QVTR^2: a Rational and Performance-aware Extension to the Relations Language*

Mauro Luigi Drago, Carlo Ghezzi, and Raffaela Mirandola

Politecnico di Milano
DeepSE Group - Dipartimento di Elettronica e Informazione
Piazza Leonardo Da Vinci, 32 - 20133 Milano, Italy
(drago|ghezzi|mirandola)@elet.polimi.it
WWW home page: http://deepse.dei.polimi.it

Abstract. Model transformations glue together models in an MDE process and represent the rationale behind it. It is however likely that in a design/development process different solutions (or alternatives) for the same problem are available. When alternatives are encountered, engineers need to make a choice by relying on past experience and on quality metrics. Several languages exist to specify transformations, but all of them bury deep inside source code rational information about performance and alternatives, and none of them is capable of providing feedback to select between the different solutions. In this paper we present QVT-Relations Rational (QVTR^2), an extension to the Relations language to help engineers in keeping information about the design rationale in declarative transformations, and to guide them in the alternatives selection process by using performance engineering techniques to evaluate candidate solutions. We demonstrate the effectiveness of our approach by using our QVTR^2 prototype engine on a modified version of the common UML-to-RDBMS example transformation, and by guiding the engineer in the selection of the most reasonable and performing solution.

1 Introduction

Model Driven Engineering (MDE) envisions a shift in the software development process by posing the attention on models instead of source code. Models become first class citizens and the theory and practice of software development should be ported to this new paradigm. The MDE paradigm should simplify the tasks of the engineer; however, developing a complete application still remains complex. Several models at different abstraction levels are required to describe all the facets of a system, and consistency between them should be maintained.

The MDE paradigm and the availability of system abstractions, also in the early stages of development, support different new kinds of software engineering tasks. For example, models can be used to predict the non-functional attributes of software artifacts and prevent performance issues which would otherwise be

* This research was partially founded by the European Commission, IDEAS-ERC Project 227977-SMScom and EU FP7 Q-ImPrESS project.

discoverable only after the system has been implemented, when the application is running. Balsamo et al. in [1] give an overview of the many approaches proposed in literature to tackle performance analyses. More recent work can also be found in the proceedings of WOSP [2]. Almost all the approaches translate high-level models annotated with appropriate information about non-functional aspects into performance related formalisms, like Queueing Networks (QNs) or Markov Chains (MCs). Analyses are then performed on these low-level models by using already existing tools. However, it is still entirely up to the engineer to understand what numbers mean and what changes to apply. Existing approaches provide poor or no support at all to interpret results, to relate them from low-level abstraction layers to high-level layers, and to identify appropriate solutions when requirements are not met. Methodologies for feedback provision to engineers are already available in literature; example are the meta-heuristic approaches described in [3][4][5] or the rule based approaches described in [6][7][8]. Indeed, meta-heuristic approaches work only in particular contexts (e.g., component based engineering) and provide solutions only for specific performance issues (e.g., finding an optimal allocation for components). Rule-based approaches are more generic and can be potentially applied to a wide variety of performance engineering tasks; our research concentrates on them.

Rules to identify problems and propose alternatives are specific to each application domain, and must still be manually written by domain experts. However, in an MDE development setting, experts are also responsible for capturing the engineering domain and embed into model transformations, as much as possible, the rationale behind the development process. Several languages exist to specify transformations [9], which may be classified along different dimensions. A first distinction can be made by considering the specification style, i.e., imperative or declarative. Imperative languages operationally specify how elements should be transformed and are more familiar to end-users given their familiarity with the paradigm of common programming languages. Examples of purely imperative languages may be QVT-Operational [10] or Kermeta [11]. Declarative languages instead specify how model elements relate to each other and what should be transformed into what. Some languages exhibit a purely declarative style, such as QVT-Relations [10], Triple Graph Grammars (TGG) [12] or Tefkat [13]. Others instead adopt an hybrid style, such as ATLAS [14], which combines declarative and imperative constructs. Another distinction can be made by considering bi-directional capabilities. Languages which provide imperative constructs are usually uni-directional, i.e., a transformation specification may be executed only to transform a model A into a model B but not in the opposite direction. Bi-directional transformations instead may be executed both forward and backwards. We argue that bi-directional transformations are advantageous when several models are used, consistency must be enforced, and performance indexes must be related to high-level abstractions. Indeed, in this paper we concentrate on them and, specifically, on the QVT-Relations language.

Our intuition is that the rationale behind rules to provide feedback to engineers is already available in transformations. This has been already pointed

out by Hettel et al. in [15]: *"transformations are also definitions of the engineering process"*. For example, let us consider the recurring problem of mapping components onto the hardware resources. Each component may be mapped onto multiple replicas deployed on different resources. The choice of replicating (or not) and the number of replicas constitute different alternative solutions for the same problem. The engineer, guided by past experience and by the performance requirements, makes a choice and solves the variability. Transformations, if extended with suitable constructs to elicit that rationale, can be used both to automate the development process and to close the feedback loop.

However, none of the currently available transformation languages provides *i*) appropriate constructs to express (and keep track of) performance decisions, but buries them deep inside the source code, and *ii*) mechanisms to assist engineers in the evaluation/selection of the different alternatives. In this paper we present QVT-Relations Rational (QVTR^2), an extension of the QVT-Relations language to tackle the afore-mentioned issues. The contribution of this paper is two-fold:

- We add new constructs to QVT-Relation, in the form of annotations similar to the ones already available for the Java language, to elicit alternatives and performance concepts inside transformation specifications.
- We provide a prototype engine to execute QVTR^2 transformations which partially automates — human intervention is still required by our engine — the feedback loop. Whenever an alternative is identified, the engine evaluates (by using standard performance engineering techniques) the possible solutions, and shows the gathered results to the end-user, who in turn can be guided in the selection of the most reasonable candidate.

The rest of this paper is organized as follows. In Section 2 we introduce some basic concepts and the running example we use in the rest of the paper. In Section 3 we describe the extensions we made to the language. Section 4 shows how we aid the developer, while transforming, when a choice needs to be made. In Section 5 we show QVTR^2 in action, by running the reference example. Sections 6 and 7 describe related work and new research ideas, respectively.

2 Preliminaries

In this section we describe some basic concepts needed in the rest of the paper and we introduce the running example we use to describe QVTR^2.

2.1 Background Terminology

We tackle the issue of providing feedback to engineer when multiple alternatives are available for the same artifact but with different non-functional attributes. Indeed, the concepts of alternative, when/where they occur, and of deciding between them have been already addressed in literature about Software Product Line (SPL) [16]. In this section we introduce the concepts we borrow from this research area in order to provide a widely accepted formal terminology for the

rest of this paper. In terms of SPLs, what we call alternatives correspond to *variant*s, while the point at which they occur is encapsulated by *variation point*s. Many definitions have been given in the past for these terms. For variation points, we stick to the definition given by Jacobson in [17]: *"A variation point identifies one or more locations at which the variation will occur"*. Indeed, a variation point defines *what* varies or the context in which the variability happens. A variant instead represents *how* software artifacts vary; indeed, it specifies which are the possible available alternatives. For example, let us consider the problem of allocating a component onto one or more hardware resources. The allocation problem is the *what* (or the variation point), the different possible allocations are the *how* (or the variants).

2.2 QVT-Relations

QVT-Relations is a declarative transformation language with bidirectional capabilities. A transformation between two or more candidate models is a set of relations which specify the constraints that must hold between model entities. Each relation has two or more domains and may contain **when** and **where** clauses. An example is shown in Listing. 1.1. Domains define the object patterns used to match/create entities in the models. For example, the *uml* domain (line 10) matches entities of type *Class* while the *rdbms* domain (line 14) matches *Partitions* containing at least a *Table* with the same *name* of the *Class*. The **when** clause defines the pre-conditions for the relation; the **where** clause instead specifies the predicates that must hold when this relation holds (post-conditions). Predicates define constraints either on entities matched by the domain patterns or on other relations. The latter case is used to define precedence between relations, if the predicate belongs to the **when** clause, or a simple call-out semantics if the predicate belongs to the **where** clause. In the example, the *AllClassAttributesToSingleTable* relation (line 33) is invoked in the **where** clause to force the mapping of all the attributes of the matched class onto columns of the matched table. Transformation can be executed in *checkonly* or *enforce mode*. In *checkonly mode*, the input models are not modified but only consistency is verified, by checking that all the top relations hold. In *enforce mode*, a direction domain must be specified and the target model is modified in order to respect the transformation relations. If an entity compliant with the enforce domain pattern is not matched in the target model, a new one is created as defined by the pattern specification. To avoid polluting models with duplicate objects, it is possible to specify keys for model entities. In the example, we specified that *Table*s are identified by their *name*, their *partitionId* and by the *schema* to which they belong.

2.3 The Running Example

We based our running example on the *Simple UML Class* model to *RDBMS* model mapping described in the QVT specification [10], which we reduced and tailored to our needs. Here we describe it briefly by concentrating on the modified

parts; for further information refer to the QVTR[2] website[1]. The example tackles the Object Relational Mapping of a domain entity model to an underlying database. The *Simple UML Class* meta-model has been taken as is; it allows for the definition of entities, attributes of entities, and associations between them. The *RDBMS* meta-model (whose relevant part is shown in Figure 1) has been instead extended to allow the definition of data partitions and generation policies for record identifiers. Tables may either live on their own inside a schema, or they can belong to a partition. If this is the case, each table is identified in the transformation by its name and by the *partitionId* attribute. Table columns are described by an extra *generationPolicy* attribute, which defines the strategy to generate identifiers for new data entries.

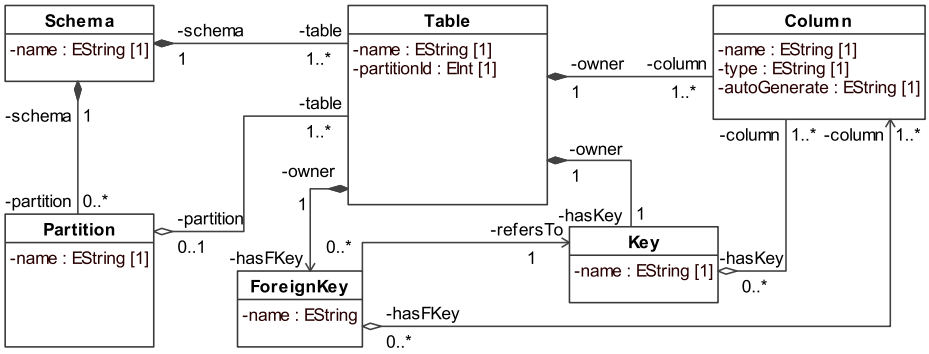


Fig. 1: The *RDBMS* meta-model.

To add some performance-related alternatives in the example, we took inspiration from performance anti-patterns [18]. Anti-patterns state what a designer should not do in order to avoid potential problems. Performance anti-patterns concentrate on performance issues, and have been widely studied in the literature. Smith et al. in [18] list the most common anti-patterns found in industrial applications. For the example, let us focus on the *One-Lane Bridge* anti-pattern. It occurs when "*at a point in the execution only one, or a few, processes may continue to execute concurrently*". For databases, this happens when multiple clients concurrently access and insert/modify records in the same table. Different solutions have been proposed. Changing the policy to generate record identifiers may mitigate the problem. When default sequential algorithms are used, sequentially inserted records share the same storage location (or memory page). When inserting a new record, the database engine locks the entire page used to store the new entry, and other clients must wait for such lock to be released. If identifiers are generated *randomly*, clients are diverted to different storage locations and may proceed concurrently. Another, more general, possibility consists in partitioning data onto multiple tables. Each client can be then redirected onto a different table, and mutual exclusion occurs less frequently.

[1] http://code.google.com/p/qvtr2/

We extended the example transformation by adding the mappings for these possible solutions. In particular, we defined three kinds of variation points. One implements the mapping of table keys onto *sequential* or *randomly-generated* identifiers. The others tackle data partitioning. A variation point handles mapping of classes onto single tables or partitions. If partitioning is chosen, the last variation point decides how many tables should belong to the partition.

3 The Language

The Relations language already provides constructs to implement alternatives in transformations. However, current implementations lack runtime support (i.e., existing engines make a non-deterministic choice between viable solutions) and features to include rational information. In this section we concentrate on the latter issue. Runtime support is discussed subsequently. We propose to use Java-like annotations for transformation predicates to include such information. Using annotations instead of modifying the grammar with new constructs brings to an important consequence: QVTR[2] transformation are also valid QVT-Relations specifications, hence we can use existing engines to interpret and execute transformations.

When building a transformation in which multiple alternative output models are viable, we express variants through relations, while we let variation points correspond to boolean predicates, in the post-condition of a relation, over the possible alternatives. To be sound, such definitions must satisfy some conceptual and syntactic criteria. Alternatives are defined by relations, and each relation is composed of several domains. When executing a transformation, one of the domains is the target, i.e., it is used to create from scratch a new output model or to check its consistency. The remaining domains are instead used to match elements in input models, and they specify the subject of the variability. Conceptually, the subject of a variability must be unique; hence we require the patterns defined for such domains to be equal. More formally, we require that *given a variation point definition and a direction, the remaining domains of all its variant relations must be equal* (**Rule 1**).

We also distinguish between mutually exclusive, optional and recursive variation points. According to this classification, the predicates defining a variation point must comply with the following conditions. Before proceeding, we need to recall that boolean operators in the Relations language are short-circuited, and our definitions work under this assumption. *Mutually exclusive variation points are expressed through predicates* xor*-ing the possible alternatives, in the other case the* or *operator must be used* (**Rule 2**). *Optional variation points are non-mutually exclusive variation points,* or*-ing a unique variant with the* true *boolean value* (**Rule 3**). *Recursive variation points are optional variation points, and the alternative must be the same relation in which the variation point resides* (**Rule 4**). Recursive variation points are useful in situations when an entity may be mapped several times onto the same type of objects. An example is shown in Listing 1.1 (lines 43-44): a class may be mapped onto multiple tables of the same data partition, each one with a different partition identifier.

Listing 1.1: The *NumberOfTables* variation point.

```
@variant {
  name := "AnotherTable",
  description := "Adds another table to partition",
  excludes := {IdGenStrategy(RandomPolicy)}
}
relation ClassToAnotherTableInPartition {
  className, pkeyName : String;
  intId : Integer;

  checkonly domain uml c:Class {
    name = className
  };

  enforce domain rdbms p:Partition {
    table = t:Table {
      name = className,
      schema = p.schema,
      partitionId = intId,
      hasKey=k:Key {
        name = pkeyName
      }
    }
  } default_values {
    pkeyName = className + '_pk';
  };

  primitive domain curId : Integer;

  when { intId = curId; }

  where {
    IdAttributesToPkeyCols(c,k);
    AllClassAttributesToSingleTable(c,t);

    ClassToTableInPartition(c,t);
    t.partition->size() = 1;

    @varpoint {
      name := "NumberOfTables",
      description := "Decides the number of tables",
      analyzer(full) := QnAnalyzer($vp, $choice, "usage.xmi")
    }
    ClassToAnotherTableInPartition(c, t.partition, curId + 1)
      or true;
  }
}
```

3.1 The Annotations

QVTR[2] provides two types of annotations for variabilities. The *varpoint* annotation can be used for relational predicates implementing variation points, and supports the following properties:

Name and Description properties serve as documentation aids. *Name* specifies the unique identifier of the variation point. It is used for cross-references both in annotations and in design documents. *Description* provides a quick synoptic of the variability, and helps engineers in understanding what transformations do.

Analyzer specifies how to evaluate the candidate variants, i.e., how to compute information to guide engineers while selecting alternatives. Analyzers are domain dependent: each application domain and design process may use specific meta-models, transformations, and analysis techniques to extract feedback information. We support their definition and implementation through Java classes, whose fully qualified name constitutes the value of this property. Parameters can be passed to the analyzer. Both user-specific parameters and special parameters can be specified with the standard method invocation notation. Examples of special parameters are $vp and $choice, respectively pointing to objects containing information about the variation point and about the variant being evaluated. A mode (i.e., *full* or *lazy*) can also be specified to regulate how the analysis is carried out and to speed-up execution when computations may take long. The *full* mode instructs analyzers to ignore timing issues and to consider the whole candidate output model. *Lazy* execution mode instead forces consideration only of the *locally-changed* elements, elements created or modified by the candidate alternative. *Lazy* execution is usually faster, but may provide only partial and imprecise information.

The *variant* annotation can be instead used to annotate relations, and supports the following properties:

Name and Description properties absolve to documentation needs.

Excluding and Including specify dependency relations between alternatives. It is a list of variation points and variants, and serves the purpose of expressing cross-reference constraints on variabilities. In detail, *excluding* specifies which solutions for other variation points should be a priori excluded when the annotated alternative is selected. Exclusion dependencies between variants are symmetric by definition, so engineers need to define them only on one of the endpoints. *Including* defines which solutions must be picked up when the annotated alternative is selected, and is not symmetric.

Listing 1.1 shows how annotations can be used to define alternatives. In detail, we show the recursive variation point to decide the number of replicas inside data partitions. Candidate variants will be evaluated by a *full* mode analyzer, which requires an extra parameter (i.e., *"usage.xmi"*) to specify the location of

a usage profile model. The candidate database model in conjunction with information about the workload to sustain is used to generate a Queueing Network, useful to predict the performance of the solution. More details about generation of QNs will be given in Section 5. An exclusion dependency is also specified in the variant annotation. Data contained in tables is usually partitioned according to row identifiers, which are automatically managed by the underlying database engine. Specifying a random generation policy for identifiers would then conflict with partitioning, hence we mutually excluded the two solutions.

4 Runtime Support

Although a QVTR^2 transformation is also a syntactically valid QVT-Relations transformation, it cannot be executed as-is. Alternatives introduce non-determinism — for the same input multiple outputs are possible — and this leads to conceptual and practical problematic consequences. QVT-Relations does not tackle properly non-determinism. Its specification clearly states that, when multiple choices are available, then they should be equivalent, which is not true from a non-functional point of view. The practical consequence is that any viable alternative can be taken while transforming. Actually, existing engines make a deterministic selection of the first possibility, as boolean operators are short-circuited. The QVTR^2 prototype we built allows the transformation engine to discriminate among variants by evaluating them automatically and providing feedback to software engineers at runtime. The prototype modifies the transformation cycle and leverages standard language constructs to intercept when variation points occur. The advantage of this solution is that we can use existing transformation engines without modifying them. The Relations language is relatively new and few (partial) implementations are available. To the best of our knowledge, only two working transformation engines are available: Medini-QVT [19] and ModelMorf [20]. We tried both and we selected ModelMorf for its reliability and number of language features implemented. In the following we give further details about how QVTR^2 transformations may be executed. First the execution cycle is presented, then we deepen inside the most critical steps.

4.1 The QVTR^2 Cycle

The abstract execution cycle of a QVTR^2 transformation is depicted in Figure 2. Dashed boxes represent steps in the cycle, solid arrows stand for control flows and dashed lines connect steps with produced artifacts. Annotated transformations undergo two analysis stages before real execution. The *Alternatives Analysis* parses relations, builds an abstract representation of them and searches for annotations. The outcome is a model including information about found variation points, their properties, and the predicates used in the source code to express them. Information to uniquely identify elements in source models (which we need in the following steps) is instead extracted by the homonymous analysis, by searching for key definitions in the transformation source in conjunction with

input models. Once these pieces of information are extracted, real execution can proceed by intercepting occurrence of variation points.

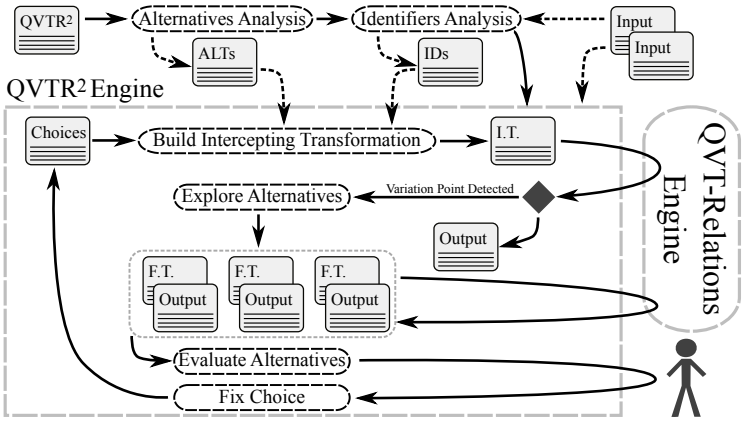


Fig. 2: The QVTR^2 Execution Cycle.

Intercepting Variation Points. Given a variation point definition and the predicate used to express it, we say that it occurs when we reach the predicate during the execution of the transformation, and the engineer has not previously selected one of the viable alternatives. To intercept these events we create intermediate transformations — which we call Intercepting Transformation (IT) — leveraging the possibility to provide black-box implementations for relations. The original QVTR^2 transformation is modified by adding, for each variation point, a new black-box relation compatible with the domains of the variant relations. Let's consider the example shown in Listing 1.1. The only viable alternative accepts three domains for the class, the partition and the current identifier. The new relation we create is shown in Listing 1.2. The original predicates used to express variation points are also modified, in order to point to the new relation.

Listing 1.2: The black-box interceptor for the AnotherReplica variation point.

```
relation NumberOfTables_interceptor {
  checkonly domain uml c : umlMM::Class {};
  enforce domain rdbms p : rdbmsMM::Partition {}
    implementedby bbox_NumberOfTables(c,curId,p);
  primitive domain curId : Integer;
}
```

Note that we must only intercept the occurrence of variation points for which an alternative has not been selected. To discern between the two cases, we use nested if constructs invoking selected alternatives for fixed variabilities, or pointing to interceptors. Let's consider our reference variability and let's suppose that the engineer has already decided to map *MyClass* onto at least three tables. Listing 1.3 shows the substitutio predicate. We intercept the variability only if conditions on line 1 and 2 evaluate to false. *PartitionId*s start from 0 and two

tables are created by default when the partitioning strategy is selected; hence, the conditions evaluate to false only when we are trying to create the fourth table in the partition.

Listing 1.3: The substitution predicate.

```
1 if (curId + 1 = 2 and c.namespace.name = 'MyNamespace'
2     and c.name = 'MyClass') then
3   ClassToAnotherTableInPartition(c, t.partition, curId + 1)
4 else
5   NumberOfTables_interceptor(c, t.partition, curId + 1)
6 endif
```

How black-box implementations are handled is still a matter specific to each transformation engine. ModelMorf handles them by using user-supplied Java implementations, which we automatically generate. When an Intercepting Transformation is executed two things may happen. Either no variation point occurred, the transformation ends regularly, and the model produced corresponds to the final output; or a variation point occurs and we start exploring the available alternatives.

Exploring, Evaluating and Choosing. When a variation point occurs, we explore it by generating the models corresponding to each possible variant. First we compute an intermediate transformation for each variant — which we call Fixing Transformation (FT) —, then we execute them to generate candidate models. Fixing transformations are modified versions of the original QVTR^2 transformation. As we did for ITs, variation point expressions are substituted with predicates similar to the one shown in Listing 1.3. When alternatives have been already selected, we fix them in the predicate. If this is not the case, the variant picked up in the else branch (line 5) depends on whether we are substituting the predicate for the variation point being explored or not. If this is the case, the FT fixes the associated variant; otherwise we automatically select one of the possible solutions. Alternatives are consistently selected across FTs for non-fixed variation points: for each variation point, the same alternative is fixed in all FTs. Hence, generated models differ only for the variation point being explored and evaluation results are consistent. This corresponds to a local search strategy, which may not be practical in every case, but has the advantage of giving results fast — complete exploration of the state space is not required —. Once models are generated, we evaluate them by invoking the analyzers as specified in annotations. The output of the analyses may be of any kind. For example, it may be textual information, a chart, or a spreadsheet. We do not interpret it automatically; this task is still left to the engineer. When the most appropriate solution has been selected by the designer, we keep track of the choice by saving it into the Choices model for future use. The Choices model is used during the next iteration over the QVTR^2 cycle, to incrementally build the new fixing and intercepting transformations, and is useful to keep track of decisions, and change them, between multiple executions.

5 Enacting the Example

In this section we demonstrate QVTR[2] in action by using our reference example[2]. For space and comprehension reasons, we use a naive *Simple UML Class* instance model as input for the transformation. The input Class model we consider blueprints a simple cart application: *Person*s are associated to *Cart*s, and *Cart*s may contain *Item*s. To evaluate alternatives (i.e., candidate *RDBMS* models) we use a *full* mode analyzer that automatically generates from database models a Queueing Network, which we then solve by using the JMT tool [21]. The analyzer also requires usage profile information for the QN, shown in Table 1. For each class of the input model we define two QN customer classes (and their relative parameters), one for read operations and one for insert/update operations. Database tables are represented by service centers; if a class is mapped onto multiple tables then multiple service centers will exists in the QN serving the same type of requests. Service times for jobs are fixed to $0.05s$ for read operations and to $0.1s$ for write operations. If a table uses the random generation policy for identifiers, the analyzer discounts the write service time by a 0.2 factor. Required response times — to discern between candidate solutions — are also shown in the table, although they are not required to generate the QN. The analyzer computes global response times for each customer class, and reports back the estimations to the designer. We are aware that this methodology to estimate the performance of databases is simple and does not take into account important facets (e.g. query involving multiple tables). Indeed, more precise and complex estimation techniques [1] may be adopted instead, and we recall that, in this paper, our target is not the prediction techniques per se, but showing how model transformations and prediction techniques can be mixed to improve engineering tasks.

Table 1: The usage profile.

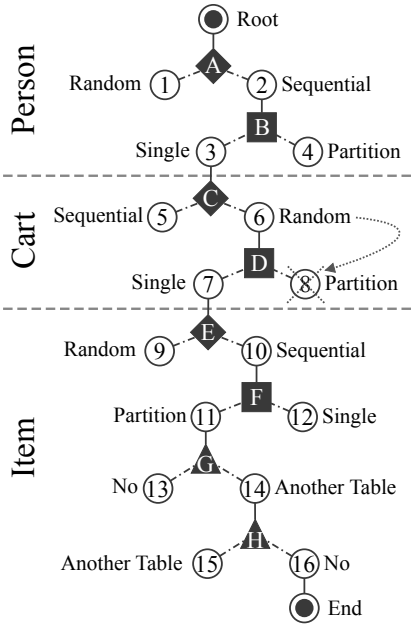
Class	Operation	Wkld. Type	Arrival Rate [s]	Service Time [s]	Required Response Time [s]
Person	Read	Open	1	0.05	0.2
	Write	Open	10	0.1	0.2
Cart	Read	Open	0.5	0.05	0.1
	Write	Open	1	0.1	0.1
Item	Read	Open	0.1	0.05	0.15
	Write	Open	0.1	0.1	0.15

Running the transformation on the simple cart model results in many passes through the QVTR[2] cycle. An overview of what happens at runtime is shown in Figure 3a. Solid boxes correspond to the occurrence of a variation point, and lead to generation/evaluation/selection of candidate models (the round boxes). The performance indices computed during model evaluation are shown in Figure 3c. These values, in conjunction with requirements, are used to select one of the viable alternatives. For example, the solid diamond C corresponds to the

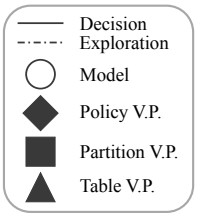
[2] Further details, models and source code are available on the QVTR[2] web site.

occurrence of the *identifiers generation policy* variation point for class *Cart*, and leads to the generation of the candidate models 5 and 6.

The corresponding lines in the table show the estimated response times for the *Cart* table, for the two solutions respectively. The *sequential* solution is not compatible with the requirements; hence we select the random generation policy. Note that models 7 and 8 are not included in the table. The random generation policy selected for the Cart entity conflicts with the partitioning solution. Only one alternative (model 7) is viable, hence, no evaluation is required. The final solution we obtain is model 16: a random policy has been selected to generate identifiers for *Cart* entities, and 3 tables are used for the *Item* partition. Identifying this solution — which respects requirements and is minimal — would have required, without the provided automation, manual exploration and evaluation.



(a) Decision Tree



(b) Decision Tree Legend

Entity	Model	Response Time [s] {read,write}
Person	1	{0.054, 0.086}
	2,3	{0.053, 0.101}
	4	{0.048, 0.102}
Cart	5	{0.067, 0.118}
	6	{0.061, 0.095}
Item	9	{∞, ∞}
	10,12	{∞, ∞}
	11,13	{0.152, 0.202}
	14,16	{0.064, 0.109}
	15	{0.053, 0.1065}

(c) Evaluation Information

Fig. 3: Running the transformation.

6 Related Work

Concerning rule-based approaches, Xu describes in [8] a feedback provision system for the PUMA framework, in which rules are derived from performance anti-patterns [18] and implemented with the Jess language. However, solutions are proposed only for trivial performance issues and, if this is the case, changes apply to the lowest abstraction layer without propagation along the modeling stack. Parsons in [7] describes a similar approach tailored to enterprise Java application. More recent work is also available in [6], where a methodology is

proposed to rank performance anti-patterns and identify the most reasonable cause of a performance problem after a design of the system has been created. We instead anticipate this task during system design and, given our generative approach, all performance problems have an implicit associated solution.

In the context of SPLs for performance engineering, it is worth citing the approach described by Tawhid et al. in [22]. They use UML and the MARTE profile to design SPLs with parametric performance information. The SPL specification is transformed into a model of the final product enriched with performance annotations. Layered-QNs are generated and analyzed; performance indices are then derived to guide engineers. Their approach is tailored to UML+MARTE models and to performance analysis with Layered-QN. We are instead more generic by giving to engineers the freedom of choosing the preferred analysis approach. Indeed, we have to admit that we lack such an holistic support for parametric performance properties. In the area of transformation languages, Petter et al. in [23] propose to extend QVT-Relations with constructs to express constraint satisfaction problems, which may be used to model some types of variabilities. Indeed, not all variabilities can be expressed, and they lack appropriate constructs to embed rational information. In [24] and in [25], high-order transformations are used to support variability. Also our approach uses high-order transformations to generate intercepting and fixing transformation; however, [24] and [25] lack support for performance engineering, and rely completely on the engineer's expertise to select and evaluate alternatives, which we instead support.

7 Conclusions

In this paper we presented an extension to the QVT-Relations language to represent rational information about alternative designs, and to provide performance feedback to engineers while transforming. Non-obtrusive annotations have been used to embed directly into transformations information about variation points, variants, dependencies between solutions, and how to gather non-functional information about them. We also described our prototype implementation of QVTR^2, based on an existing QVT-Relations transformation engine. A sample transformation mapping *UML Class* models to *RDBMS* has been run, and we showed how the possible solutions may be evaluated using performance engineering techniques, and how the engineer is involved in the selection of the most reasonable solution. For future work, we plan to extend our annotations in order to represent explicitly performance indices and parametric performance properties. We also plan to add automatic exploration of the decision space and concurrent evaluation of multiple dependent variants, in order to find global optima instead of local ones. As a conclusion, we also plan to use QVTR^2 at runtime to tackle evolution when performance estimation requires runtime data.

References

1. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE TSE **30**(5) (2004) 295–310

2. International Workshop on Software and Performance (WOSP) proceedings, ACM
3. Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: WOSP/SIPEW. (2010)
4. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: An approach for qos-aware service composition based on genetic algorithms. In: GECCO, ACM (2005)
5. Grunske, L.: Identifying "good" architectural design alternatives with multi-objective optimization strategies. In: ICSE, ACM (2006)
6. Cortellessa, V., Martens, A., Reussner, R., Trubiani, C.: A process to effectively identify "guilty" performance antipatterns. In: FASE. (2010)
7. Parsons, T.: A framework for detecting performance design and deployment antipatterns in component based enterprise systems. In: DSM, ACM (2005)
8. Xu, J.: Rule-based automatic software performance diagnosis and improvement. In: WOSP, ACM (2008)
9. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3) (2006) 621–646
10. Object Management Group (OMG): MOF QVT Specification. (April 2008)
11. Muller, P., Fleurey, F., Drey, Z., Pollet, D., Fondement, F.: On executable meta-languages applied to model transformations. In: Model Transformations in Practice Workshop at MoDELS 2005. (2005)
12. Schürr, A.: Specification of graph translators with triple graph grammars. In: 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG). Volume 903 of LNCS., Springer (June 1994) 151–163
13. Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In: Model Transformations in Practice Workshop at MoDELS 2005. Volume 3844 of LNCS., Springer (2005) 139–150
14. Jouault, F., Kurtev, I.: Transforming models with atl. In: Model Transformations in Practice Workshop at MoDELS 2005. Number 3844 in LNCS, Springer (2005)
15. Hettel, T., Lawley, M., Raymond, K.: Model synchronisation: Definitions for round-trip engineering. In: ICMT. Volume 5063 of LNCS. (2008) 31–45
16. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (2005)
17. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. ACM Press (1997)
18. Smith, C.U., Williams, L.G.: More new software antipatterns: Even more ways to shoot yourself in the foot. In: Int. CMG Conference. (2003) 717–725
19. IKV++ technologies ag: Medini-qvt. http://projects.ikv.de/qvt
20. Tata Research: Modelmorf. http://121.241.184.234:8000/index.php
21. Bertoli, M., Casale, G., Serazzi, G.: Jmt: performance engineering tools for system modeling. SIGMETRICS Perform. Eval. Rev. **36**(4) (2009) 10–15
22. Tawhid, R., Petriu, D.C.: Integrating performance analysis in the model driven development of software product lines. In: MoDELS. Volume 5301 of LNCS., Springer (2008) 490–504
23. Petter, A., Behring, A., Mühlhäuser, M.: Solving constraints in model transformations. In: ICMT. Volume 5563 of LNCS., Springer (2009) 132–147
24. Avila-García, O., Estévez, A., Rebull, E.V.S.: Using software product lines to manage model families in model-driven engineering. In: SAC, ACM (2007)
25. Oldevik, J., Haugen, Ø.: Higher-order transformations for product lines. In: SPLC, IEEE (2007) 243–254