Ensuring Quality of Geographic Data with UML and OCL*

Miro Casanova, Thomas Wallet, and Maja D'Hondt

System and Software Engineering Laboratory
Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels Belgium
mcasanov | twallet | mjdhondt@vub.ac.be

Abstract. Geographic data is the backbone of sophisticated applications such as car navigation systems and Geographic Information Systems (GIS). Complexity quickly arises in the production of geographic data when trying to ensure quality. We define quality as the integrity and well-formedness of the contents of the geographic data, usually enforced by external applications where constraints ensuring quality (referred to as *quality constraints*) are implicit, low-level and scattered throughout the application code. This has significant consequences with respect to manageability, adaptability and reuse of these constraints.
This paper explains our use of UML class diagrams as conceptual model for geographic data, and how we exploited the Object Constraint Language (OCL) for describing the quality constraints in an explicit, declarative and high-level way. As our use of OCL is slightly different than it was originally intended, we present our adaptations and explain the main issues of evaluating the resulting OCL.
We are confident that our specific application of OCL can be put to use in other domains where complex constraints need to be expressed in a knowledge-oriented domain.

1 Introduction

This paper recounts how we use class diagrams from the *Unified Modeling Language* to represent geographic data in a conceptual model, and how the *Object Constraint Language* can be exploited to describe quality criteria on this domain in the form of complex constraints.

Section 2 gives a general overview of the domain of geographic data, describes the challenges in its production process, and reports on the current practices and their problems. It becomes clear that most difficulties are encountered ensuring the quality of the produced geographic data. Such quality is indeed defined by means of implicit domain constraints which are generally expressed in a low-level and implementation-dependent way, thus hard to localise, understand and modify. We explain how to overcome these drawbacks: a high-level representation of

* This research, in collaboration with *Tele Atlas Data Gent NV*, is part of the project *GeoObjects (IWT 990025)*, funded by the *Flemish Institute for the Improvement of the Scientific-Technological Research in the Industry* (IWT).

the domain knowledge is required, accompanied by a high-level, unambiguous, explicit and modular representation of the quality ensuring constraints (hereafter referred to as *quality constraints*). In this research, we use the class diagrams from UML together with a subset of OCL to achieve this. Considering the subtle differences between our application of OCL and the way OCL is usually employed, we had to make some adaptations in order to express elegantly the quality constraints. These adaptations are described in Sect. 3, where they are also extensively illustrated with real-world examples taken from the domain of geographic data.

This research is the by-product of a project we are involved in with TeleAtlas – an important supplier of geographic data in Europe – as the other partner. Since our partner manipulates a vast amount of geographic data, the checking of the quality constraints in OCL should be performed automatically. Although the development of an OCL evaluator is still ongoing work, we discuss some pertinent issues in evaluating OCL in Sect. 4.

In Sect. 5 we touch upon a few more advanced adaptations to OCL. We conclude in Sect. 6 that although this research originated in the domain of geographic data, we strongly believe that the results can be reused in other domains where complex constraints need to be expressed on a conceptual, knowledge-oriented model. In addition to this we provide some issues about our ongoing work, and hint at related work performed by other research groups or companies.

2 Quality of Geographic Data

2.1 Geographic Data

Digital geographic data, and especially data concerned with the road network, is used in sophisticated applications such as Geographic Information Systems (environmental planning and control, alarm call and dispatch), Fleet Management Systems, car navigation and geo-marketing. Suppliers of geographic data are responsible for the production process, which consists of capturing the real-world geographic data and storing it in a persistency layer. The source material typically comes from satellite images, scanned maps, Global Positioning System data, and so on. Since this may lead to mistakes creeping into the geographic data, an important and ever continuing concern in the production process is quality assurance.

The most widely accepted format for geographic data is the Geographic Data Files (GDF) standard [GDF], which has been created in order to improve the efficiency of capturing and producing road related geographic information. GDF achieves this efficiency by providing a common reference model on which clients can base their requirements and suppliers can base their product definition. The foundation of the GDF standard consists of a general, non-application specific planar-graph representation of the real world. On top of this model, a road network specific application model has been built. The last model describes real-world concepts in the domain of geographic road network data, as well as attributes of these concepts and relations between these.

2.2 Challenges in Producing Geographic Data

The geographic data market is highly competitive. Factors that determine success are among others adaptability, flexibility and time to market. Moreover, enormous effort goes into eliminating errors and inaccuracies in the contents of the geographic data in order to guarantee high quality and reduce maintenance. *Quality* of geographic data is defined here as the integrity and well-formedness of the data, and is specified at the hand of specific domain constraints. The importance of delivering high-quality geographic data is easily understood when thinking of the consequences of using data of poor quality in applications such as car navigation and alarm call and dispatch.

2.3 Current Practices and Their Drawbacks

Although most suppliers of geographic data use the high-level and implementation independent GDF standard for interchange and as a common reference model, we observe that in practice no actual model is actively and explicitly employed in the production process. In other words, an implicit mental picture is used instead of a tangible high-level representation of the geographic data and the corresponding quality constraints. The only tangible model of geographic data in current practices is its specific implementation in a file system, a commercially available database or any other medium. Since there is the inevitable impedance mismatch between the high level and the implementation level, the concepts, attributes, relations and quality constraints residing in the high level are irretrievably lost in the actual production process.

More specifically, in today's production processes, high quality of the geographic data is usually ensured by a pool of software applications that check the quality constraints. Since these applications can only operate on a low-level description of geographic data, the quality constraints are consequently described on an equally low level, i.e. in terms of the actual implementation of the geographic data. This approach results in a tangling of the implicit quality constraints in the application code, hence causing them to be non-modular and hard to localise. This has significant consequences with respect to manageability, adaptability and reuse of the constraints, making the production process prone to mistakes and resulting in increased time to market.

2.4 Setting an Approach to Satisfy the Requirements

From the previously described drawbacks of the current approaches we observe that we need a high-level, conceptual description of the knowledge in the domain of geographic data. Preferably this model should be based on a standard, such as GDF, to facilitate exchange. But more importantly, the conceptual nature of the model will ensure implementation independence and a close match with reality to minimise loss of information.

Current descriptions of geographic data, with GDF as representative example, are compatible with the philosophy of object-orientation: geographic features or concepts can be mapped to classes, whereas relations between concepts correspond to associations between classes. Because of this, but also because it is a standard in modelling, UML was selected in this project for representing a conceptual model of the geographic data.

The quality constraints that reason about the geographic data should also be expressed explicitly on a conceptual level for the same reasons. But there are other requirements involved: they need to be described in a modular manner, allowing for easy localisation and flexible manipulation of a particular constraint. Additionally, we should be able to change a quality constraint without this having to affect other constraints. Moreover, the medium for expressing these constraints should embody a level of intuitiveness and declarative power comparable to that of natural language, yet be formal and unambiguous.

These requirements and the fact that the quality constraints reason about domain knowledge that is described in UML class diagrams, point to the use of UML's accompanying OCL [KW99] [UML1.3]. OCL is used to express constraints on UML models by attaching constraints to classes (class invariants), operations on classes (pre- and postconditions), and more. This means that OCL has built-in constructs for navigating UML models, more specifically class diagrams. OCL fulfils our desires for a formal and unambiguous language, while still being relatively simple and intuitive.

3 Using OCL for Quality Constraints

We present in Sect. 3.1 parts of the conceptual model, both to illustrate the result of modelling geographic data in UML class diagrams, and to provide a context for the examples we use later on, which are realistic examples directly taken from the domain of geographic data.

In Sect. 3.2 to 3.5 we present a set of adaptations to OCL as well as their motivating reasons.[1]

We will elaborate on the consequences of these adaptations for the semantics of OCL in Sect. 4.

3.1 The Conceptual Model in UML

We will discuss one theme of geographic data here in order to use this throughout the rest of the paper as the entirely realistic context in which the example quality constraints are defined. It concerns `Relationships`, where a relationship refers to a real-world entity that needs to be represented as a mutual link between concepts and may indicate the sequence of those concepts. We will discuss two kinds of relationships: the first is `Manoeuvre`, where we can distinguish `PriorityManoeuvre`, `RestrictedManoeuvre` and `ProhibitedManoeuvre`,

[1] Note that as a starting point only a subset of OCL is selected since not all of its language constructs are required. For example, constructs related to pre- and postconditions are discarded.

and the second is ServiceBelongingToServices. Figure 1 shows the relevant part of the Relationship hierarchy.[2]

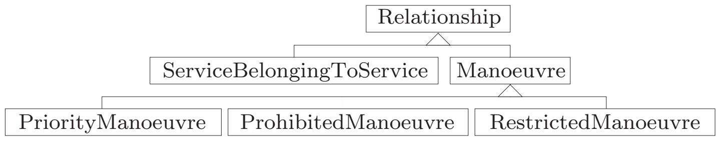

Fig. 1. The Relationship hierarchy.

Manoeuvre. A Relationship such as Manoeuvre forms a link between a number of RoadElements and a Junction, where the order of these concepts is important. A Manoeuvre indicates a certain path that can be followed by a vehicle. Figure 2 depicts the exact associations between the concepts that are involved. Moreover, it also shows the relations between RoadElement and Junction.

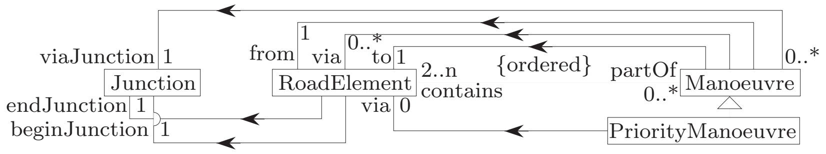

Fig. 2. The relationship Manoeuvre and its relation to RoadElement and Junction.

A PriorityManoeuvre represents a manoeuvre that has priority over other manoeuvres at an intersection. A ProhibitedManoeuvre indicates a manoeuvre that is prohibited, whereas a RestrictedManoeuvre is obligatory. Note in Fig. 2 that a PriorityManoeuvre has exactly two RoadElements which it refers to, thereby overriding Manoeuvre's behaviour.

ServiceBelongingToService. Another kind of relationship is ServiceBelongingToService, which represents the fact that two Services functionally belong to each other. Examples of services are Restaurant, RestArea, PetrolStation, AirlineAccess, Airport, etc. When a Restaurant belongs to a RestArea, they are linked together by a

[2] Since the applied modelling method in itself is outside the scope of this paper, we only show the resulting conceptual model of the examples of geographic data that are described.

ServiceBelongingToService relationship. Likewise, a PetrolStation can belong to a RestArea. Figure 3 shows the conceptual class diagram. A special kind of Service is EntryPoint, which denotes a location from which one has to leave the publicly accessible road network in order to a enter a Service. Most services are connected to the EntryPoint via a ServiceBelongingToService.

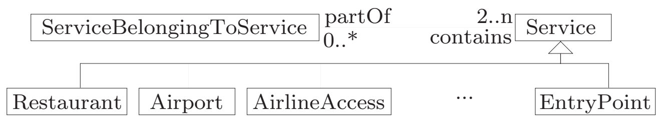

Fig. 3. The relationship ServiceBelongingToService and its relation to Service.

3.2 Constraints on Multiple Classes

The original purpose of OCL is to write *local* constraints on parts of the UML model, such as class invariants, pre- and postconditions on operations, etc. In our context, where we use UML class diagrams to represent a knowledge model, we need OCL to express *global* constraints that may potentially reason about the entire model. Below is an excellent example of a constraint that expresses some characteristic of geographic data that cannot be restrained to one single concept or class of the conceptual model.

Constraint. *A road-element shall not be the first road-element of a restricted-manoeuvre and a prohibited-manoeuvre in case both these manoeuvres refer to the same via-junction.*

It is clear that this constraint reasons about the concepts RestrictedManoeuvre and ProhibitedManoeuvre, and that neither one can be preferred over the other to serve as main context. Nevertheless, if we would choose for example the first concept as context, the constraint in current OCL would be:

```
context RestrictedManoeuvre inv:
ProhibitedManoeuvre.allInstances->forall(p : ProhibitedManoeuvre |
self.viaJunction = p.viaJunction implies self.from <> p.from)
```

This constraint shows signs of imbalance: there is an implicit looping over all instances of RestrictedManoeuvre, whereas for ProhibitedManoeuvre the loop is expressed explicitly, using the undesirable allInstances construct.[3] We would like to use this built-in language construct for all classes that occur in a

[3] In OCL specifications [UML1.3], it is explicitly said that the use of allInstances is problematic and is discouraged in most cases.

constraint. This issue was introduced in [GKR99], where they propose to allow multiple type names after the keyword `context`. We adopted a different notation, illustrated in the new version below of the example constraint:

```
RestrictedManoeuvre.viaJunction = ProhibitedManoeuvre.viaJunction
implies RestrictedManoeuvre.from <> ProhibitedManoeuvre.from
```

We use the name of the class of the conceptual model to denote the context and at the same time as the contextual instance. Note that as a result of this notation the reserved word `self` becomes obsolete.

3.3 Constraints on Multiple Instances

Class invariants in the original OCL specifications are meant principally to allow manipulation of one instance of a class at a time,[4] whereas our quality constraints require access to more than one instance of the same class at the same time in order to express some characteristic about them. The following constraint exemplifies this (see class diagram on Fig. 3):

Constraint. *Every airport which is in relation with an airline-access, shall also be related to an entry-point of this airline-access. Consequently, this airline-access itself relates to at least one entry-point.*

The services `Airport`, `AirlineAccess` and `EntryPoint` are related through some `ServiceBelongingToService` relationships. More specifically, a first one connects `Airport` and `AirlineAccess`. The constraint expresses that this should imply the connection by means of a second `ServiceBelongingToService` between `Airport` and `EntryPoint` of the `AirlineAccess`. The last two services are in turn connected by a third relationship. The main context of this constraint is the class `ServiceBelongingToService`. However, this constraint involves three different instances of this class, which cannot be expressed gracefully, as is shown below:

```
ServiceBelongingToService.allInstances->exists(s1 |
     ServiceBelongingToService.allInstances->exists(s2 |
          ServiceBelongingToService.allInstances->exists(s3 |
<expression with s1 s2 s3>) ) )
```

Again, the undesirable explicit iteration over all instances with `allInstances` is required because different instances need to be referred to in the same constraint. Therefore we propose the introduction of *identifiers* in the context of a constraint. The syntax we adopt is:[5]

```
constraint :: ('id' idDecl 'in')? exp
idDecl :: <name> (',' <name>)* (':' <type>)? '=' exp (';' idDecl)*
```

[4] It is however possible to refer to multiple instances of a same class, but this can only be achieved in a complex way using the undesirable construct `allInstances`.

[5] Syntax definitions are given in extended BNF.

An identifier consists of a name, an initial value, and an optional type. The identifier can be referred to several times within a constraint by means of its name, thus accessing its value. This value cannot be altered since constraints are side-effect free due to their declarative nature. The initial value of an identifier can be any valid OCL expression, including navigation paths such as `Manoeuvre.contains->first` and `Manoeuvre.contains->select(m | m.oclIsTypeOf(RoadElement))`.

Consequently, the previous constraint can be expressed as follows:

```
id s1, s2, s3 : ServiceBelongingToService in
    <expression with s1 s2 s3>
```

Note that our notion of identifiers closely resembles that of the *let* expression presented in [CHH+98] and adopted in the specification of OCL 1.3 [UML1.3]. It is not clear though if this construct can only be used as an abstraction and reference mechanism for expressions that are used more than once in a constraint, or if the semantics also allows it to express constraints about different instances of the same class, as does our approach.

In any case, our syntax adaptation is equally suited to name a repeated expression in a particular constraint and refer to it in this constraint. This avoids repeatedly writing long and complicated navigation paths, an advantage when trying to reduce the complexity and length of constraints, and thus the chance of making mistakes.

3.4 Dot and Arrow Notations

In [KW99] and [UML1.3] navigation through elements of a model is achieved through navigation paths using dot and arrow notations. Associations or attributes on a class in the user-defined model are accessed through the dot notation as in the following expressions: `Manoeuvre.contains`, `RoadElement.boundedBy`, etc. OCL contains a set of predefined classes and corresponding operations. Examples of these operations are `attributes` and `supertypes` defined on `OclType`, whose instances are all the types (user-defined or predefined) existing in the UML model, and `oclIsTypeOf` defined on `OclAny` which is the superclass of all types. Another category of predefined OCL classes is the `Collection` hierarchy. This class and its subclasses `Set`, `Sequence` and `Bag`, have predefined operations such as `size` and `isEmpty`, but also for instance `select` and `forall` which iterate on all elements of a `Collection`. Inconsistency results from the fact that the notation for applying operations from the `Collection` hierarchy is the arrow operator, whereas for the other predefined operations it is also the dot operator.

Therefore it is complicated to distinguish when an OCL predefined operation or a user-defined property is used. The fact that many OCL predefined classes or operations have a prefix *Ocl* highlights this.

We adopted a more consistent notation, which is easier to understand for users of OCL, and easier to evaluate. Our notation stresses the difference between user-defined and predefined properties by using the dot notation to express navigations from the former and the arrow notation from the latter, even if it does not concern an instance of the `Collection` hierarchy.

3.5 Accumulators

The *accumulator* concept, used in the predefined `iterate` operation for iterations over a `Collection`, lacks a clear definition in [KW99] and [UML1.3]. For this reason we adopted the following syntax for the single parameter `oclExpression` of this `iterate` operation:

```
oclExpression :: <name> (',' <name>)* accumulator '|' expression
accumulator :: '[' <name> (':' <type>)? '=' expression ']'
```

`accumulator` is assigned an initial value before the iteration mechanism. Then for each iteration step, the result of the evaluation of the expression following the horizontal bar is assigned to the accumulator. At the end of all the iterations, the current value of the accumulator is returned as the result of the evaluation of the iterate operation.

Our experience highlighted the fact that accumulators are necessary for context propagation in iterations. However accumulators must be expressed with a procedural-like syntax which definitely contrasts with the relatively intuitive and declarative way to write constraints with OCL. The following example presents a constraint where the use of an accumulator is needed. In the same time it shows that such a constraint requires a procedural-like syntax, thus loosing part of the intuitiveness and declarativity of OCL.

Constraint. *The set of road-elements which a manoeuvre refers to is continuous, meaning that it is an ordered set of road-elements and that each road-element, except for the last, has exactly one junction in common with the following road-element.*

```
id res = Manoeuvre.contains in
res->subSequence(2,res->size)->iterate( item
     [path = Sequence{res->at(1).endJunction}] |

     if path->isEmpty
     then Sequence{}
     else
          if item.beginJunction = path->last
          then path->append(item.endJunction)
          else
               if item.endJunction = path->last
               path->append(item.beginJunction)
               else Sequence{}
               endif
          endif
     endif)->notEmpty
```

As presented in Fig. 2, a `Manoeuvre` refers to a sequence of `RoadElements` through the association `contains`, and each `RoadElement` has two extremities (`startJunction` and `endJunction`). We iterate over this sequence (except its first `RoadElement`, which is the starting point), to check if each of its elements (represented by the identifier `item`) have a shared extremity with the previous one. The accumulator `path` is used to store the continuous sequence of shared extremities already encountered. Particularly, `path->last` is the extremity of the previous `RoadElement` that should be shared with the current element. Consequently, either the `beginJunction`, either the `endJunction` of the current element must be equal to this extremity, otherwise the continuity is broken. When the continuity is broken, a null sequence (`Sequence{}`) is assigned to the accumulator `path`. For simplification we suppose here that we know that the first `RoadElement` of the sequence is linked to the second by its end junction.

4 Evaluating Constraints

In this section we explore the main issues of evaluating constraints which are expressed using our modified version of OCL (as proposed in Sect. 3). For a further description of OCL evaluation issues, we refer to [HHK98].

4.1 Checking Navigations Paths

OCL constraints make extensive use of navigation path expressions, referring to associations, attributes or methods described in the UML class diagrams of the conceptual model. For this reason an important issue in constraint evaluation is the checking of navigation path validity against the conceptual model. Such checking must actually be done statically, in order to avoid run-time interruptions of costly quality routines due to type errors.

Whereas classical OO-language type checking is typically performed through the checking of property names, return types and parameter types, the type checking on OCL constraints is based on checking navigation paths against associations described in the conceptual model (according to role names, association cardinalities, etc.). Note however that dot and arrow notations, in the way we adopted them (see Sect. 3.4), involve a simple mechanism of navigation path checking. When type checking a constraint, a property of an expression accessed using the dot notation is checked in the user-defined conceptual model in UML (for searching the specified property, its return type, etc.). In a similar way, an operation of an expression accessed using the arrow notation is checked in the UML class diagram of the predefined OCL types.

4.2 Evaluating Constraints on Multiple Classes

As explained in Sect. 3.2, our OCL can be used for expressing constraints on several classes. The mechanism of evaluating an OCL constraint depends on the number of classes it involves. Based on the checking of navigation paths, classes involved can be identified. The constraint evaluation is achieved in as many evaluation loops as the number of classes involved. Evaluation loops are nested loops which aim to evaluate the given OCL constraint for each permutation of the instances of the classes involved, as shown in the following example:

```
RestrictedManoeuvre.viaJunction = ProhibitedManoeuvre.viaJunction
implies RestrictedManoeuvre.from <> ProhibitedManoeuvre.from
```

In such case, the constraint will be evaluated through two evaluation loops, the first one over the `RestrictedManoeuvre` instances, and the second one, nested within the first one, over the `ProhibitedManoeuvre` instances. For each loop iteration the constraint will be evaluated with the current pair of instances.

4.3 Evaluating Constraints with Identifiers

Constraints with identifiers (as presented in Sect. 3.3) are evaluated through the same evaluation loop mechanism. Evaluation of a constraint with identifiers is equivalent to the evaluation of that same constraint with the references to the identifiers within the constraint textually replaced by their value. For instance, the constraint below will be evaluated in exactly the same way as the constraint given in Sect. 4.2.

```
id rm = RestrictedManoeuvre ; pm = ProhibitedManoeuvre in
  rm.viaJunction = pm.viaJunction implies rm.from <> pm.from
```

However, when having two or more global identifiers with the same initial value, as in the following example, the evaluation mechanism is slightly different.

```
id re1, re2 = RoadElement in
  <some_constraint_using_re1_and_re2>
```

Evaluation of such a constraint will be achieved through two nested evaluation loops, each over all instances of the class `RoadElement`. The reason for having two identifiers (`re1` and `re2`) with the same value (`RoadElement`) is to be able to refer to two instances of the same class which are guaranteed to be distinct.

5 Advanced Adaptations to OCL

We already proposed some syntactic modifications to OCL and defined informally their procedural semantics. However we plan some further modifications that will simplify the process of writing and reusing quality constraints on geographic data, by introducing constraint referencing and composition as well as parametric constraints.

5.1 Referencing Constraints

In OCL specifications [UML1.3], a constraint name can optionally be written after the `inv` keyword. Because we chose to not link explicitly a context to each constraint, we had to replace this way of naming a constraint. That is the reason why we propose a mechanism for naming each constraint within its OCL definition. Sections 5.2 and 5.3 will highlight the interest of referencing such names in other OCL constraints.

The naming mechanism is compulsory for each constraint definition and is done through the following syntax:[6]

```
constraint  :: <name> ':' constraintBody
```

The following constraint illustrates the naming mechanism:

A restaurant-service shall always be referred to by a service-belonging-to-service relationship in which the other object is an entry-point

```
entryPointBelongsToRestaurant :
id s = ServiceBelongingToService.services in
s->select(s1 | s1->oclIsTypeOf(Restaurant))->notEmpty implies
s->select(s2 | s2->oclIsTypeOf(EntryPoint))->notEmpty
```

5.2 Parametric Constraints

Let's consider the three following constraints (see class diagram on Fig. 3):

*1- A **restaurant**-service shall always be referred to by a service-belonging-to-service relationship in which the other object is an entry-point*

*2- An **airline-access**-service shall always be referred to by a service-belonging-to-service relationship in which the other object is an entry-point*

*3- An **airport**-service shall always be referred to by a service-belonging-to-service relationship in which the other object is an entry-point*

Since only the class name changes, it is useful to factor it out of the OCL constraint definition, enabling us to write the OCL rule only once and reuse it with different classes. To achieve this we propose to have *parametric OCL constraints*. As with C++ templates, such constraints take one or more parametric types as arguments that can be used in the OCL expressions composing the parametric constraint.

It is important however to provide a way to bound the parametric types of a parametric constraint to avoid a defined parametric constraint from being used

[6] This is a simplified syntax. The complete syntax, allowing parametric constraints, will be presented in Sect. 5.2

with any parametric type. Inspired by *Pizza*[7] [OW97] we specified a bounding mechanism for parametric types through a `where` clause. For instance, parametric types can be bounded in the `where` clause using some predefined OCL operations such as `supertypes` (to get the direct superclasses of a class), `name` (to get the name of a class) or `allSupertypes` (to get all the ancestor classes of a class). Note that a `where` clause can only hold meta-constraints on the parametric types.

The syntax for parametric constraints is given below:

```
constraint   :: <name> parametrics? ':' constraintBody
parametrics  :: '<' <parametric> (',' <parametric>)* '>' whereExp?
whereExp     :: 'where' expression
parametric   :: "@" "a"-"z" ( "a"-"z" | "A"-"Z" | "0"-"9" | "_" )*
```

The following example shows the OCL definition of a parametric constraint that can be used for the three constraints given at the beginning of this subsection:

```
entryPointBelongsToService<@param>
where @param->supertypes->includes(Service) :
id s = ServiceBelongingToService.services in
s->select(s1 | s1->oclIsTypeOf(@param))->notEmpty implies
s->select(s2 | s2->oclIsTypeOf(EntryPoint))->notEmpty
```

The `where` clause of this example specifies that the parametric type of the constraint (`@param`) must be a direct subclass of the class `Service`.

To obtain a concrete constraint out of a parametric constraint, its parametric type(s) must be filled in. This can be done within a new OCL constraint by referencing the parametric constraint with the `%` symbol. For instance, the first of the three previous constraints is obtained as follows:

```
entryPointBelongsToRestaurant :
%entryPointBelongsToService<Restaurant>
```

Checking the validity of such an expression will be achieved by evaluating the `where` clause of the parametric constraint with the `Restaurant` type.

We can construct an OCL constraint that can be evaluated by textually assigning the parametric type of the constraint to a given type. This means that the above constraint is equivalent to:

```
entryPointBelongsToRestaurant :
id s = ServiceBelongingToService.services in
s->select(s1 | s1->oclIsTypeOf(Restaurant))->notEmpty implies
s->select(s2 | s2->oclIsTypeOf(EntryPoint))->notEmpty
```

[7] *Pizza* is an extension to Java, which offers parametric and bounded parametric types.

5.3 Constraints Composition

In geographic data quality, quality constraints are often reused or grouped together. Composing existing quality constraints reduces copy-pasting, or rewriting of parts or entire OCL constraints. With the referencing mechanism introduced in the previous subsection, it becomes relatively straightforward to realise constraint composition. As explained before, existing constraints can indeed be reused within any OCL expression by referencing them with the `%` symbol.

For instance we can compose the three constraints of the previous subsection into a more general one as follows:

```
entryPointBelongsToAllServices :
%entryPointBelongsToService<Restaurant>
and
%entryPointBelongsToService<AirlineAccess>
and
%entryPointBelongsToService<Airport>
```

This is a basic example of an `and` composition, but more complicated compositions can be created, using the full expressive power of OCL.

It is important to note that composite constraints are always independently evaluated, i.e. only their final result is used to compose the final constraint. This mechanism enables powerful features for reusing or grouping constraints.

6 Conclusion

6.1 Ongoing Work

We are currently applying the results presented in this paper to the development of an OCL evaluator. Our objective is to provide an efficient way for checking OCL constraints on the extensive geographic data of our industrial partner TeleAtlas. The resulting OCL evaluator will be integrated into a distributed quality assurance system for the geographic data production process of TeleAtlas.

6.2 Related Work

UML is well suited for modelling on a conceptual level, as is described in [BFS99]. Therefore, UML class diagrams, conforming to the principles of entity-relationship modelling, are frequently used for modelling domains that consist of large amounts of static knowledge.

A body of work exists concerning OCL and its syntax and semantics performed by the *Software and Systems Engineering Research Group* at the University of Kent, Canterbury [HHK98] [CHH+98] [GKR99]. A few other research groups and companies are working on OCL improvements. The *Klasse Objecten* [KO] holds pointers to most of these projects.

6.3 Conclusion

This paper reports on the exploitation of OCL to describe quality ensuring constraints on a domain which is essentially knowledge-oriented. This domain is represented at a conceptual level by means of UML class diagrams. The source of inspiration for this work is the domain of geographic data where constraints embody criteria for the integrity and well-formedness, in other words quality of the vast amount of knowledge.

Although this use of OCL results in an explicit, high-level and unambiguous description of the quality constraints, some adaptations to the syntax and semantics, and some additional language constructs were necessary. Once more, these adaptations originated and were established in the domain of geographic data. Nevertheless they are of a general kind and therefore applicable in other domains that require conceptual modelling of extensive amounts of domain knowledge, enriched with quality ensuring constraints. Examples of such domains are broadcast management for television and radio stations, and resource schedulers, for instance for the management of passengers, cargo, transport and so on at airports and railways.

7 Acknowledgments

Viviane Jonckers and Tom Toutenel reviewed earlier drafts of this paper. We thank them for their corrections and valuable comments.

References

[BFS99] G. Booch, M. Fowler, K. Scott. "UML Distilled: A Brief Guide to the Standard Object Modeling Language". Addison-Wesley. 1999.

[CHH+98] F. Civello, A. Hamie, J. Howse, S. Kent, M. Mitchell. "Reflections on the Object Constraint Language". In Post Workshop Proceedings of UML98. Springer Verlag, June 1998.

[GDF] "The Geographic Data Files Standard". Committee for Road Transport and Traffic Telematics of the Comité Européen de Normalisation.

[GKR99] S. Gaito, S. Kent, N. Ross. "A Meta-model Semantics for Structural Constraints in UML". In H. Kilov, B. Rumpe, and I. Simmonds editors, Behavioural specifications for businesses and systems, chapter 9, pages 123-141. Kluwer Academic Publishers, Norwell, MA, September 1999.

[HHK98] A. Hamie, J. Howse, S. Kent. "Interpreting the Object Constraint Language". In Proceedings of Asia Pacific Conference in Software Engineering. IEEE Press, July 1998.

[KO] Klasse Objecten group. `http://www.klasse.nl`

[KW99] A. Kleppe, J. Warmer. "The Object Constraint Language: Precise Modeling with UML". Addison-Wesley, 1999.

[OW97] M. Odersky, P. Wadler. "Pizza into Java: Translating theory into practice". In Conference Record of POPL 97': The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 146-159, Paris, France, 15-17 January 1997.

[UML1.3] "UML 1.3 Specifications" (including OCL). `http://www.omg.org/uml`