent = cn_targetEvent : cpm::Event {}
27         };
28         enforce domain diagram gd_dependency : gantt::Dependency {
29             offset = offset,
30             diagram = gd_diagram : gantt::GanttDiagram {},
31             predecessor = gd_predActivity : gantt::Activity {},
32             successor = gd_succActivity : gantt::Activity {},
33             dependencyType = gd_dependencyType
34         };
35         when {
36             Network2Diagram(cn_network, gd_diagram);
37             mapToDependency(cn_activity);
38             cn_predActivity = owningActivity(cn_sourceEvent);
39             cn_succActivity = owningActivity(cn_targetEvent);
40             Activity2Activity (cn_predActivity, gd_predActivity);
41             Activity2Activity (cn_succActivity, gd_succActivity);
42         }
43         where {
44             gd_dependencyType = dependencyType(cn_activity);
45         }
46     }
47 }
```

Figure 5: Backward transformation.

component-wise. Merging is straightforward for domains: The respective domain patterns coincide in common parts and contain only elementary expressions (variables and constants). Thus, the union of patterns is well-defined, and matching may be performed in both directions. In contrast, manual adjustments may be required after a union of the when and where clauses to ensure bidirectional executability.

Execution of a relation involves the following kinds of steps: *match* an object in the source domain, *bind* a variable, *call* a relation, *check* a constraint, *create* a target domain object, or *assign* an object property. The order in which these steps are executed must satisfy *data flow constraints* which are described in the QVT-R standard ((Object Management Group, 2014), Section 7.5), and does not necessarily match the textual order in the specification. Depending on the binding states of the variables, equations may play a dual role: If all variables are bound, an equation acts as a check; otherwise, it is used to bind an object property or assign a value to a variable. In the case of a bidirectional transformation, it has to be

```
1   top relation Activity2Activity {
2       name : String; duration : Integer;
3       sourceEventNumber, targetEventNumber : Integer;
4       enforce domain diagram gd_activity : gantt::Activity {
5           name = name, duration = duration,
6           diagram = gd_diagram : gantt::GanttDiagram {}
7       };
8       enforce domain network cn_activity : cpm::Activity  {
9           name = name, duration = duration,
10          network = cn_network : cpm::CPMNetwork {},
11          sourceEvent = cn_sourceEvent : cpm::Event {
12              network = cn_network, number = sourceEventNumber
13          }, -- Forward transformation
14          targetEvent = cn_targetEvent : cpm::Event {
15              network = cn_network, number = targetEventNumber
16          } -- Forward transformation
17      };
18      when {
19          Diagram2Network(gd_diagram, cn_network);
20          if gd_activity.oclIsUndefined() then
21              mapToActivity(cn_activity)
22          else
23              true
24          endif; -- Backward transformation
25          sourceEventNumber =
26              if cn_activity.oclIsUndefined() then
27                  2*noOfActivity(gd_activity) - 1
28              else
29                  cn_activity.sourceEvent.number
30              endif; -- Forward transformation
31          targetEventNumber =
32              if cn_activity.oclIsUndefined() then
33                  2*noOfActivity(gd_activity)
34              else
35                  cn_activity.targetEvent.number
36              endif; -- Forward transformation
37      }
38  }
```

Figure 6: Bidirectional transformation: Activities.

ensured that the data flow constraints are satisfied for both directions.

A bidirectional relation may contain *unidirectional elements*, i.e., elements which are relevant for only one transformation direction. QVT-R does not provide language constructs for designating elements as unidirectional. The QVT-R user has to ensure that conceptually unidirectional elements do not prevent execution of the relation in the opposite direction.

Figure 6 displays the bidirectional relation between Gantt and CPM activities. Unidirectional elements are annotated with comments. While sourceEvent and targetEvent are relevant only for the forward direction, they do no harm in the backward direction. However, the unidirectional expressions in the when and the where clause cannot be copied literally from the forward and the backward relation.

In the backward relation, the call of the query mapToActivity (line 15 in Figure 5) checks that the CPM activity actually represents a Gantt activity. If this call is copied literally to the when clause of the bidirectional relation, execution in the forward direction aborts with a runtime error because the variable cn_activity is still unbound. Thus, the evaluation of the query is performed in a *conditional expression* (lines 20–24 in Figure 6): The transformation direction is queried by accessing the *binding state* of the variable gd_activity. If the variable is unbound, the relation is executed in backward direction, and the query mapToActivity is evaluated. Otherwise, the constraint is ignored by evaluating the constant true.

A similar problem occurs in the calculation of the event numbers. In the forward relation, the numbers are calculated in the where clause from the number of the Gantt activity in the opposite domain (lines 28–29 in Figure 4). Therefore, we rewrite these expressions into conditional expressions (lines 25–36 in Figure 6). If these expressions are evaluated in backward direc-

```
1   top relation Dependency2Activity {
2       cn_predActivity, cn_succActivity : cpm::Activity;
3       gd_dependencyType : gantt::DependencyType;
4       offset : Integer;
5       name : String;
6       enforce domain diagram gd_dependency : gantt::Dependency {
7           offset = offset,
8           diagram = gd_diagram : gantt::GanttDiagram {},
9           predecessor = gd_predActivity : gantt::Activity {},
10          successor = gd_succActivity : gantt::Activity {},
11          dependencyType = gd_dependencyType
12      };
13      enforce domain network cn_activity : cpm::Activity {
14          duration = offset,
15          network = cn_network : cpm::CPMNetwork {},
16          name = name,
17          sourceEvent = cn_sourceEvent : cpm::Event {},
18          targetEvent = cn_targetEvent : cpm::Event {}
19      };
20      when {
21          Diagram2Network(gd_diagram, cn_network);
22          if gd_dependency.oclIsUndefined() then
23              mapToDependency(cn_activity)
24          else
25              true
26          endif; -- Backward transformation
27          cn_predActivity = owningActivity(cn_sourceEvent); -- Backward transformation
28          cn_succActivity = owningActivity(cn_targetEvent); -- Backward transformation
29          Activity2Activity (gd_predActivity, cn_predActivity);
30          Activity2Activity (gd_succActivity, cn_succActivity);
31          gd_dependencyType =
32              if gd_dependency.oclIsUndefined() then
33                  dependencyType(cn_activity)
34              else
35                  gd_dependency.dependencyType
36              endif; -- Backward transformation
37      }
38      where {
39          cn_sourceEvent = sourceEvent(cn_predActivity, gd_dependencyType);
40          -- Forward transformation
41          cn_targetEvent = targetEvent(cn_succActivity, gd_dependencyType);
42          -- Forward transformation
43          name = gd_predActivity.name + '->' + gd_succActivity.name;
44          -- Forward transformation
45      }
46  }
```

Figure 7: Bidirectional transformation: Dependencies.

tion, they return the value to which the variable has been bound in pattern matching. Please notice that these expressions were moved from the where clause to the when clause. In the where clause, the expressions would not work because the variable cn_activity is already bound (by instantiation of the target pattern).

The bidirectional relation Dependency2Activity is constructed similarly (Figure 7). Again, all unidirectional expressions are annotated by comments. In lines 22–26 and 31–36, we constructed conditional expressions querying the binding state, as demonstrated above. Since this approach does not work in the where clause, the expression for the dependency type was moved to the when clause.

For each transformation direction, we have to check whether the data flow constraints described in the QVT-R standard (see Section 7.5, p. 18) are satisfied. Figure 8 demonstrates this for the forward direction. To this end, a graph is constructed which contains a node for each expression occurring in the relation. Each node is partitioned into three compartments. The first compartment references the line number of the expression. The second compartment specifies the action to be performed. If the action re-

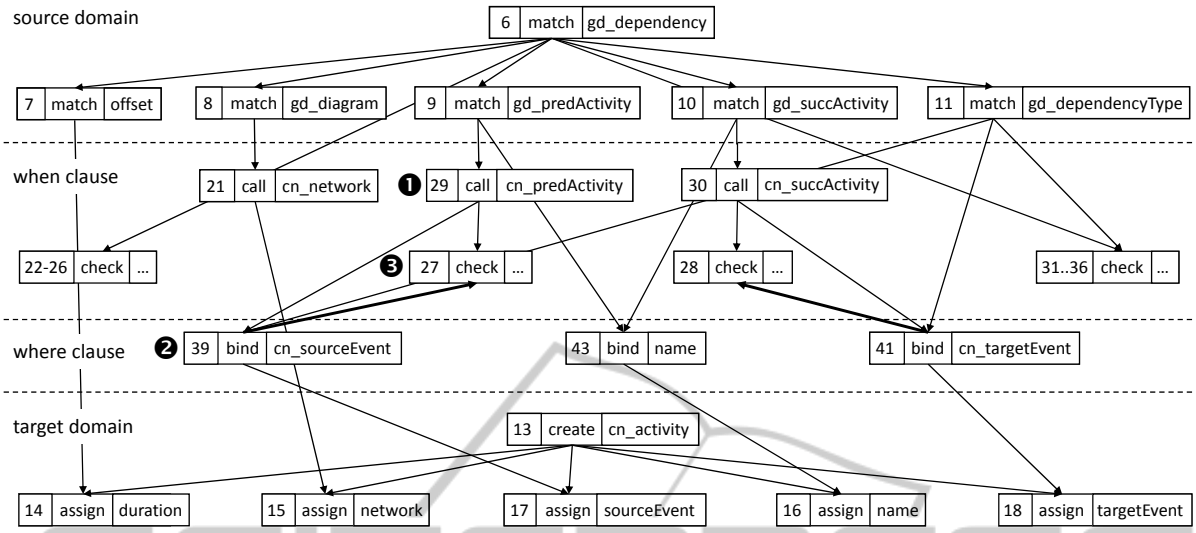

Figure 8: Dependency graph for the forward execution of Dependency2Activity.

sults in the binding of some variable, the name of this variable is shown in the third compartment. Arrows represent *data flow dependencies*. An arrow from node *s* to node *t* implies that *s* binds a value to some variable which is used in *t*.

The graph is acyclic, but contains an anomaly: Two arrows point upwards from nodes in where clauses to nodes in when clauses. For example, we obtain the following chain of actions: (1) Bind cn_predActivity by a relation call (when clause, line 29). (2) Bind cn_sourceEvent by navigation in the target model (where clause, line 39). (3) (Redundantly) check the equation in line 27 of the when clause.

Dependencies of the when clause on the where clause are allowed in QVT-R. Section 7.5 of the standard demands only that the expressions of a relation may be arranged into a consistent order of evaluation satisfying the data flow constraints.

While the bidirectional transformation behaves as required in enforcing mode in both directions, it does not work correctly in *checking mode*. The simulation of unidirectional expressions by conditional expressions querying the binding state implies that these expressions are effectively ignored. In checking mode, all variables in domain patterns have already been bound when an expression in a when or where clause is evaluated. For example, the conditional expression for checking in backward direction whether a CPM activity should be mapped onto a Gantt activity (lines 20–24 in Figure 6) always evaluates to true. However, for CPM activities to be mapped onto Gantt dependencies there is no corresponding Gantt activity. This results in an error even if the diagram and the network are mutually consistent.

4.5 Evaluation

In the following, we summarize the lessons learned from the case study. The presentation is structured according to the criteria introduced in Section 1.

Expressiveness. Construction of bidirectional transformations is inherently difficult due to *heterogeneity* of the underlying metamodels (Figure 1). Thus, there is a problem-inherent limitation concerning the functionality of bidirectional transformations. In our case study, Gantt diagrams and CPM networks are closely related, but the concepts of activity and dependency in Gantt diagrams cannot be mapped 1:1 to the concepts of activity and event in CPM networks. While the forward transformation is total, the backward transformation may reconstruct the original Gantt diagram, but can be applied only to CPM networks generated by the forward transformation. This problem-inherent limitation would also hold if we had used any other model transformation language.

An important added value of QVT-R consists in its language support for *relational specifications* of bidirectional transformations. In this respect, *expressiveness* of QVT-R is concerned with the following issue: Suppose that we constructed two unidirectional transformations that exhibit the desired behavior (as in our case study). Is it possible to synthesize a single, equivalent bidirectional transformation? The more often the answer to this question is positive, the more expressive is QVT-R compared to languages supporting only unidirectional transformations.

In our case study, the synthesis succeeded to a

large extent. The resulting bidirectional transformation works correctly in enforcing mode and exhibits the same behavior as the unidirectional transformations from which it was synthesized. Unfortunately, the transformation does not behave correctly in checking mode. This is due to the fact that our simulation of unidirectional expressions does not work in checking mode (see previous subsection).

Conciseness. Since QVT-R is a declarative rather than procedural language, specifications written in QVT-R are concise inasmuch as they just describe the patterns to be searched and instantiated, without specifying algorithmic details to be taken care of by a QVT-R execution engine.

Furthermore, as far as bidirectional transformations are concerned, conciseness results from the fact that both transformation directions can be expressed by a single relational specification. In other languages, separate transformations have to be defined for each direction.

Readability. Another relevant issue is *cognitive complexity*: How easy is it to understand a QVT-R specification of a transformation? In our case study, the *unidirectional transformations* are relatively easy to understand. The relations have a straightforward semantics: Search an instance of the source pattern, check the when clause, and instantiate the target pattern using the where clause. The when clause and the where clause have clearly separated responsibilities: They act as pre- and postconditions. The order in which relations are executed may be constrained by relation calls in the when clauses.

However, if we specify separate forward and backward transformations, we have to ensure manually that the opposite transformations are mutually consistent. The full power of QVT-R is exploited only when we construct a *bidirectional transformation* which is executable in both the enforcing mode (in both directions) and the checking mode. The term *relational specification* suggests that such a relation may be defined declaratively at a high level of abstraction.

Unfortunately, the bidirectional transformation constructed in the case study does not meet these expectations. The author of the transformation has to consider in detail the *execution semantics* of the specification. For each mode and each direction, it has to be verified that the relations work correctly and execution satisfies all data flow constraints. To make the transformation executable in both directions, it may be necessary to simulate unidirectional expressions. However, this simulation impedes readability and does not work in checking mode. Furthermore, the separation of responsibilities is blurred: Conditional expressions have to be moved to the when clause because they do not work in the where clause. Finally, execution of the when clause and the where clause may be intertwined, resulting in further confusion. Altogether, we conclude that the bidirectional transformation which we constructed in this paper is very hard to understand (although it was constructed systematically from two unidirectional transformations).

Semantic Soundness. To a large extent, cognitive complexity results from flaws in the semantics definition. In unidirectional transformations, source pattern, target pattern, when and where clause have clearly separated responsibilities. But for bidirectional transformations, the *responsibilities* are *blurred*. An expression acting as a precondition in the forward direction may play the role of a postcondition in the backward direction. Thus, it is by no means straightforward to decide whether an expression should go the when clause or the where clause.

The unclear separation of responsibilities even goes as far as allowing dependencies of expressions in the when clause on expressions in the where clause. This liberty results from the fact that a relation is essentially considered as a set of expressions which are evaluated in an order conforming to data flow constraints (before a variable may be accessed, it must be bound). Unfortunately, this declarative approach to defining the execution semantics does not guarantee confluence in general: If the data flow constraints permit different orders of evaluation, the results of execution may differ (*inadvertent non-determinism*).

4.6 Proposals for Revising QVT-R

Based on the experiences gained through the case study, we propose several revisions to QVT-R and illustrate these revisions with an example.

Model Qualifiers. In QVT-R, domain qualifiers determine in which way a domain of a relation may be used in a transformation. These qualifiers may differ from relation to relation. Since we did not encounter any use cases requiring this degree of freedom, we suggest to replace local domain qualifiers by global *model qualifiers* specifying the roles of models in transformations. The model qualifiers are inherited by all relations, i.e., the qualifier of a model consistently applies to the domains of all relations.

Unidirectional Expressions. Frequently, expressions specified in a relation are relevant in only one direction. The “solution” presented in this paper —

```
1  transformation bidirectionalRevised(enforce diagram : gantt, enforce network : cpm) {
2      ...
3      top relation Dependency2Activity {
4          cn_predActivity, cn_succActivity : cpm::Activity;
5          gd_dependencyType : gantt::DependencyType;
6          offset : Integer; name : String;
7          domain diagram gd_dependency : gantt::Dependency {
8              offset = offset,
9              diagram = gd_diagram : gantt::GanttDiagram {},
10             predecessor = gd_predActivity : gantt::Activity {},
11             successor = gd_succActivity : gantt::Activity {},
12             dependencyType = gd_dependencyType
13         };
14         domain network cn_activity : cpm::Activity {
15             duration = offset, name = name,
16             network = cn_network : cpm::CPMNetwork {},
17             sourceEvent = cn_sourceEvent : cpm::Event {},
18             targetEvent = cn_targetEvent : cpm::Event {}
19         };
20         when {
21             Diagram2Network(gd_diagram, cn_network);
22             to diagram {
23                 mapToDependency(cn_activity);
24                 cn_predActivity = owningActivity(cn_sourceEvent);
25                 cn_succActivity = owningActivity(cn_targetEvent);
26             }
27             Activity2Activity (gd_predActivity, cn_predActivity);
28             Activity2Activity (gd_succActivity, cn_succActivity);
29         }
30         where {
31             to network {
32                 cn_sourceEvent = sourceEvent(cn_predActivity, gd_dependencyType);
33                 cn_targetEvent = targetEvent(cn_succActivity, gd_dependencyType);
34                 name = gd_predActivity.name + '->' + gd_succActivity.name;
35             }
36             to diagram {
37                 gd_dependencyType = dependencyType(cn_activity);
38             }
39         }
40     }
41 }
```

Figure 9: Bidirectional relation for dependencies in revised notation.

conditional expressions querying the binding state — may be considered only a workaround which significantly increases cognitive complexity and does not work in the checking mode. Therefore, we propose to support *unidirectional expressions* directly in the QVT-R language. To this end, we propose to place an optional *direction specification* before an expression list occurring in a relation. The direction specification is indicated by the keyword to followed by a model name. If the direction specification is present, the expression list is considered only in the direction of the specified model.

Structured Execution of Relations. To eliminate inadvertent non-determinism and to ensure separation of responsibilities, we propose to structure the execution of relations into clearly delineated *phases*: (1) Match source domain. (2) Evaluate when clause. (3) Evaluate where clause. (4) Instantiate target domain. Each phase may rely only on previous phases. Since expressions may have to be evaluated after instantiation of the target domain, we propose to add an optional *post clause* which is executed in phase (5) (which, however, is not needed in our case study).

Example. Figure 9 presents the revised bidirectional relation for transforming dependencies into activities. The domain qualifiers were replaced with model qualifiers. The relation may be synthesized in a straightforward way from the forward and the backward relation. The when and the where clause are merged separately; unidirectional expressions are

marked with a direction specification. The resulting relation is slightly more concise and considerably easier to understand than the standard-compliant bidirectional relation in Figure 7. Furthermore, it works in checking mode, as well.

5 RELATED WORK

The vast majority of transformation languages supports only unidirectional transformations; consider e.g. the well-known ATL language (Jouault et al., 2008). However, bidirectionality is not a unique feature of QVT-R (Czarnecki et al., 2009). The most prominent competitors are languages and tools based on *triple graph grammars* (Schürr, 1995; Königs and Schürr, 2006; Schürr and Klar, 2008; Kindler and Wagner, 2007). In contrast to the grammar-based approach, QVT-R follows a relational paradigm where a transformation is specified by a set of relations among patterns to be instantiated in the participating models.

To the best of our knowledge, our work is unique inasmuch as it addresses the evaluation of QVT-R with respect to bidirectional transformations. Other work focuses primarily on the formal definition of QVT-R's semantics (Stevens, 2010; Stevens, 2013; Guerra and de Lara, 2012; de Lara and Guerra, 2009). While support of bidirectional transformations constitutes the most interesting (and challenging feature) of QVT-R, this issue has been neglected severely from the application point of view. The standard itself provides only examples of unidirectional transformations. We are aware of only a single application-oriented paper dealing with bidirectional transformations in QVT-R (Schwichtenberg et al., 2014), which uses, but does not evaluate QVT-R's features for specifying bidirectional transformations.

6 CONCLUSION

Bidirectional transformations are required in a number of applications of industrial relevance, including bidirectional data converters and round-trip engineering. In this paper, we presented a case study for evaluating QVT-R, a declarative language which was defined in an OMG standard. QVT-R is distinguished from other model transformation languages by providing the concept of a relational specification of a transformation which may be executed in different modes and different directions. For bidirectional transformations, this approach reduces the effort of writing specifications and contributes to providing mutually consistent forward and backward transformations. However, the case study revealed severe deficiencies of QVT-R: The specification of a bidirectional transformation between Gantt diagrams and CPM networks works only in enforcing mode, it is very hard to understand since the QVT-R user has to consider the details of the execution semantics carefully, and the execution semantics is defined in an unclear way. Therefore, we proposed several revisions of QVT-R to remove these deficiencies.

The work presented in this paper is embedded into a more comprehensive investigation into bidirectional transformations in QVT-R. In (Westfechtel, 2015), seven case studies are explored, ranging from simple cases (e.g., migration between unweighted and weighted Petri nets) to complex cases such as an object-relational mapping. For each of the cases, a QVT-R solution is provided. Furthermore, the transformations are classified with respect to their formal properties, and guidelines are given for applying the QVT-R language as it stands. The problems occurring the project management case study presented here were observed also in other case studies, and most of the proposed language revisions may already be deduced from the project management case.

Currently, we are using QVT-R to implement a tool for round-trip engineering between UML class diagrams and Java code. The tool is realized as part of Valkyrie, a model-driven environment for model-driven software engineering (Buchmann, 2012). Valkyrie is based on the EMF version of the UML 2 metamodel; the Java model is provided by MoDisco (Brunelière et al., 2014). The round-trip engineering tool will operate incrementally, propagating changes forth and back. Thus, the tool provide us with a non-trivial case study dealing with incremental transformations, which go beyond the scope of the current paper. Furthermore, we will compare the QVT-R-based tool against an earlier implementation based on triple graph grammars (Buchmann and Westfechtel, 2013). In addition, we will also solve the transformation problems presented in (Westfechtel, 2015) with triple graph grammars. In this way, we will obtain a set of case studies for a comparative evaluation of QVT-R and triple graph grammars.

ACKNOWLEDGEMENTS

The specifications presented in this paper were developed and tested with medini QVT, version 1.7.0 under Eclipse Indigo.

REFERENCES

Brunelière, H., Cabot, J., Dupé, G., and Madiot, F. (2014). MoDisco: A model-driven reverse engineering framework. *Information and Software Technology*, 56(8):1012–1032.

Buchmann, T. (2012). Valkyrie: A UML-based model-driven environment for model-driven software engineering. In Hammoudi, S., van Sinderen, M., and Cordeiro, J., editors, *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*, pages 147–157. ScitePress.

Buchmann, T. and Westfechtel, B. (2013). Towards incremental round-trip engineering using model transformations. In Demirors, O. and Turetken, O., editors, *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2013)*, pages 130–133. IEEE Conference Publishing Service.

Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F. (2009). Bidirectional transformations: A cross-discipline perspective. In Paige, R. F., editor, *Proceedings of the Second International Conference on Theory and Practice of Model Transformations (ICMT 2009)*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283, Zurich, Switzerland. Springer-Verlag.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

de Lara, J. and Guerra, E. (2009). Formal support for QVT-Relations with coloured petri nets. In Schürr, A. and Selic, B., editors, *Proceedings of the 12th International Conference on Model Driven Engineering and Systems (MODELS 2009)*, volume 5795 of *Lecture Notes in Computer Science*, pages 256–270, Denver, CO. Springer-Verlag.

Guerra, E. and de Lara, J. (2012). An algebraic semantics for QVT-Relations check-only transformations. *Fundamentae Informaticae*, 114(1):73–101.

Jakumeit, E., Buchwald, S., Wagelaar, D., Dan, L., Hegedüs, A., Herrmannsdörfer, M., Horn, T., Kalnina, E., Krause, C., Lano, K., Lepper, M., Rensink, A., Rose, L., Wätzold, S., and Mazanek, S. (2014). A survey and comparison of transformation tools based on the transformation tool contest. *Science of Computer Programming*, 85A:41–99.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Science of Computer Programming*, 72(1–2):31–39.

Jungnickel, D. (2008). *Graphs, Networks and Algorithms*, volume 5 of *Algorithms and Computation in Mathematics*. Springer, Berlin, Germany, 3rd edition.

Kerzner, H. (1998). *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*. John Wiley & Sons, New York, NY, 6th edition.

Kindler, E. and Wagner, R. (2007). Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, University of Paderborn, Paderborn, Germany.

Königs, A. and Schürr, A. (2006). Tool integration with triple graph grammars - a survey. In Heckel, R., editor, *Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004)*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 113–150, Dagstuhl, Germany. Elsevier Science.

Kühne, T. (2006). Matters of (meta-)modeling. *Software and Systems Modeling*, 5(4):369–385.

Object Management Group (2012). *Object Constraint Language Version 2.3.1*. Needham, MA, formal/2012-01-01 edition.

Object Management Group (2013). *OMG Meta Object Facility (MOF) Core Specification Version 2.4.1*. Needham, MA, formal/2013-06-01 edition.

Object Management Group (2014). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.2 Beta*. Needham, MA, ptc-2014-03-38 edition.

Reddy, S., Venkatesh, R., and Zahid, A. (2006). A relational approach to model transformation using QVT Relations. Technical report, Tata Research Development and Design Centre, Pune, India.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31.

Schürr, A. (1995). Specification of graph translators with triple graph grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994)*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163, Herrsching, Germany. Springer-Verlag.

Schürr, A. and Klar, F. (2008). 15 years of triple graph grammars – research challenges, new contributions, open problems. In Ehrig, H., Heckel, R., Rozenberg, G., and Taentzer, G., editors, *Graph Transformations: 4th International Conference (ICGT 2008)*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425, Leicester, UK. Springer-Verlag.

Schwichtenberg, S., Gerth, C., Huma, Z., and Engels, G. (2014). Normalizing heterogeneous service description models with generated QVT transformations. In Cabot, J. and Rubin, J., editors, *Proceedings of the 10th European Conference on Modelling Foundations and Applications (ECMFA 2014)*, volume 8569 of *Lecture Notes in Computer Science*, pages 180–195, York, UK. Springer-Verlag.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2nd edition.

Stevens, P. (2010). Bidirectional model transformations in QVT: Semantic issues and open questions. *Software and Systems Modeling*, 9(1):7–20.

Stevens, P. (2013). A simple game-theoretic approach to checkonly QVT Relations. *Software and Systems Modeling*, 12(1):175–199.

Westfechtel, B. (2015). Bidirectional transformations in QVT Relations: Potentials and limitations. *Journal of Object Technology*. Submitted for publication.