Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 217 (2008) 203–220
www.elsevier.com/locate/entcs

Reliable UML Models and Profiles

Kirsten Berkenkötter

Faculty of Computer Science
University of Bremen
Bremen, Germany

Abstract

Formerly, models have been used mostly in design and documentation. MDA and its surrounding techniques put them into the center of the software development process as the platform-independent model serves as foundation for tasks such as platform-dependent code generation or testing. Obviously, unambiguous models are crucial for the successful accomplishment of these tasks. The UML as the most popular modeling language is not able to ensure this which delegates the validation of models to further tasks. Our goal is to improve this situation by making models reliable as it is neither likely that another modeling language will displace UML in the near future not that a new - improved - UML version will be adopted soon. We reuse the existing OCL-based static semantics of UML and strengthen them by rectification and extension. As a result, the structural soundness of class and object diagrams is automatically ascertained and model-based tasks can be smoothly performed afterwards. Our approach supports the usage of profiles as long as these specify their static semantics on OCL. We show this by an example taken from the railway control systems domain. Behavioral soundness is not checked as we believe that it is not desirable to define one concrete behavioral semantics for UML as different application domains require different semantics at least in details.

Keywords: UML, OCL, MDA, Profiles, Validation

1 Introduction

During the last few years, Model-driven Architecture (MDA) [15] gained more and more importance in software development. This includes attended techniques such as Model-driven Development (MDD) or Model-based Testing. The main idea in MDA is the platform-independent model (PIM) that serves as source for all other activities, e.g. transformation to platform-specific models (PSM), code generation, validation, verification, or testing. Hence, the model becomes the center of the software development process.

One of the reasons for the success of MDA is the increasing popularity of the Unified Modeling Language (UML) [17,18] that consolidates a variety of modeling techniques. The thirteen diagram types and numerous modeling elements of the current version UML2 provide means to model all kinds of software systems

[1] Email: kirsten@informatik.uni-bremen.de

doi:10.1016/j.entcs.2008.06.050

independent from scale and domain. The UML is accompanied by the Object Constraints Language (OCL) [16] that allows to navigate in a model and formulate constraints on it and several other helpful mechanisms. One of them - the profile mechanism - is paid a lot of attention since the standardization of UML2 as it allows to tailor the UML to a specific domain.

New ideas in software development call for new modeling strategies. This holds especially for the UML as a successful application with respect to MDA makes new demands on the language. At the moment, UML is a loose compound of modeling techniques that lack formal semantics. Static semantics are at least partly formally defined by OCL constraints. Behavioral semantics and more static semantics are defined in natural language that is claimed to be precise. Nevertheless, lots of ambiguities are introduced. To give an example, Fecher et. al. list 29 uncertainties in the definition of UML statemachines [11].

The imprecise definition of UML is partly intended as the developers did not want to fix all semantic details to allow the application of UML for all kinds of software systems. Semantic variation points offer the possibility to adapt the UML to each domain. This is sensible as each application area has its own needs. Even if UML as a broad approach is used, developers tend to used domain-specific semantics to interpret their models. This is one reason for the success of profiles.

A completely different point is static semantics that defines the well-formedness of a model. Albeit the behavioral interpretation of models differs from domain to domain, a sound structure is needed as foundation since behavior is always related to structure in UML. Furthermore, transformations and code generation are tasks that need an unambiguous structure as source. Currently, the static semantics of UML that have been formalized are not sufficient to ensure sound models which delegates the validation of models to the tools that process them. This situation is in particular disappointing with respect to models that use profiles as each profile may define additional static semantics that have to be considered.

Approaches to formal semantics for UML tend to focus on behavioral semantics as e.g. in [11]. It is time to turn attention also on the structural well-formedness of UML models. We tackle this problem with the help of OCL as suggested in [13]. UML provides a set of OCL constraints that form the basis of static semantics. Unfortunately, many constraints are erroneous as e.g. listed in [2] or only defined in natural language. The first ones must be corrected, the latter ones formalized. Furthermore, the lengthy paragraphs intended to describe the behavioral semantics in natural language have to be carefully analyzed to detect hidden static semantics. The resulting set of OCL constraints can be automatically validated with the help of the tool USE [1,23] for class and object diagrams. In this way, we gain a reliable model that can be used by all further applications that use the model as reliable input. It is also possible to extend this process to UML profiles if these specify their static semantics in OCL [4]. We document this approach with an example profile designed for the development of railway controllers.

The paper is organized as follows: Sec. 2 presents the current degree of formalization of the static semantics of UML. After that, we show in Sec. 3 how these

semantics can be strengthened by analyzing the UML specification and formulating constraints. Sec. 4 deals with the extension to profiles, while Sec. 5 handles the automated validation process. We conclude with an outlook to future work and a discussion in Sec. 6.

2 Some Notes on Syntax and Semantics of UML

The specification of a (modeling) language is structured in two parts - *syntax* and *semantics* [8]. Syntax is split into abstract syntax that defines the elements of the language and concrete syntax that provides a usable notation for users. Semantics give meaning to a language. On the one hand, these are *static semantics* that define the correct composition of syntactic elements to programs or models and *behavioral semantics* that define their runtime properties.

This separation of concepts in language design is not mirrored in the UML specification documents. Abstract syntax is given in diagrams while concrete syntax is mostly given in textual descriptions combined with figures for each modeling element. The combination of modeling elements to diagrams is described in separate sections afterwards. Static semantics are split into OCL constraints, constraints in natural language, and partly in semantics sections given for each element, mixed with behavioral semantics. Variation points have been introduced to allow different interpretations of modeling concepts due to the fact that UML has the ambition to model all kinds of systems, independent from domain and scale.

Since its first standardization in 1998, the degree of formalization of UML semantics has been fervently discussed. The developers of UML argued - and still argue - that natural language is sufficient for the description of UML semantics. A formal specification of the language *"would have added significant complexity without clear benefit"* [17]. It is also argued that *"currently, the semantics are not considered essential for the development of tools; however, this will probably change in the future."* [17]. This point of view is underlined by the fact that tool compliance to the UML standard is only defined in ways of abstract and concrete syntax but not semantics [17,18]. A tool is UML-compliant even if e.g. the interpretation of state machines is different from the one in the UML specification.

Contrary, critics argue that a certain degree of formality is definitely needed for several reasons: clarity, expendability, interoperability [10]. It is neither desirable that different persons interpret the same model differently nor that tools do the same; especially if verification or code generation is performed. This does not mean that every detail in UML shall be explicitly specified. More convenient would be a solid foundation and a well-defined extension mechanism that allows to tailor the UML to specific domains. Numerous approaches address formal (behavioral) semantics for UML, e.g. denotational [6], by Z [7], or process algebra [12]. Most of these have in common that they focus on a subset of UML, often inspired by later usage in a specific domain like real-time systems [9]. Static semantics seem to be a poor cousin of behavioral ones as they are not discussed in detail.

Some of this criticism has been regarded in UML2. The extensibility of UML

has been increased and is better specified. Abstract syntax is described quite well in diagrams, but concrete syntax is still nebulous. It is still unclear, which elements can be used in which kind of diagram, as only recommendations are given. Furthermore, semantics are still informal and mostly given in natural language, even if OCL has been used more frequently than in older language versions. As many OCL constraints are erroneous [2,22], their expressiveness is dubious. The result is that the use of modeling elements in diagrams differs from tool to tool just as semantics and their interpretation e.g. for code generation.

Profiles add significantly to this problem due to their ability to define more static semantics. These further constraints on the composition of models belong to the metamodel level and not to the model level. A tool that intends to guarantee valid UML models must be able to check both standard and profile-dependent static semantics. This requires formalization, e.g. with OCL as already done for some constraints in UML. Currently, only few tools are able to check OCL constraints on the model level let alone on the metamodel level.

Our goal is to overcome this deficiency by a validation that includes both the static semantics of UML and those of each profile. This demands an improvement of the current static semantics of UML and a formalization of static semantics in OCL for each profile. With respect to UML, we concentrate on the frequently used modeling elements of class and object diagrams as these are available in all kinds of tools. The approach can be extended afterwards to more modeling elements. As already mentioned above, we believe that each domain has its on demands on behavioral semantics just as the need for different tools for code generation, transformation, verification, automated test case generation, etc. Model validation with profile support will reduce the development time for such domain-dependent tools significantly as the model can be assumed reliable.

3 Well-formedness of UML Models

For UML, an appropriate means to define static semantics is OCL as we can navigate in the model and perform checks automatically, e.g. with the tool USE (see Sec. 5). In the UML specification documents, each modeling element has a constraints section, where OCL is partly used. Unfortunately, these constraints are often not flawless and definitely not complete. Some static semantics are hidden in the semantics section of each modeling element that in fact should contain behavioral semantics; some - that are obvious - are not mentioned at all. To improve this situation, the following steps have to be taken: (a) rectify mistakes in existing constraints, (b) formalize constraints in natural language, (c) identify static semantics in descriptions and formalize them, (d) identify missing parts.

Rectification of Erroneous OCL in UML

The reasons of errors in OCL constraints in the UML specification range from simple syntactic problems to inconsistencies with respect to the abstract syntax or misused types [2]. To give an example, we look at the classes *Association* and

Property, defined in the *Kernel* package of UML [18]. In Fig. 1, a small excerpt of the UML metamodel that includes these classes is shown. The first problem here is related to abstract syntax. Recall that if an association end has no name, its name is the same as the class at this end with the first letter in lower case. This means that *Property* has two association ends named *association* so they are not distinct. We will use *_association* as name of the yet unnamed association end at the bottom of Fig. 1.

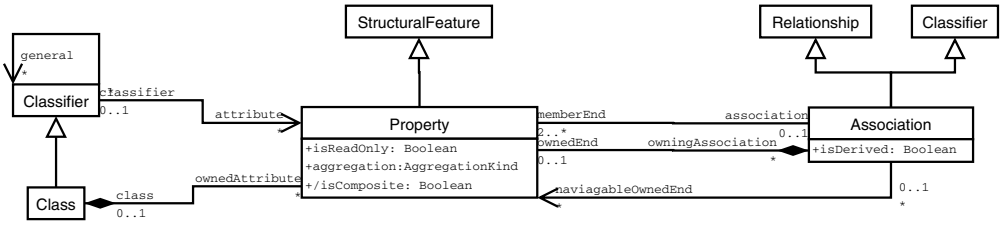

Fig. 1. Excerpt from UML metamodel - classes

Several OCL problems occur with respect to this small UML excerpt, some of these are listed below:

- As a *Classifier*, each *Association* can be specialized. In this case, the number of ends should be the same. This is formalized as follows:

```
context Association
inv:
self.parents()->forAll(p |
  p.memberEnd.size() = self.memberEnd.size())
```

  Note that *parents()* returns the set of direct parents of the classifier. This OCL constraint has to be corrected for two reasons: (a) memberEnd is an ordered set whose function *size()* must be called with the '->' operator; (b) operation *parents()* returns a set of type *Classifier*, therefore a cast must be performed.

```
context Association
inv:
self.parents()->forAll(p |
  p.oclAsType(Association).memberEnd->size() =
    self.memberEnd->size())
```

- Further, an *Association* with more than two ends, must own all its ends:

```
context Association
inv:
if memberEnd->size() > 2 then
  ownedEnd->includesAll(memberEnd)
```

  Again, there is an error as each *if-statement* in OCL has an *else* branch and ends with an *endif*. A correct version would be:

```
context Association
inv:
memberEnd->size() > 2 implies
```

```
ownedEnd->includesAll(memberEnd)
```

- With respect to class *Property*, a constraints states that only a navigable property can be marked as *isReadOnly*:

```
context Property
inv:
isReadOnly implies isNavigable()
def:
isNavigable() =
  not classifier->isEmpty() or
  association.owningAssociation.navigableOwnedEnd->includes(self)
```

  Obviously, the operation *isNavigable()* is not correctly defined, as an *Association* does not have a property *owningAssociation* as we can see in Fig. 1. Instead, a *Property* can directly refer to this attribute:

```
context Property
def:
isNavigable():Boolean =
  not classifier->isEmpty() or
  owningAssociation.navigableOwnedEnd->includes(self)
```

Formalization of Constraints in Natural Language

In the same excerpt of the metamodel, we can find an unformalized constraint for an *Association*:

- `When an association specializes another association, every end of the specific association corresponds to an end of the general association, and the specific end reaches the same type or a subtype of the more general end.`

  Naturally, there is no reason that this constraint cannot be formalized, as we see below:

```
context Association
inv:
self.memberEnd->iterate(
  e:Property;
  res : Boolean = true |
  res = self.parents()->forAll(p |
    p.oclAsType(Association).memberEnd->exists(e2 |
      (e = e2 or
       e.redefinedProperty->includes(e2))
      and
       e.type.oclAsType(Classifier).general->includes(
        e2.type.oclAsType(Classifier)))))
```

Identification of Static Semantics throughout the Specification

An attentive reading of the UML specification brings hidden static semantics to light. Throughout the document, lots of requirements of models can be found that may be formalized. One example is the class *Classifier*. In the semantics section of this modeling element, we find the following claims:

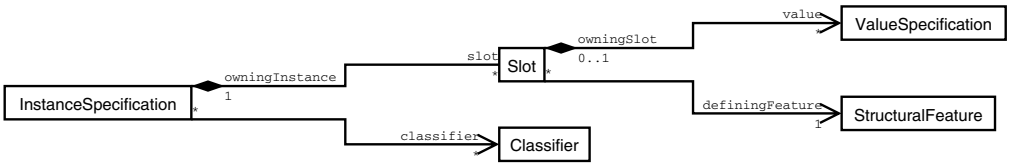

Fig. 2. Excerpt from UML metamodel - instances

- `An instance of a specific Classifier is also an (indirect) instance of each of the general Classifiers.`
  Obviously, this is not a requirement for a classifier, but for its instance that is specified by *InstanceSpecification* as we can see in Fig. 2.

```
context InstanceSpecification
inv:
classifier->forAll(c | c.parents()->notEmpty() implies
  classifier->includesAll(c.parents()))
```

- `Therefore, features specified for instances of the general classifier are implicitly specified for instances of the specific classifier.`
  This requirement focuses again on instances and not on classifiers. Instances may have slots for each structural feature of the classifier it instantiates. As generalization may be involved, also the structural features of the parent classifiers are accessible.

```
context InstanceSpecification
inv:
slot->forAll(s | classifier.attribute->includes(s.definingFeature.
                    oclAsType(Property)) or
                  classifier.parents()->exists(p | p.attribute->
                    includes(s.definingFeature.
                      oclAsType(Property))))
```

- `Any constraint applying to instances of the general classifier also apply to instances of the specific classifier.`
  Here, the constraint applies to *Classifier* and not its instances. The intention is that all constraints of classifiers apply also to their subclassifiers. Hence, the constraint must be linked - implicitly - to all subclassifiers.

```
context Classifier
inv:
parents()->notEmpty() implies
  parents()->forAll(p | self.constraint->includesAll(p.constraint))
```

Absent Static Semantics

As we have seen above, we can find a large amount of static semantics throughout the UML specification - some formalized, some not, some correctly associated to a modeling element, some not. Other constraints that are maybe too obvious are not mentioned at all, but should also be given in OCL to allow automated checks for well-formedness of models.

To give a simple example, each *NamedElement* can have a name, but must not as the multiplicity of the attribute *name* is 0..1. Each *Classifier* is a *Type* by inheritance just as *Class*, *Association*, or *DataType* who again inherit from *Classifier*. A *Type* should be identifiable by its name as the modeler must have a means to specify the type of a property. It is therefore reasonable to require that at least each *Class* and each *DataType* must have a name (a property cannot be typed by an *Association*):

```
context Class
inv:
  name->size() = 1

context DataType
inv:
  name->size() = 1
```

Another evident fact is that a class can only specialize another class while an association can only specialize another association. For *Class*, this fact is already mirrored in the abstract semantics, but not for *Association*. The same holds for *AssociationClass*. Hence, we require:

```
context Association
inv:
  general->forAll(c | c.oclIsTypeOf(Association))

context AssociationClass
inv:
  general->forAll(c | c.oclIsTypeOf(AssociationClass))
```

Other static semantics are left open intentionally due to the fact that UML can be used in various domains and in combination with various programming languages as target. One example is a semantic variation point for *Association*:

- `The interaction of association specialization with association end redefinition and subsetting is not defined.`

It is of course reasonable to define more precise constraints for modeling elements in specific applications of UML, e.g. in profile definitions as we can see in Sec. 4. It may be necessary to require that objects must have a name, or that redefining properties in generalizations must have the same name as the redefined one for code generation purposes. Nevertheless, such constraints should not be formalized in the UML metamodel in general as different projects have different requirements.

4 Well-formedness of Profiled Models by Example

Profiles as described in [17,18] are a mechanism to tailor the UML to specific application domains by (a) introducing new terminology, (b) introducing new syntax/notation, (c) introducing new constraints, (d) introducing new semantics, and (e) introducing further information like transformation rules. Changing the existing metamodel itself e.g. by introducing semantics contrary to the existing ones or removing elements is not allowed. Consequently, each model that uses profiles is a valid UML model. A UML2 profile consists mainly of stereotypes, i.e. extensions of already existing UML modeling elements. A UML modeling element is chosen as basis and add-ons are specified. In the following, we focus on the introduction of new static semantics with OCL.

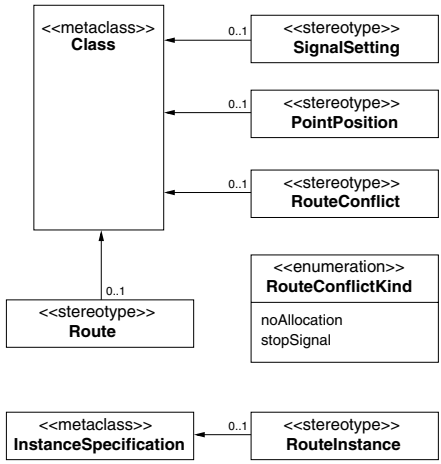

Fig. 3. Excerpt from RCSD profile

Our example is the Railway Control Systems Domain (RCSD) profile [5,20] that is designed to be used for the development of railway controllers. Its static semantics are completely defined with OCL [3,4]. Several stereotypes are defined that are elementary for the design of railway systems [19]: track segment, crossing, point, sensor, signal, and routes. In addition, domain-specific associations and datatypes are defined that are needed for accurate modeling of the domain. Base classes of the UML metamodel are *Class*, *Association*, and *InstanceSpecification*. An excerpt of the profile is shown in Fig. 3.

The RCSD profile is intended to improve the collaboration between software developers and domain experts. A specific railway system, e.g. trams, railways in Germany, or railways in Great Britain, is modeled in class diagrams as we can see in Fig. 4. The concrete projection - that is a concrete track layout with routes - is than modeled as an object diagram. Here, we provide also a domain-specific notation as shown in Fig. 5 to improve communication with domain experts. Class and object diagrams are automatically validated with respect to its static semantics as described in Sec. 5. Behavioral semantics are based on a state transition system that serves as foundation for code generation for controllers as well as formal verification by bounded model-checking. For details, we refer to [5].

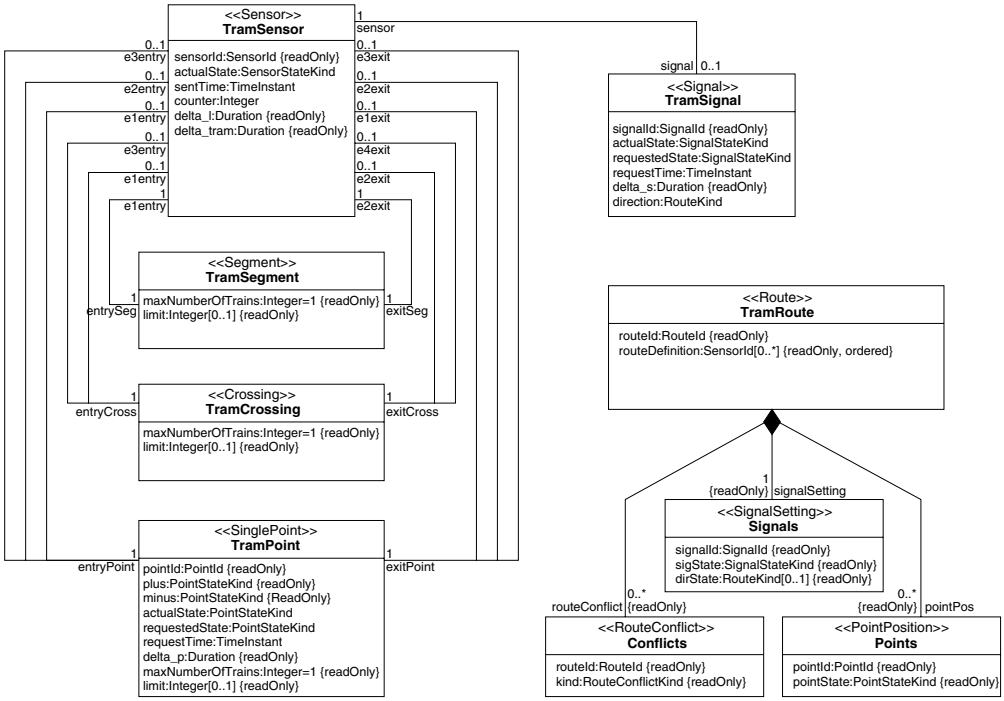

Fig. 4. Simple tram system - class diagram

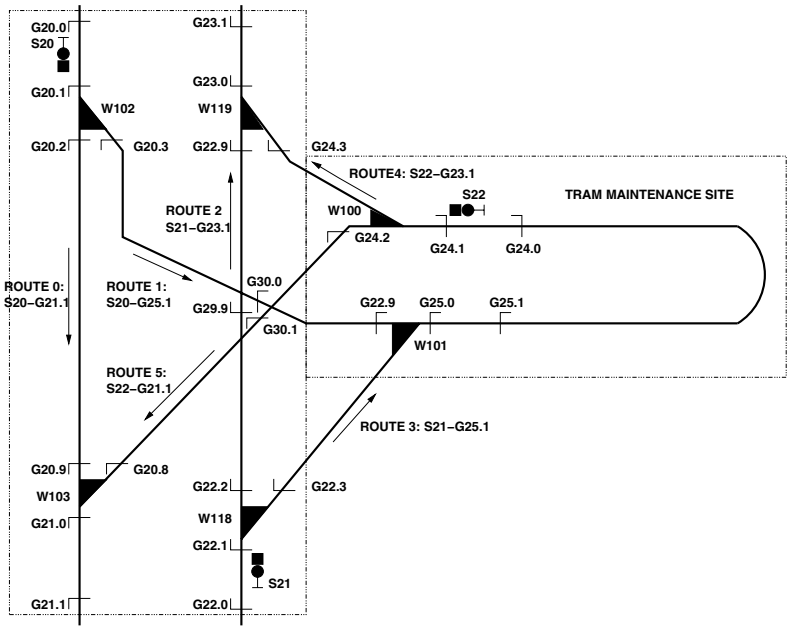

Fig. 5. Simple tram system - object diagram

OCL can be used in various ways to specify stereotypes precisely:

- Constraining property values: A stereotype has all properties of its base class and can add only attributes. Defining new associations to classes in the reference

metamodel or other stereotypes is not allowed. Therefore, constraining values of existing attributes and associations is a useful means to give a stereotype the desired functionality.

- Specifying dependencies between values of different properties of one element: Often, it is necessary to describe dependencies between the values of properties of a modeling element precisely.
- Specifying dependencies between property values of different instances of one element: Some properties like identification numbers need specific values for different instances of one element.
- Specifying dependencies between property values of different instances of different elements: In the same way, several elements may have properties whose values have some kind of relationship. Here, it is important to chose the context of the constraint carefully such that the constraint is not unnecessarily complicated because another modeling element would have been the better choice as basis for the constraint.

Constraints on Classifiers

Each *Route* must have a constant attribute *routeId* with type *RouteId*:

```
context Route
inv:
ownedAttribute->one(a | a.name->includes('routeId') and
                        a.type.name->includes('RouteId') and
                        a.upperBound()->asSequence()->first()=1 and
                        a.lowerBound()->asSequence()->first()=1 and
                        a.isReadOnly = true)
```

To understand the structure of such a constraint, a look at the UML metamodel is helpful. As all network elements are stereotypes of *Class* from the UML2 *Kernel* package (see Fig. 1), we can refer to all properties of *Class* in our constraints. Properties on the model level are instances of class *Property* on the metamodel level, which are associated to *Class* by *ownedAttribute*. As a *StructuralFeature*, *Property* is also a *NamedElement*, a *TypedElement*, and a *MultiplicityElement*, which allows to restrain name, type, and multiplicity as shown in the constraints above. Such constraints are extensively used as RCSD diagrams must provide certain information for code generation purposes.

Constraints on Instances - Properties of One Instance

Another example are *Points* that are - together with *Segments* and *Crossings* - part of the track layout. Each *PointInstance* has a *plus* and *minus* position that is modeled as an attribute. One of these has to point *STRAIGHT* and the other one *LEFT* or *RIGHT*:

```
context PointInstance
```

```
inv:
slot->select(s1 | s1.definingFeature.name->includes('minus') or
                  s1.definingFeature.name->includes('plus'))->
  one(s2 | s2.value->size()= 1 and
           s2.value->first().oclIsTypeOf(InstanceValue) and
           s2.value->first().oclAsType(InstanceValue).instance.
             name->includes('STRAIGHT')) and
slot->select(s1 | s1.definingFeature.name->includes('minus') or
                  s1.definingFeature.name->includes('plus'))->
  one(s2 | s2.value->size()= 1 and
           s2.value->first().oclIsTypeOf(InstanceValue) and
           (s2.value->first().oclAsType(InstanceValue).instance.
              name->includes('LEFT') or
            s2.value->first()->oclAsType(InstanceValue).instance.
              name->includes('RIGHT')))
```

Constraints on Instances - Properties of Different Instances of One Classifier

To give an example for this constraint category, each *RouteInstance* requires a unique identifier. We must therefore assure that the set of all identifier values of all route instances contains unique elements:

```
context RouteInstance
inv:
RouteInstance.allInstances->collect(slot)->asSet->flatten->
  select(s | s.definingFeature.name->includes('routeId'))->
    iterate(s:Slot;
            result:Set(LiteralRouteId) =
              oclEmpty(Set(LiteralRouteId)) |
            result->including(s.value->asSequence->first.
              oclAsType(LiteralRouteId)))->isUnique(value)
```

Constraints on Instances - Properties of Different Instances of Different Classifiers

Each *Route* is defined by an ordered sequence of sensors identifications. The signal setting for entering the route and sets of required point positions and of conflicts with other routes are further necessary information. This implies that the identification numbers belong to existing instances, e.g. the sensor identifications given in the definition of a route. Hence, the following constraint must hold for each *RouteInstance* with respect to the named *SensorInstances*:

```
context RouteInstance
inv:
let i:Set(Integer) =
  slot->select(s | s.definingFeature.name->includes
    ('routeDefinition'))->asSequence->first().value->
      iterate(v:ValueSpecification;
```

```
            result:Set(Integer)=oclEmpty(Set(Integer)) |
            result->including(v.oclAsType(LiteralSensorId).value))
in
  i->forAll(id | SensorInstance.allInstances->exists(sens |
    sens.slot->select(s | s.definingFeature.name->
      includes('sensorId'))->asSequence->first().value->first().
        oclAsType(LiteralSensorId).value = id))
```

5 Automated Validation with USE

The concrete validation process is performed with the tool USE [1] that expects a (meta)model in textual notation as input. For syntax, we refer to [22] and [23]. The tool implements the set-theoretic semantics described in [21] in detail. The key feature for our purposes is the interpreter that evaluates OCL constraints. The evaluation time is dependent on the number of elements in the model, the number of instances of the modeling element, the number of constraints and also the kind of constraints that range from very simple to quite complex as we have seen in Sec. 3 and Sec. 4. In general, we can only say that the complexity is polynomial with respect to the named inputs.

In our case, the input model is the UML metamodel - respectively a part of it that is necessary for class and object diagrams as described below. Further, profile can be added to the metamodel. On this basis, instance models can be checked with respect to the invariants in the metamodel. The instance model consists of both class layer and object layer. A similar application of USE with respect to the four metamodeling layers of UML is shown in [13]. Here, the application of USE in metamodeling is shown by a small example.

Our metamodel is constructed from one input file for the UML2 metamodel and one input file for each applied profile. The result is one large metamodel. The reasons for this procedure is simple as we want to be able to check models that do not apply profiles and models that apply one or more profiles. Also, we are interested if a profile is compliant to UML. So far, no OCL tool is capable of checking the consistency of a set of OCL constraints. Therefore, we assume a profile compliant to UML as long as both the constraints in the metamodel and the constraints in the profile(s) are all valid which is of course no proof. At least, we are able to test the compliance of profiles and their reference metamodels by example.

Modeling the UML Metamodel and the RCSD Profile with USE

In the metamodel file, a description of classes with attributes and operations, associations, and OCL constraints is expected. OCL expressions are either invariants, definitions of operations, or pre-and postconditions of operations. Only operations whose return value is directly specified in OCL and not dependent on preconditions are considered side-effect free and may be used in invariants. For the validation process all invariants must be fulfilled by the instance model(s).

From the UML metamodel, the packages *Kernel*, *Dependencies*, *Interfaces*, and

BasicBehaviors have been modeled with few changes: (a) The erroneous OCL statements have been corrected and additional constraints added as described in Sec. 3. (b) Some names - mostly association ends - had to be changed to guarantee unambiguous navigation in the model. (c) USE does not support *UnlimitedNatural* as type. This problem has been overcome by using *Integer* and additional constraints that restrict corresponding values to $\mathbb{N}$. We do not support nested packages. A short excerpt of the metamodel in the textual notation of USE is shown below:

```
class Association < Classifier, Relationship
attributes
  isDerived:Boolean
end
...
association Association2Type between
  Association[*]
  Type[1..*] role endType
end
...
context Association
inv Association_1:
  self.parents()->forAll(p |
    p.oclAsType(Association).memberEnd->size()=self.
    memberEnd->size())
```

Profiles are not directly supported by USE. This problem has been overcome by modeling each stereotype as a subclass from its base class, i.e. a metamodel extension. Modeling profiles as restricted extensions to metamodels is feasible with respect to [14]. Here, modifications to metamodels are classified in level one (all extensions to the reference metamodel allowed), level two (new constructs can be added to the referenced metamodel, but existing ones cannot be changed), level three (each new construct must have a parent in the reference metamodel), and level four (new relationships are only allowed as far as existing ones are specialized. The lower levels include all restrictions of the levels above. Therefore, profiles can be considered a level four metamodel extension and modeled as such in USE. [2] The profile designer must keep in mind that associations cannot be added and existing attributes and associations can only be restricted, e.g. my narrowing a multiplicity.

Checking Compliance on Class and Object Level

Evaluating constraints is possible for instances of the given (meta)model. In our case, this includes the class and object level of UML, as classes and instances are both defined in the metamodel and can be instantiated. Classes, associations, objects, links, etc. form one large instance model that is generated from a UML model created with some kind of CASE tool. The generation of USE code is decoupled

[2] [14] considers profiles as level three which is incorrect as the relationship restriction has to be respected by profiles.

from the CASE tool in use to preserve independence. Tool-specific is only the parser of the output of the tool. This step can be hopefully omitted in the future as we expect tools to comply to the XMI standard soon.

The generation of USE code itself is straightforward as the main tasks are creating instances of the elements in the input model, setting properties, and instantiating associations. The correct implementation of types, inheritance, and interfaces requires some attention, but is not complicated. Important is the correct order of instances creation as obviously the more general parts must be created before the specific parts of the instance model. The complexity - dependent on the number of classes, attributes, methods, objects, etc. never increases $O(n^3)$.

As an example, a tram network description is used on class level. Tram networks consist of segments, crossings, and single points that are all used unidirectionally. Furthermore, there are signals, sensors, and routes. This constellation is shown in Fig. 4. The class diagram is contained in one USE input file. An excerpt from the corresponding USE instance model on class level is shown in the following:

```
!create TramSensor:Sensor
!set TramSensor.name := Set{'TramSensor'}
...
!create sensorId:Property
!set sensorId.name := Set{'sensorId'}
...
!insert (sensorId, SensorId) into TypedElement2Type
!insert (TramSensor, sensorId) into Class2Property
```

A concrete network of a tram maintenance site with six routes is shown in Fig. 5. The explicit route definitions have been omitted for the sake of brevity, but can be easily extracted from the figure. This diagram has been used for the validation on the instance level. It consists of 12 segments, 3 crossings, 6 points, 25 sensors, 3 signals, and 6 routes, specified in a second USE input file. In this way, it is possible to create one input file for each object diagram and validate them separately in combination with the class diagram file. The object diagram files have the same appearance as the class diagram ones as we can see in the following excerpt:

```
!create s1:SegmentInstance
!insert (s1, TramSegment) into InstanceSpecification2Classifier
...
!create s1exit:Slot
!insert (s1exit, e2Exit) into Slot2StructuralFeature
!create s1exitValue:InstanceValue
!insert (s1exit, s1exitValue) into Slot2ValueSpecification
!insert (s1, s1exit) into InstanceSpecification2Slot
```

Results

In this example, all invariants have been fulfilled. The correctness of the OCL constraints can be easily checked by adding intentional errors like incorrect association

ends or signals with the same identification number. USE facilitates tracing of such errors by (a) showing which instance of the metamodel has violated an invariant and by (b) decomposing the invariant in all sub-clauses and giving the respective evaluation.

Some effort has to be made with respect to the USE model. Once this model is ready, all UML models can be checked for well-formedness. The same holds for profiles; the UML metamodel must be extended for each profile to allow validation of models with USE. As this task is performed once per profile, the effort seems reasonable. With respect to the RCSD profile, the instance model on class level has to be modeled once per specific railway system, e.g. once for trams. With this part of the instance model, all kinds of concrete track layouts can be checked. The tram example consists of approximately 1500 lines of input data to USE dedicated to the class level. These are generated from class diagrams by parsing the output of CASE tools and mapping them to the USE input language. Concrete track layout are generated - also automatically - from object diagrams. In this way, all kinds of track layouts for one system can be checked. The example track layout requires about 5000 lines of USE code.

6 Discussion

The validation of UML models has been proven useful in several ways: (a) it can be shown that a model complies to the static semantics of UML, (b) it can be shown by example that a profile complies to the static semantics of UML, and (c) it can be shown that a profiled model is valid according to the - added - static semantics of the profile.

As the validation is performed automatically, it is highly useful to ensure the soundness of a model before other tasks such as verification, simulation, or code generation are performed. The validation of the model for each of these tasks can be omitted as the model is assumed reliable. Another effect of the validation with USE is the improvement of the OCL constraints themselves. As most case tools have no OCL support, it is hard to detect if constraints exhibit syntax errors or if complicated constraints really have the intended meaning.

Some effort has to be made to attain this goal. First, static semantics of UML have to be improved as they are erroneous and incomplete. Second, static semantics for a profile must be defined formally. Both UML and a potential profile must be expressed as a USE model to allow automated validation. As these tasks are performed once (per profile), the effort seems reasonable. The generation of USE input code based on CASE tool output is performed automatically.

So far, most work on UML semantics focuses on behavioral semantics - e.g. [6], [7], [9], or [12] - while static semantics are not discussed in detail. As we can see from these examples, these behavioral semantics differ from approach to approach; not at least a result from the different application domains their developers had in mind. This seems appropriate as each domain has it own needs. In contrast, we believe that static semantics for UML can be unified and formalized as shown in

this paper to allow automatic validation. The resulting models are reliable and can be used in further tasks, independent from the application domain.

The validation of OCL constraints of the UML metamodel with USE has been performed for UML1 and UML2.0 before as shown in [22] and [2]. This has been extended to UML2.1 in this work. We are not aware that a thorough correction of existing constraints and formalization of constraints in natural language and of absent constraints has been performed before for class and object diagrams of UML2. In [11], similar work is done with respect to the static and behavioral semantics of statecharts given in natural language in UML2. The usage of OCL in metamodeling has been suggested in [13], but the application of this concept for profiles and the automatic validation of profiled models designed with CASE tools has not been performed before. More details with respect to the example profile can be found in [4].

The validation of profiles has been shown by an example. An adaption of the validation process to other profiles can be performed straightforward as the same kinds of constraints should appear. Validation is sensible for each profile whose application relies on a solid and unambiguous model. The RCSD profile used as example has shown that this is possible for real-world applications and problems. Even complex constraints can be formalized with OCL. Future work should investigate the usage of OCL for the formalization of static semantics of other profiles.

Also, the improvement of static semantics of UML should be pushed further to stabilize the backbone of the language. In this paper, only the basic features of UML in class and object diagrams have been considered. More work in this direction seems reasonable as a new UML standard cannot be expected in the next few years as we know from experience from the standardization of UML2. Furthermore, results for static semantics are valuable as they can be incorporated in new UML versions.

References

[1] *A UML-based Specification Environment*, http://www.db.informatik.uni-bremen.de/projects/use/.

[2] Bauerdick, H., M. Gogolla and F. Gutsche, *Detecting OCL Traps in the UML 2.0 Superstructure*, in: T. Baar, A. Strohmeier, A. Moreira and S. J. Mellor, editors, *Proceedings 7th International Conference Unified Modeling Language (UML'2004)*, LNCS **3273** (2004), pp. 188–197.

[3] Berkenkötter, K., *Design of a Railway Domain Profile and its OCL-based Validation*, in: B. Demuth, D. Chiorean, M. Gogolla and J. Warmer, editors, *OCL for(Meta-)Models in Multiple Application Domains 2006*, Electronic Communications of the EASST **5** (2007).

[4] Berkenkötter, K., *OCL-basesd Validation of a Railway Domain Profile*, in: T. Kühne, editor, *Models in Software Engineering*, LNCS **4364** (2007), pp. 159–168.

[5] Berkenkötter, K. and U. Hannemann, *Modeling the Railway Control Domain rigorously with a UML 2.0 Profile*, in: J. Górski, editor, *Safecomp 2006*, LNCS **4166** (2006), pp. 398–411.

[6] Breu, R., U. Hinkel, C. H. andCornel Klein, B. Paech, B. Rumpe and V. Thurner, *Towards a Formalization of the Unified Modeling Language*, in: M. Aksit and S. Matsuoka, editors, *Proceedings of ECOOP'97 - Object-Oriented Programming*, LNCS **1241** (1997), pp. 344–366.

[7] Clark, T. and A. Evans, *Foundations of the Unified Modeling Language*, in: *2nd Northern Formal Methods Workshop*, electronic Workshops in Computing (1998).

[8] Clark, T., A. Evans, P. Sammut and J. Willans, "Applied Metamodeling," Xactium, www.xactium.com, 2005.

[9] Damm, W., B. Josko, A. Votintseva and A. Pnuelli, *A Formal Semantics for a UML Kernel Language*, http://www-omega.imag.fr/doc/d1000009_6/D112_KL.pdf (2003).

[10] Evans, A. and S. Kent, *Core Meta-Modelling Semantics of UML: The pUML Approach*, in: *UML*, LNCS **1723** (1999), pp. 140–155.

[11] Fecher, H., J. Schönborn, M. Kyas and W. P. de Roever, *29 New Unclarities in the Semantics of UML 2.0 State Machines*, in: K.-K. Lau and R. Banach, editors, *ICFEM*, LNCS **3785** (2005), pp. 52–65.

[12] Fischer, C., E.-R. Olderog and H. Wehrheim, *A CSP View on UML-RT Structure Diagrams*, in: H. Hußmann, editor, *FASE*, LNCS **2029** (2001), pp. 91–108.

[13] Gogolla, M., J.-M. Favre and F. Büttner, *On Squeezing M0, M1, M2, and M3 into a Single Object Diagram*, Technical Report LGL-REPORT-2005-001, Ecole Polytechnique Fédérale de Lausanne (2005).

[14] Jiang, Y., W. Shao, L. Zhang, Z. Ma, X. Meng and H. Ma, *On the Classification of UML's Meta Model Extension Mechanism*, in: *The Unified Modelling Language: Modelling Languages and Applications*, 2004, pp. 54–68.

[15] Object Management Group, *MDA Guide Version 1.0.1*, http://www.omg.org/docs/omg/03-06-01.pdf (2003).

[16] Object Management Group, *Object Constraint Language*, http://www.omg.org/docs/formal/06-05-01.pdf (2006).

[17] Object Management Group, *Unified Modeling Language: Infrastructure*, http://www.omg.org/docs/formal/07-02-06.pdf (2007).

[18] Object Management Group, *Unified Modeling Language: Superstructure*, http://www.omg.org/docs/formal/07-02-05.pdf (2007).

[19] Pachl, J., "Railway Operation and Control," VTD Rail Publishing, Mountlake Terrace (USA), 2002, iSBN 0-9719915-1-0.

[20] Peleska, J., K. Berkenkötter, R. Drechsler, D. Große, U. Hannemann, A. E. Haxthausen and S. Kinder, *Domain-Specific Formalisms and Model-Driven Development for Railway Control Systems*, in: *TRain workshop at SEFM2005*, 2005.

[21] Richters, M., "A Precise Approach to Validating UML Models and OCL Constraints," BISS Monographs **14**, Logos Verlag, Berlin, 2002, Ph.D. thesis, Universität Bremen.

[22] Richters, M. and M. Gogolla, *Validating UML Models and OCL Constraints*, in: A. Evans, S. Kent and B. Selic, editors, *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings*, LNCS **1939** (2000), pp. 265–277.

[23] Richters, M. and M. Gogolla, *OCL: Syntax, Semantics, and Tools*, in: T. Clark and J. Warmer, editors, *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, LNCS **2263** (2002), pp. 42–68.