Vladiela Petraşcu, Dan Chiorean, Dragoş Petraşcu

Techniques for a Rigorous Design of Languages and Applications

Cluj University Press

Acknowledgements

This work was supported by CNCSIS - UEFISCSU, project number PNII – IDEI 2049/2008.

Preface

This book has been conceived, realized and printed with the support of the CNCSIS - UEFISCSU research project ”Extensive Metamodeling-based Framework for Specifying, Implementing and Validating Languages and Applications”, acronym EMF_SIVLA (CUEM_SIVLA in Romanian).

As both the project and the book titles suggest, the present work has been focused on using modeling and rigorous specification techniques in producing software. The development paradigm has been Model Driven-Engineering (MDE) and the main instrument used to support the development of reliable and predictable software has been the Design by Contract (DBC) method. At the model and metamodel levels, assertions have been specified using Object Constraint Language (OCL), the standard language proposed by the Object Management Group (OMG).

The book is structured in seven chapters, four of them, chapters 3, 4, 5 and 6, being based on papers written by the authors and published in the context of the EMF_SIVLA project. The first chapter, *The Necessity and Role of Assertions in Modeling*, contains two sections, *Design by Contract* and *Model-Driven Engineering* and briefly introduces the design method and the paradigm, respectively. In the next chapter, entitled *Object Constraint Language*, the authors present the main features of this language, highlighting controversial aspects found in the last standard version, 2.3.1, and proposing solutions for these drawbacks or describing how a part of them have been managed in OCLE. The chapter ends with a very short presentation of OCLE. The third chapter, *Constraints Patterns in Object-Oriented Modeling* presents authors' proposal for specifying constraint patterns in OCL. The given solutions fix or improve some wrong patterns (anti-patterns) existing in the literature. This may represent a real support for developers using OCL, irrespective of the fact that they consider this work difficult or not. In *The static Semantics of (Meta)Modeling Languages*, the authors focus on a crucial problem referred by them as ”model compilability” (by similarity with programming languages). This is due to the fact that, in order to have a predictable model-to-code (M2C) transformation, models have to comply with the specification of their modeling languages. In the second part of this fourth chapter, advice and rules for producing correct Well Formedness Rules (WFRs) are presented and exemplified in UML, MOF, Ecore, XCore. In the fifth chapter, *Domain Specific Modeling Languages*, the authors describe the specification and validation of a language entitled ContractCML. In chapter six, *Avoiding Hastiness in Modeling*, based on their experience, the authors try to explain why hastiness represents a real danger in modeling and provide some guidance to developers in avoiding superficiality. Finally, in the last chapter, *Producing Java Applications Using Models: Conclusions Drawn from Case Studies*, functionalities offered by OCLE in transforming UML models complemented with OCL specification in Java applications are presented. Based on the feedback received from students using the tool in projects and also from tool usage in software development, the authors conclude that it worth using models in producing software. Moreover, some new functionalities and the improvement of some existing functionalities in OCLE are described.

Authors' contribution to this book is as follows: Vladiela Petraşcu authored chapter 1, coauthored chapters 2, 3, 4 and 6 with Dan Chiorean and chapter 5 with Dragoş Petraşcu; Dan Chiorean coordinated the book, authored chapter 7 and the Preface, and coauthored chapters 2, 3, 4 and 6. The most influential contribution of Vladiela was in chapter 5; Dan's most influential contribution was in chapters 2, 3, 4, and 6.

Dan Chiorean
Vladiela Petraşcu
Dragoş Petraşcu

Contents

Chapter 1
The Necessity and Role of Assertions in Modeling

1.1 Design by Contract

Software contracts originated from C.A.R Hoare [50] with his so called ”Hoare Triples”. Hoare defines the relationship $\{P\}A\{Q\}$, where P and Q are assertions and A is an operation. The relationship is defined as follows: if P is an assertion which is evaluated to true, then, after the operation A, the assertion Q will be true. The phrase *Design by Contract* (DBC) was coined by Bertrand Meyer in connection with the Eiffel programming language and first described in [55] and after in [56], [57], [58], [59]. Consequently, this first section of this chapter contains a brief review of concepts and ideas described in Meyer’s above mentioned work.

The core proposal is that of a contractual approach to the development of object-oriented software components[1], based on the use of *assertions*. The approach has been aimed at increasing the reliability of object-oriented software. *Reliability* has been defined as a software quality attribute concerned with both *correctness* (software’s ability to behave according to its specification) and *robustness* (the ability to properly handle situations outside of the specification) [58]. Roughly speaking, reliability concerns the absence of errors, being a critical quality in the context of large-scale software reuse, as promoted by the object-oriented paradigm.

The understanding and putting into practice of the DBC principles has been expected to positively contribute to [58]:

- the development of correct and robust object-oriented systems;
- a deeper understanding of inheritance and related concepts (overriding, polymorphism, dynamic binding), by means of the *subcontracting* concept;
- a systematic approach to *exception handling*.

In the following subsections, we will only provide basic knowledge of DBC, as required by the chapters of this book. For further details (especially regarding subcontracts and exception

[1] In the context of Design by Contract, the term *component* is used in its general acceptance, as a software entity intended to be (re)used by other software entities.

handling), the interested reader is invited to refer to [58, 59]. In fact, the overview of concepts we provide in the following is based on these two references precisely.

1.1.1 Software Contracts. Assertions

Following the reasoning from [58], let us imagine an algorithm/routine designed to solve a particular task. Assuming the task in question as non-trivial, it would naturally allow a top-down decomposition in subtasks, as shown below (the sequential structuring of subtasks does not restrain generality).

```
Algorithm mainTask is:
  @subTask_1;
  @subTask_2;
  ...
  @subTask_n;
End-mainTask
```

Each of the subtasks above may be either inlined or triggering the call of a subroutine. The call of a subroutine to accomplish a subtask may be thought of by analogy to the real-life situation in which a person (to which we refer as the *client*) asks the services of a third-party (the *supplier*/*provider*) to accomplish a task he cannot or would not carry out personally. As an example, the task of delivering a package to a certain address in a foreign city may be solved by contracting the services of a fast-courier company [58].

Such human contracts involving two parts have the following defining characteristics.

1. They set the terms of a relationship. For each part, the contract stipulates some benefits in exchange of some obligations; a benefit for a part in the contract is an obligation for the other.
2. Both benefits and obligations are clearly stated in the contract, which protects the interests of each part involved. In addition, the contract may also reference general laws or regulations that should be obeyed by both parts.

According to DBC, the same principles should apply to software construction. Each time a routine depends on the call of a subroutine for executing one of its subtasks, there should be an explicit, contractual specification of the relationship established among the client (the caller) and the provider (the callee). The clauses of such a software contract should be formalized by means of *assertions*.

Definition 1.1 (*Assertion* [59]). An *assertion* is an expression involving a number of software entities, which states a property that these entities will have to fulfill at certain stages of software execution.

The closest concept to an assertion is that of predicate; in Eiffel, as well as in other object-oriented programming or modeling languages supporting this concept, assertions are stated as boolean expressions. Some kinds of assertions (preconditions and postconditions) apply to individual operations; others (invariants) transcend all individual operations, applying to all instances of a class. Assertions are specified at the class level and evaluated before the execution of an operation - in case of preconditions, after executing its body - in case of postconditions, and after the execution of modifiers that may change the object's state - in case of invariants.

1.1.2 Pre/Post-conditions

A primary use of assertions concerns the semantic specification of a class' methods. The task accomplished by such a method can be described by means of two assertions - the *precondition* and the *postcondition*; the former states which are the properties that should be fulfilled prior to any call of the method in question, while the latter encloses those properties that are guaranteed after the call.

A <precondition, postcondition> pair for an operation expresses the software contract that the method in question (provider of some service) publishes for its callers (clients of that service) [59]. Similar to human contracts, this also imposes demands and provides advantages for the parts involved. Namely:

- The precondition defines the situations in which a method call is legitimate; it is an obligation for the client (caller) and a benefit for the provider (method);
- The postcondition states which properties should be fulfilled once the execution has ended; it is an obligation for the service provider (method) and a benefit for the client (caller).

The contractual clause according to which the precondition is to be regarded as a benefit of the service provider is, by far, the major contribution brought by the DBC methodology in the field of software reliability [59]. This obviously leads to a simplification in programming style, since the explicit stating of a property as a precondition makes its testing inside the method body useless. Moreover, this in not only useless, but even inacceptable, as argued by the DBC *non-redundancy principle* saying that "*The body of an operation should never check for its precondition!*".

The above principle is the exact opposite of the ideas promoted by the *defensive programming* approach, according to which redundant checks contribute to an increased code reliability. Yet, for medium-sized to large systems, such checks do only increase the code volume and complexity. From here, the chance of introducing more errors, followed by the necessity of more tests and so on. Obeying to the Design by Contract theory involves placing the responsibility related to the verification of a particular consistency constraint on a part only; either the constraint appears as a precondition and then it should be solely checked by the client, or it does not and then it should be treated as a special case by the service provider.

In Eiffel, the original DBC language, assertions are part of the language itself, allowing for run-time monitoring. Any run-time assertion violation is an evidence of a bug. A precondition violation points to a bug in the client, which has not obeyed to the rules of a valid call; a postcondition violation points to a bug in the method implementation, which has failed to fulfill its contractual obligations.

1.1.3 Invariants

Preconditions and postconditions capture properties of individual operations. In addition to these, it is also possible to express global properties for the instances of a class, that should be preserved by all its public methods. Such properties compose the class *invariant*; an invariant captures all the semantic constraints and integrity rules applying to the class in question [59]. While pre and postconditions can be employed by both the structured and the object-oriented approach, invariants only make sense in the context of object-orientation.

The invariant of a class *C* consists of a number of properties that any class instance has to fulfill in any of its stable (observable) states, namely:

- Immediately after its creation as a result of a constructor call;
- Before and immediately after each call of a public method of the class on the instance in question.

Figure 1.1 illustrates the lifecycle of an object *a*, as a sequence of transitions among observable states. These states are, from left to right, the state following the creation of *a* (S1), as well as the states entered after the execution of any public method of the class on the object in question (S2, S3, S4, etc). In any of these states, *a* should satisfy the invariant; its fulfillment is however not required in the time interval between states (during the execution of a method).

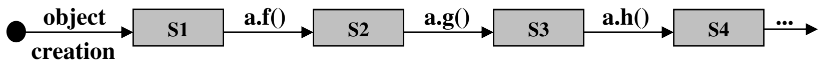


Fig. 1.1: Lifecycle of an object (Figure 5 of [58])

From here, it follows the definition of a class invariant.

Definition 1.2 (*Class Invariant* [59]). An assertion *I* is a correct invariant for the class *C* if and only if the following conditions hold:
(C1) Each constructor of *C*, applied to arguments that fulfill its precondition, in a state in which the class attributes have default values, leads to a state in which *I* is fulfilled.
(C2) Each public method of the class, applied to a set of arguments and to a state satisfying both *I* and the method precondition, leads to a state satisfying *I*.

The responsibility of preserving the invariant concerns the public methods of the class exclusively; a private method, which cannot be accessed by clients, will not be affected by the invariant.

1.1.4 Correctness of a Class

The assertion types introduced in the previous subsections provide the basis for defining the concept of *class correctness*. Informally speaking, a class is said to be correct with respect to its specification if and only if its implementation, as given by the method bodies, is consistent with its preconditions, postconditions and invariant.

To give a formal definition of class correctness, let C be a class and INV its invariant assertion. For an arbitrary method r of C, with arguments x_r, let us denote by $pre_r(x_r)$, $post_r(x_r)$ and $body_r$ its precondition, postcondition and body, respectively. Let $default_C$ be an assertion stating that the attributes of C have default values for their types. With these notations, we can provide the following general definition for the correctness of a class.

Definition 1.3 (*Correctness of a Class* [59]). The class C is said to be correct with respect to its assertions (pre/post-conditions and invariant), if and only if the following conditions hold:

$$[default_C \wedge pre_p(x_p)]\ body_p\ [post_p(x_p) \wedge INV], \tag{1.1}$$

for each class constructor p and each set of valid arguments of p - x_p, and

$$[pre_r(x_r) \wedge INV]\ body_r\ [post_r(x_r) \wedge INV], \tag{1.2}$$

for each public class method r and each set of valid arguments of r - x_r.

1.1.5 The Purpose of Using Assertions

As emphasized in [59], there are at least four major advantages deriving from the use of assertions in software development, the latter two requiring run-time monitoring facilities:

1. Support in writing correct software, including the means to formally define correctness;
2. Support for a better software documentation;
3. Support for testing, debugging and quality assurance;
4. Support for the development of fault tolerant systems.

The first above-mentioned benefit is, by far, the most important. Identifying and stating precisely both the requirements of each method and the global class properties allows developers to better understand the problem and write code correctly from the beginning. Therefore, the formal specification of all reusable software entities, on whose correctness rely a number of other applications, becomes a mandatory requirement. The writing of explicit contracts comes as a prerequisite of their enforcement in software.

The documentation role of assertions is itself essential when it comes to the development of reusable assets. Preconditions, postconditions and invariants provide potential clients of a module with basic knowledge concerning its functionality, in a short and precise form. The absence of such information can have disastrous effects on the correctness of client code, as taught by the Arianne 5 failure, reported in [51]. In this respect, a language with embedded assertion support, such as Eiffel, is highly convenient, since it enables the automatic generation of documentations by specialized tools; in addition, the chance of a gap between specification and implementation is reduced, since embedded assertions are easy to be kept synchronized with the implementation itself.

The ability to execute assertions at run-time is the major contribution brought by the DBC methodology in software development [59]. As previously stated, any run-time assertion violation points to a bug: a precondition failure points to a bug in the client, while a postcondition or invariant failure points to a bug in the provider. Therefore, the major goal of assertion monitoring is debugging.

Different levels of assertion monitoring may be employed; from lowest to highest, these are: preconditions only, preconditions and postconditions, or all three types of assertions. While testing, the highest level is always recommended; during operation however, this decision is a tradeoff between the trust in the provided code, the desired efficiency level and the critical nature of the system to be developed.

1.2 Model-Driven Engineering

Model-Driven Engineering (*MDE*) is a software engineering paradigm proposed at the middle of 2000 [21, 22, 76], which promises to revolutionize software development, by automating a major part of the process. As inferable from its name, the MDE vision relies on extensive use of models, which are no longer "contemplative" artifacts, as they were in traditional software engineering, but turn into operational "first class citizens" of software development. Therefore, by analogy with the object-oriented paradigm that has been guided by the law "*Everything is an object*", the core principle of MDE may be rightly captured by the phrase "*Everything is a model*" [21].

The MDE approach originates in the reality that software systems are complex entities, exposing several views or aspects. Each such view can be captured by a *model*, which is conformant to its *metamodel*. The metamodel provides the concepts in terms of which the model is defined, along with the relationships established among these concepts.

Definition 1.4 (*Model*). A *model* is an abstract description of a system that enables us to answer questions about the system.

Definition 1.5 (*Metamodel*). A *metamodel* is a model that describes the concepts and relationships used in the model. As the name suggests, the metamodel is the model of a (several) model(s). In other words, a metamodel is a model of a language. Models have to conform to their metamodel.

Definition 1.6 (*Meta-metamodel*). A *meta-metamodel* is a model of a (several) metamodel(s). In the OMG (Object Management Group) approach, the meta-metamodel conforms to itself. A meta-metamodel is a model enabling to describe any kind of abstract language.

By exposing a set of concepts in a domain along with their relationships, a metamodel describes what has been called a *Domain-Specific Modeling Language* (*DSML*). DSMLs are used to build models of the problems at hand, by employing only domain-specific concepts. These initial models are then successively refined through a series of model transformation steps, until reaching a final implementation. Therefore, at the core of MDE stand, on the one side, domain-specific modeling languages, and on the other, model transformation tools and techniques [76]. On the road from the initial abstract model to the final implementation, the intermediate transformations are usually model-to-model (M2M), while the last one is

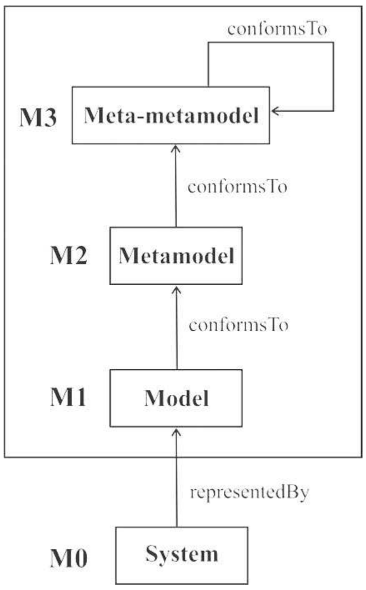


Fig. 1.2: The MDE four-layer modeling architecture ([21])

model-to-text (M2T). The output of this last transformation is thus a full-fledged software system, specified in a programming language.

The field of model-driven approaches has been pioneered by OMG's Model Driven Architecture (MDA [64]). MDA has been introduced at the end of 2000, beginning of 2001, being based on OMG standards solely: MOF, UML, OCL, CWM, QVT (Query/View/Transformation [71]), XMI (XML Metadata Interchange [72]). As stated in [64], "MDA is an approach to system development ... [that] ... provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification". The primary aims have been those of detaching a system's model from the technologies and platforms used in its detailed design and implementation, as well as automating the code generation process.

The development process envisioned by MDA starts from a *platform-independent model* of the system (*PIM*), which is further transformed (in one or several steps) into a model specific to the platform on which the system will be implemented (*platform-specific model* or *PSM*) and then into code. These transformations are assumed to be described by means of the QVT standard. Regarding the modeling language used to represent the PIM, this can be either UML (a general-purpose modeling language) or a domain-specific language such as CWM. The UML-based MDA equivalent of the four-layer modeling architecture is shown in Figure 1.3. As illustrated there, the MDA meta-metamodel is represented by MOF.

Starting with 2003-2004, the MDA philosophy has been extended by the academic community, jointly with some of the most influential actors in the industry (e.g. IBM, Microsoft, Borland). The aim has been that of integrating formalisms and technologies independent of the OMG standards. All proposals made in this respect are commonly gathered under

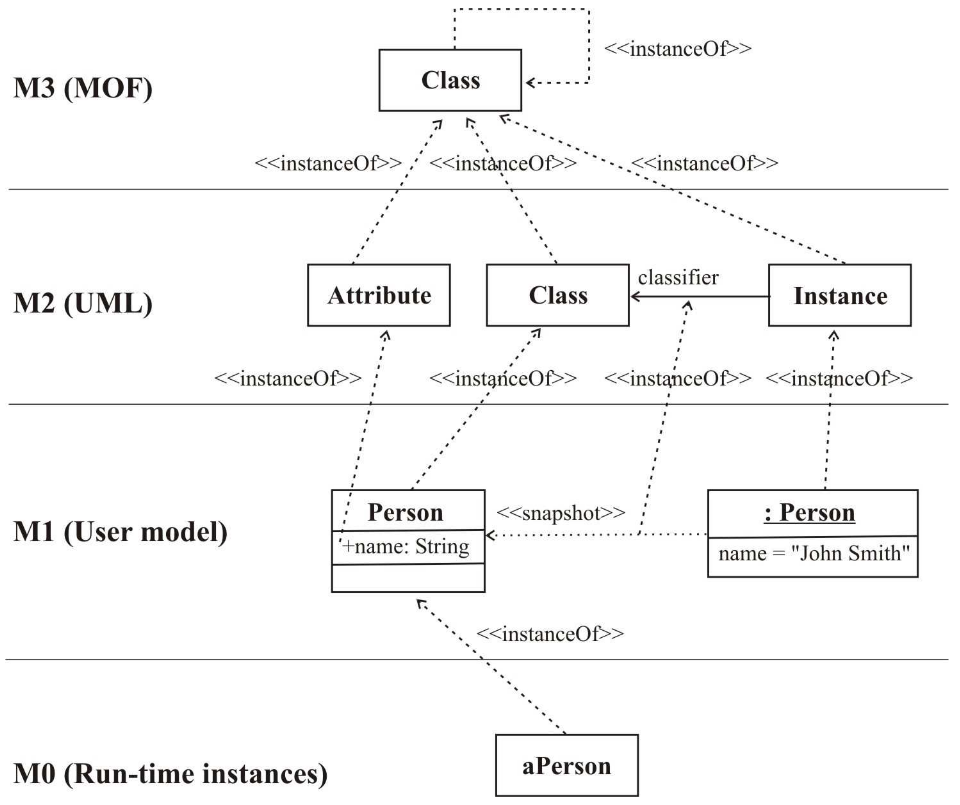


Fig. 1.3: The MDA four-layer modeling architecture ([21])

the name of MDE. The best known and used MDE framework is EMF (Eclipse Modeling Framework [40]), whose meta-metamodel, Ecore, is an implementation of a core subset of MOF named EMOF (Essential MOF). EMF also integrates with several other MDE tools or toolchains, such as GMF (Graphical Modeling Framework [42]), MDT OCL (Model Development Tools OCL [43]) or oAW (openArchitectureWare [11]).

In 2004, the authors of [33] have considered an even broader framework, including not only modeling languages, but also programming ones. The approach, which promotes executable modeling, has been called LDD (Language Driven Development). The LDD modeling architecture is rooted in the XCore meta-metamodel.

Chapter 2
Object Constraint Language (OCL)

OCL (*Object Constraint Language* [68, 85]) is a formal language used to define expressions on UML (Unified Modeling Language [69, 70]) models. Rooted in the Syntropy method [35] and introduced initially as a business modeling language at IBM, OCL is nowadays an OMG (Object Management Group) standard, similar to UML itself, MOF (Meta Object Facility), or CWM (Common Warehouse Metamodel [63]).

OCL has arised from the necessity of covering an expressivity gap of UML; the diagrammatic nature of the later makes it unable to capture most of the constraints imposed by non-trivial software systems. Therefore, OCL is not a standalone language, but a language meant to complement UML. As a consequence, any OCL expression is only meaningful in the context of a valid UML model.

Although there are extensions and dialects of OCL which provide action-support, the language is, in its standard OMG release, a pure specification formalism. Therefore, the evaluation of an OCL expression is guaranteed to have no side-effects on the state of its associated UML model. At the level of such a model, OCL can be used for the following purposes as least:

- *model navigation* - querying the information in a model through (repeated) navigation of its association relationships using role names;
- *specification of assertions* - explicit definition of pre/post-condition and invariant assertions, as promoted by Design by Contract;
- *definitions of behavior* - body specifications for the query operations included in the model, derivation rules for existing attributes and references, as well as the definition of new operations and attributes for the model;
- specification of guards, of type invariants for stereotypes, etc.

In this book, our attention will be focused on facilities and advantages that OCL may offer in the context of the MDE paradigm. In addition, in order to support a more efficient usage of OCL specifications, we will analyze some drawbacks of existing OCL standard versions; as natural, most references concern the latest version The objectives of using OCL go beyond the purpose of accomplishing a complete and unambiguous description of the problem solution by means of models. The final target is to produce high quality software by using models. Translating model-level OCL specifications into code must be done in a natural and unequivocal manner. Moreover, the results obtained when evaluating OCL specifications on model instantiations should be equivalent to the results obtained at run time, when evaluating their corresponding code on similar configurations of objects.

2.1 Language Features

OCL is a:

- **complementary language**. From the point of view of application developers, practical OCL specifications are just those described in the context of the User Defined Types in the model. Literal OCL examples frequently used to explain the semantics of various expressions may induce the false perception that OCL could be a standalone language.
- **strongly-typed language**. Each OCL (sub)expression has a type and is subject to type conformance rules.
- **declarative language**. Evaluating constraints or querying the model does not change the state of the model.
- **first-order logic-based language**
- **language that supports main features of OOP**. OCL specifications are inherited in descendents, where they may be overwritten. Constraint redefinition complies with the rules established in Design By Contract. The language supports type-casting, including upper type-casting.

2.1.1 The OCL Type System

Discussing about the OCL type system requires considering both the language features and the need to transform models in applications. From there, comes the requirement of transforming OCL specifications in programming language specifications in a straightforward manner.

Since OCL is complementary to MOF-based modeling languages, its type system integrates with that of the complemented language. Consequently, each classifier from the MOF-based model is a valid OCL type in any OCL expression attached to the model in question. Apart of types defined in the model, the OCL type system contains other categories: OCL specific types, predefined types, collection types and two types defined in advanced programming languages: Enumeration and Tuple type.

2.1.1.1 OCL Specific Types

OclAny

`OclAny` is the supertype of all types in UML models and is an instance of the metatype `AnyType`. Features of `OclAny` are available on each object. Each class of UML user models inherits all operations defined on `OclAny`. Most of them are basic operations in object-oriented languages, such as:

- operations for testing the equality of two objects
  - `=(object2:OclAny):Boolean`
  - `<>(object2:OclAny):Boolean`
- operations inferring the objects type or state
  - `oclIsTypeOf(type:Classifier):Boolean`
  - `oclIsKindOf(type:Classifier):Boolean`
  - `oclType():Classifier`
  - `oclIsInState(statespec:OclState):Boolean`
- operations specifying type casting
  - `oclAsType(type:Classifier):T`, where T is a classifier.

Another operation, `oclIsNew():Boolean`, has been conceived to be used in postconditions, in order to support the identification of objects created after the method starts executing, while `oclIsUndefined():Boolean` and `oclIsInvalid():Boolean` are two operations returning true when the receiver object or data value is undefined/unknown or when an exception has been triggered, respectively.

OclVoid and OclInvalid

OCL supports model specification beginning with early development stages. Therefore, the language has to manage `undefined` values, a problem previously encountered in databases. All types may have an `undefined` value (`void`/`null`), representing the absence of a value or signifying that the value is unknown at the current time. In standard specifications [10], the `null` value is represented by the ε symbol and has been included from the 1.x versions. The latest versions of the OCL 2.x.y standard have introduced a new value, that any OCL type has to have, `invalid`, represented by the $\perp$ symbol. The `invalid` value may result from exceptions such as division by 0, accessing a value outside of the range of an indexed type, and so on.

`Null` is the unique value of `OclVoid` and `invalid` is the unique value of the `OclInvalid` type, that conforms to all the other types, including `OclVoid`. At its turn, `OclVoid` conforms to all the other types, excluding `OclInvalid`.

OCL 1.x.y specifications erroneously state that any OCL expression containing a `null` value should be evaluated to `null`. As mentioned in various papers, evaluating expressions containing `null` values may return accurate values, as illustrated by the following OCL expressions containing literals:

`Set{1, 2, 5, null}->size()` - returns 4;

`Bag{1, 2, 1, null}->count(1)` - returns 2;

`Bag{1, 2, 1, null, 5, null}->reject(null)` - returns `Bag{1, 2, 1, 5}`.

In [10], pag. 146, it is mentioned that ”Any property call applied on `null` results in `invalid`, except for the `oclIsUndefined()`, `oclIsInvalid()`, `=(OclAny)` and `<>(OclAny)` operations.” However, when evaluating `Bag{1, 2, 1, null, 5, null}->select(i |`

`i > 1`), there are tools (e.g. USE [13]) that return `Bag{2, 5}` and not `invalid`, as correct according to the OCL specification (since `null` cannot be compared with defined values using the `>` operator).

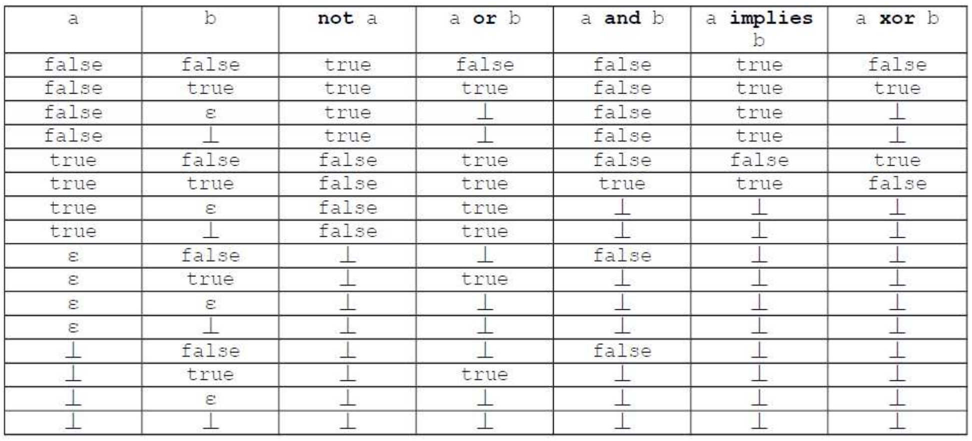
However, if we consider the previous statement saying that `OclVoid` is a descendent of all the other types, except for `OclInvalid`, this means that we may call on `OclVoid` all operations defined in parents, including comparison operations on `Integer`.

In order to manage this kind of problems, we have implemented in OCLE[1] ([52]) the `Undefined` type (the equivalent of `OclVoid`), having as unique instance `undefined` (the equivalent of `null`), as direct descendant of `OclAny`. In our implementation, the type of collections containing `undefined` values is `Collection(OclAny)`, therefore, the operations accepted on iterators of these collections are only those defined on `OclAny`.

We believe that the safe way to work with collections containing `null` values is to first reject the `null` values and only after apply other operations. Operations such as `size()` and `isEmpty()` are excepted from the above rule. Proceeding in this manner, we increase the probability of obtaining the same results when performing similar evaluations on the code corresponding to these specifications. We think that working with `invalid` values is a delicate problem that still needs to be clarified, especially concerning the management of these values in different programming languages.

2.1.1.2 OCL Predefined Types

OCL predefined types provide the bricks of the specification language and include: `Boolean`, `Integer`, `Real` and `String`. As OCL is based on first-order logic, working with `Boolean`



a	b	**not** a	a **or** b	a **and** b	a **implies** b	a **xor** b
false	false	true	false	false	true	false
false	true	true	true	false	true	true
false	ε	true	⊥	false	true	⊥
false	⊥	true	⊥	false	true	⊥
true	false	false	true	false	false	true
true	true	false	true	true	true	false
true	ε	false	true	⊥	⊥	⊥
true	⊥	false	true	⊥	⊥	⊥
ε	false	⊥	⊥	false	⊥	⊥
ε	true	⊥	true	⊥	⊥	⊥
ε	ε	⊥	⊥	⊥	⊥	⊥
ε	⊥	⊥	⊥	⊥	⊥	⊥
⊥	false	⊥	⊥	false	⊥	⊥
⊥	true	⊥	true	⊥	⊥	⊥
⊥	ε	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥

Fig. 2.1: Four-valued logic

values is extremely important. The decisions regarding `null` and `invalid` values imply that,

[1] The OCLE tool will be described in Section 2.3.

apart of classical values true and false, the Boolean type has to also manage null and invalid values. Thats why Kleene's three-valued logic was replaced with four-valued logic, as seen in Table 2.1

2.1.1.3 Collection Types

The importance of collections comes from the need to navigate the model's object graph to refer to other objects and their properties. At the class level, this is done by accessing the appropriate opposite association end. When the multiplicity of the association end is greater than 1, navigation will result in a Set. When the association end is adorned with {ordered}, the result will be an OrderedSet. Therefore, the collection types defined in the OCL Standard Library play an important role in OCL expressions.

Apart of the two above mentioned types, generic collection types also include Bag, Sequence and Collection, the latter being the common supertype of the other four collection types (Collection factors the common behavior of all the other four collection types). The hierarchy and behavior of these types has been inspired by Smalltalk Collection classes. As a consequence, OrderedSet is a direct descendent of Collection and not of Set. An interesting discussion on this topic can be found in [26]. Usually, operations applied to collections are specified by using the arrow operator (->). This is meant to stress that, in case of collections, operations are potentially applied to many receivers (collection elements). However, in case of the collect operation, an exception (enabling to replace the -> operator with the dot operator .) is made. So, the expression aCollection->collect(property) is equivalent to aCollection.property. This exception, known as "shorthand collect", is very used in practice, since it leads to slightly shorter specifications.

In order to illustrate the power of querying collections, let us consider a basic model, represented by the class diagram in Figure 2.2, and a legal instantiation of this class diagram, represented in Figure 2.3.

Usually, users interested in knowing the products ordered by customer evaluate the following OCL specification (the short for of self.orders->collect(products)).

```
context Customer
 def productsOrderedByClient:Bag(Product) = self.orders.products
```

The result of this evaluation on c1 is Bag{p1, p2, p3, p4, p5 , p6, p7}. This information says nothing about orders. Slightly more information can be obtained by using the collectNested operation, as follows.

```
context Customer
 def productsOrderedByClient_cn:Sequence(Sequence(Product)) =
  self.orders->collectNested(products)
```

In this case, the result returned by the OCL evaluator will be Sequence{ Sequence{p1, p2, p3}, Sequence{p4, p5}, Sequence{p6, p7}}. Compared with the result of the previous query, this time we know the products grouped by order, but we do not know the id or the reference of each order. Slightly detailing the previous collectNested operation will support us in obtaining the missing information.

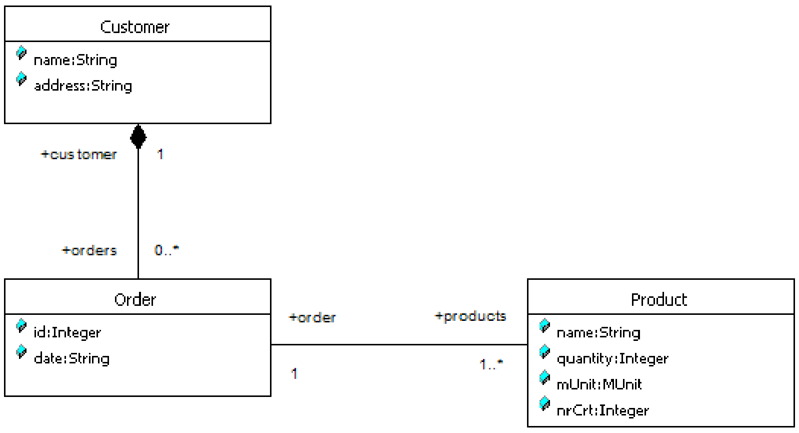


Fig. 2.2: The Customer-Order-Product model

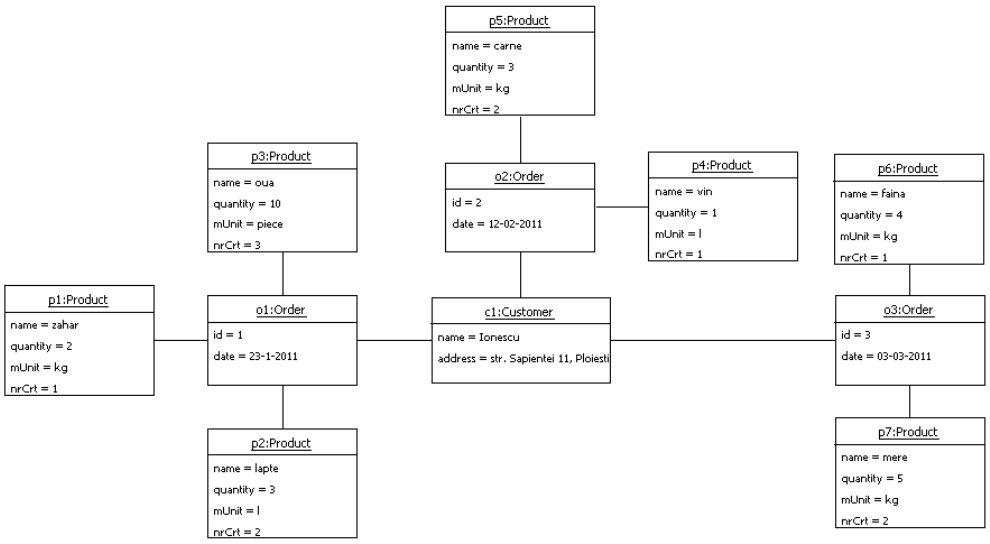


Fig. 2.3: A legal instantiation of the Customer-Order-Product model

```
context Customer
 def productsOrderedByClient_cnd:Sequence(Sequence(OclAny)) =
  self.orders->collectNested(o| Sequence{o, o.products})
```

The result returned will be `Sequence{ Sequence{o1, Sequence{p1, p2, p3}}, Sequence{o2, Sequence{p4, p5}}, Sequence{o3, Sequence{p6, p7}}}`. Moreover, if one is interested in knowing the date of each order, the query can be refined as follows.

```
context Customer
 def productsOrderedByClient_cndd:Sequence(Sequence(OclAny)) =
  self.orders->collectNested(o| Sequence{o, o.date, o.products})
```

This returns `Sequence{Sequence{o1,’23-1-2011’,Sequence{p1, p2, p3}}, Sequence{o2, ’12-02-2011’, Sequence{p4, p5}}, Sequence{o3, ’03-03-2011’, Sequence{p6, p7}}}`. Using a `TupleType`, the specification could be rewritten as:

```
context Customer
 def clientProducts_nsct:Sequence(TupleType(date:String,
                 id:Integer, ordProds:Sequence(Product))) =
  self.orders->collectNested(o| Tuple{date=o.date, id=o.id,
                                       ordProds = o.products})
```

returning `Sequence{Tuple{’23-1-2011’, 1, Sequence{p1, p2, p3}}, Tuple{’12-02-2011’, 2, Sequence{p4, p5}}, Tuple{’03-03-2011’, 3, Sequence{p6, p7}}}`. As known, each `collect` operation can be specified by using of the iterate operation. In our case, the last specification is equivalent to:

```
context Customer
 def clientProducts_nsctI:Sequence(TupleType(date:String,
     id:Integer, ordProds:Sequence(Product))) =
  self.orders->iterate(o:Order; acc: Sequence(TupleType(date:String,
    id:Integer, ordProds: Sequence(Product))) =
    oclEmpty(Sequence(TupleType(date: String, id:Integer,
    ordProds:Sequence(Product))))| acc->including(Tuple{date=o.date,
    id=o.id, ordProds = o.products }))
```

that returns the same result. In case of using the `iterate` operation, the OCL specification is more complex than its equivalent using `collectNested`. This is due to the fact that the `iterate` operation is the most generic operation on collections.

At a first glance, the examples presented above may look natural. However, the second and third `collectNested` operations are based on an architectural decision implemented in OCLE. The type of `Collection` iterators can be `OclAny` or a direct descendent of `OclAny`. In OCLE, we may work with homogeneous collections on `OclAny`. The above example shows that using `collectNested` and navigating ordered associations, we may obtain similar results to those obtained when working with tuples.

Nondeterministic Operations on Collections

As mentioned in [18],

> ”Widely neglected and often misunderstood are up to now so-called nondeterministic constructs in OCL. The most basic non-deterministic construct is the library operation `asSequence()` that expects as an argument a term of type `Set(T)` and yields a term of type `Sequence(T)`. Semantically, `asSequence()` is used to turn a set into a sequence that has the same elements as the set. The construct `asSequence()` is called non-deterministic, because it imposes a nondeterministically chosen ordering on the elements in the resulting sequence which is not given for the elements of the argument set.”

As a consequence, there are different decisions regarding this problem. Proposals range from radical positions, requiring to remove nondeterministic operations from OCL specifications [18] to the most permissive, which is to accept these nondeterministic operations. Our position stands somewhere in the middle; we propose to establish an unequivocal ordering criterion, the order in which the elements appear in the set, and continue with this.

2.1.1.4 Re-typing or Casting in OCL

UML supports object-orientation. Therefore, OCL has to support it as well. However, there are parts in the specification that do not. For example, upper cast is forbidden in [10] in paragraph 7.5.6 and accepted in paragraph 7.6.8. The following is an excerpt from [10], paragraph 7.5.6.

> In some circumstances, it is desirable to use a property of an object that is defined on a subtype of the current known type of the object. Because the property is not defined on the current known type, this results in a type conformance error. When it is certain that the actual type of the object is the subtype, the object can be re-typed using the operation `oclAsType(Classifier)`. This operation results in the same object, but the known type is the argument `Classifier`. When there is an object object of type `Type1` and `Type2` is another type, it is allowed to write:
> `object.oclAsType(Type2)` – changes the static type of the expression to `Type2`.
> An object can only be re-typed to a type to which it conforms. If the actual type of the object, at evaluation time, is not a subtype of the type to which it is re-typed, then the result of `oclAsType` is `invalid`.

Below, there is a quotation from [10], paragraph 7.6.8.

> Whenever properties are redefined within a type, the property of the supertypes can be accessed using the `oclAsType()` operation. Whenever we have a class `B` as a subtype of class `A`, and a property `p1` of both `A` and `B`, we can write:

```
context B
 inv: self.oclAsType(A).p1
 -- accesses the p1 property defined in A
 self.p1
 -- accesses the p1 property defined in B.
```

> Figure 7.4 shows an example where such a construct is needed. In this model fragment there is an ambiguity with the OCL expression on `Dependency`:

```
context Dependency
 inv: self.source <> self
```

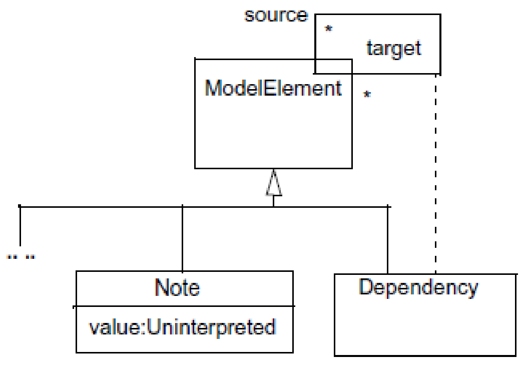


Figure 7.4 - Accessing Overridden Properties Example

This can either mean normal association navigation, which is inherited from `ModelElement`, or it might also mean navigation through the dotted line as an association class. Both possible navigations use the same role-name, so this is always ambiguous. Using `oclAsType()` we can distinguish between them with:

```
context Dependency
   inv: self.oclAsType(Dependency).source->isEmpty()
   inv: self.oclAsType(ModelElement).source->isEmpty()
```

2.1.2 *Properties and Navigation*

Each OCL expression is written in the context of an instance of a particular type. Within the expression in question, the contextual instance may be referenced using the `self` keyword (the keyword may be also omitted, when there is no risk of ambiguity). Starting from the contextual instance, it is possible to access any of its attributes, query operations/methods, or opposite association ends, using an object-oriented dot-style notation. As a first example,

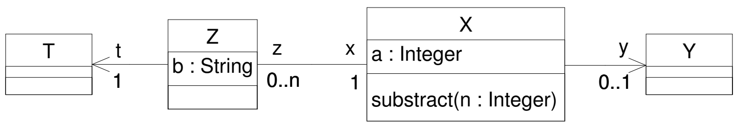


Fig. 2.4: Sample UML model

`self.a` and `self.y` are two OCL expressions, of type `Integer` and `Y` respectively, written in the context of class `X` from Figure 2.4. The second expression involves a navigation of the association between `X` and `Y`, using the opposite association end `y`. When the opposite association end has multiplicity at most one, the type of the resulting expression is given by the classifier at that end[2]. When the multiplicity is greater than one, single navigation results in a `Set` or `OrderedSet`, depending on whether the opposite end is adorned or not with an {`ordered`} constraint. Therefore, the type of the OCL expression `self.z`, written in the context of `X`, is `Set(Z)`. Combined navigation results in a `Bag`, from where is follows that the type of the OCL expression `self.z.t` is `Bag(T)`.

In addition to accessing properties of the contextual instance, it is possible, within an OCL expression, to call the `allInstances()` operation on a specific classifier. This results in the set of all currently instantiated objects having that classifier as type. Thus, the OCL expression `X.allInstances()->select(x | x.a > 10)` can be used to find out the set of all instances of `X` whose values of `a` exceed 10.

2.1.3 Design by Contract in OCL

The primary use of OCL expressions in an UML model is in the body of assertions - pre/post-conditions and invariants.

An OCL invariant is stated in the context of a classifier, which gives the type of its contextual instance. The name of the classifier is introduced within a `context` clause. This is followed by the `inv` keyword, together with an optional identifier, and the OCL expression itself. Below, we provide a basic invariant requiring that the values of the `X::a` attribute should be always positive.

```
context X
 inv invX1: self.a >= 0
```

In case of pre/post-conditions, the `context` clause indicates the signature of the operation for which the assertion is written. The contextual instance `self` is then an instance of the type that owns the operation in question. In a postcondition, the `@pre` notation can be used to designate the value of an object/property at the start of the operation. We exemplify this in the following, by providing the contract corresponding to a potential `substract()` operation for class `X`, designed so as to enforce preservation of the `invX1` invariant written in the context of this class.

```
context X::substract(n:Integer)
 pre substractPre:   self.a >= n
 post substractPost: self.a = self.a@pre - n
```

OCL includes a mechanism for structuring specifications. When a subexpression appears several times in a constraint, it is possible to extract it in a local variable, by means of a `let` statement. This brings an improvement with respect to both the understandability of the whole constraint and its efficiency, by avoiding the repeated computation of the same

[2] The result of a navigation over an association with multiplicity `0..1` can also be used as a `Set` (having at most one element). This is particularly useful when there is the need to check whether there exists or not an object at the other end of the association; this can be done by using the `isEmpty()` operation on sets.

subexpression. As an example, the invariant below uses a `let` statement for constraining the result of a combined navigation to evaluate to a set, rather than a general bag.

```
context X
 inv invX2: let allT:Bag(T) = self.z.t in
            allT->size() = allT->asSet()->size()
```

2.1.4 Definition Constraints

The `let` construct introduced previously enables the definition and reuse of local variables within a particular OCL expression. To allow the reuse of expressions over multiple constraints, OCL provides a “definition” type of constraint. Such a constraint, identified by the `def` keyword, may be attached to a classifier and consists in the definition of a helper attribute or operation for that classifier. An attribute or operation defined this way behaves identically to a normal attribute or operation of the classifier, being known in the same context as any of the latter. As an example, the listing below illustrates the definition of a helper attribute and operation for `X`. The attribute, `hasY`, checks by means of the `oclIsUndefined()` operation whether there is or not an object of type `Y` currently attached to the contextual instance; the operation, `hasZWithValue()` verifies the existence of an instance having a particular value for its `b` attribute, within the set of `Z` objects attached to the contextual instance.

```
context X
 def: hasY:Boolean = not self.y.oclIsUndefined()
 def: hasZWithBValue(value:String):Boolean =
       self.z->exists(zz | zz.b = value)
```

2.2 OCL in Metamodeling. OCL Dialects

Even from the first version of the OMG UML Specification (1.1), OCL has been used to define constraints on the UML metamodel itself. Such constraints were needed for ensuring an accurate specification of the modeling concepts provided by the language. In the OMG documents, these constraints appear under the name of *well-formedness rules*, being formalized as OCL invariants that are accompanied by natural language descriptions. Quite often, the specification of well-formedness rules makes use of *additional* (query) *operations*, which are themselves formalized as OCL expressions.

Although initially developed as a formal specification language for UML, following the advent and maturation of the MDA (Model Driven Architecture) and MOF OMG standards, OCL can now be used in conjunction with any MOF-based metamodel (e.g. MOF itself, UML, CWM). Moreover, the success enjoyed by the model-driven initiative has triggered the emergence of several concrete approaches in this area, each relying on either the standard OMG OCL or an OCL dialect[3].

EMF (Eclipse Modeling Framework [40, 78]) for instance, is a modeling framework and code generation facility that has started as an implementation of MOF. In fact, it may be regarded as a highly efficient Java implementation of a core part of MOF, called EMOF (Essential MOF). The EMF meta-metamodel is named Ecore. EMF provides standard OCL support by means of the MDT (Model Development Tools [43]) OCL project.

Apart from MDT OCL, there are various other MDE tools or toolchains built around or integrated with the EMF framework (see [41] for a comprehensive list). Among them, oAW (openArchitectureWare [11, 81]) distinguishes itself by its flexible code generation technology, as well as by the workflow-based approach that it proposes for the management of operations on models. For the various model-related tasks (definition of constraints, extensions or model transformations), the oAW framework relies on three textual languages (Check, Xtend and Xpand), that are built over a common expression language and type system. The metamodel-level constraints are defined using Check - a declarative constraint language with an OCL-similar syntax, to which it adds the facility of defining custom error or warning messages to be displayed in case of constraint violation. All the required metamodel extensions (such as additional properties) are stated in a non-invasive way, by means of the Xtend language.

XMF (eXecutable Metamodeling Facility [3]) is the metamodeling facility standing at the core of the LDD (Language Driven Development [33, 34]) approach. XMF consists of a number of languages which enable it to cope with all aspects of a language definition, with extra-support for executability and metamodel mappings (XCore, XOCL, XBNF, XMap, and XSync). XOCL (eXecutable OCL) is an executable metamodeling language built on top of OCL, to which it adds a number of action primitives. The main syntactic differences compared to the standard OCL consist in the use of a different syntax for invariants and queries: invariants are introduced by the `@Constraint` keyword (instead of the standard `inv`), while queries are defined as `@Operations` (instead of using the standard `def`). The action primitives provided are: assignment (by means of the "`:=`" operator, as in `self.a := self.a + 1`), object creation (by means of a constructor call, as in `z := Z()`) and sequencing (by means of the "`;`" delimiter, as in `self.a := 1; self.a := self.a + 1`). In addition, the language provides useful executable extensions, such as the `@For` and `@While` constructs for looping. Other differences to the OMG standard include a `sel` operation used to select a single element from a collection, the `of()` reflective operation which returns the class from which the current object is instantiated, as well as a `fail` clause enabling the specification of an action to be performed in case of constraint violation.

Irrespective of the approach and dialect involved, the main use of OCL in metamodeling consists in the definition of well-formedness rules and additional operations. For the purpose of this book, we rely on the following definitions.

Definition 2.1 (*Well-Formedness Rule*). A *Well-Formedness Rule* (*WFR*) is a constraint imposed on a metaclass or a group of metaclasses. Such a constraint is generally formalized as a

[3] The phrases *OCL dialect* and *OCL-like language* are used interchangeably in this book to designate a formal language rooted in the OMG OCL, but exposing syntactic differences and/or extensions when compared to the standard.

metamodel-level invariant, using either OCL or the OCL dialect attached to the metamodeling language in which the metamodel is defined.

Definition 2.2 (*Additional Operation*). An *Additional Operation* (*AO*) is an extra query operation needed on a metamodel. AOs are generally defined with the purpose of simplifying the writing of WFRs. In standard OCL, these are stated as definition constraints.

2.3 Tool Support. OCLE

There are various OCL-supporting tools available at the moment, emerging from both industry and academia. Among the industrial tools, Borland Together [1] is definitely the most relevant with respect to the functionalities offered. In the field of the academic ones, OCLE (OCL Environment [52]), USE (UML-based Specification Environment [13]), Dresden OCL Toolkit [2] and MDT OCL [43] are some of the main representatives. Since most of the proposals reported throughout this book have been validated using OCLE, we provide below a brief overview of the assertion evaluation facilities offered by this tool.

OCLE is a CASE (Computer Aided Software Engineering) tool that allows the specification of OCL assertions at two abstraction levels (metamodel-level and user model-level). Assertions are stored in ascii files whose extensions denote the abstraction level employed: ".ocl" for metamodel-level assertions (WFRs) and ".bcr" for user model-level assertions (Business Constraint Rules or BCRs). Once compiled, these assertions can be evaluated using any of the following three model validation approaches:

1. Validation of the entire model, with respect to all specified constraints. Each object is validated against all constraints specified for its class and its ancestors;
2. Validation of all instances of one or several classes belonging to a given package, with respect to either all constraints specified for those classes or a specific subset;
3. Validation of a particular object, with respect to a particular constraint.

Information regarding the errors identified during the validation of a model or of a set of objects is exposed in a tree-like manner: each broken constraint is represented by a node having as a direct ancestor its context class and as direct descendants rule failure messages pointing at the "responsible" instances. Starting from such an error message, the user can access/edit:

- the assertion (constraint) whose assessment has failed,
- the diagram pointing out the bogus instance (which is automatically set as the contextual instance),
- the model excerpt containing the base class of the problematic instance.

By means of the textual editor, the user can then evaluate any of the constraint's subexpressions on the contextual instance, with the aim of identifying the exact failure reasons.

Chapter 3
Constraint Patterns in Object-Oriented Modeling

Similar to design patterns, *constraint patterns* embody a high level of expertise concerning the specification of recurrent constraints occurring in object-oriented class models. They have primarily emerged from the reality that the editing of constraints on sizable models is a tedious, time-consuming and error-prone task. Therefore, the definition of constraint patterns and, most of all, their automatic application, has been seen as a means of speeding the writing task and increasing its correctness. And, indeed, the constraint patterns identified so far in the literature, with their current solutions, have been reported to fulfill well this initial target. However, the state of the art paradigm of software engineering, MDE, places new requirements on model assertions, triggering the necessity of a new approach concerning the solutions given to constraint patterns.

This chapter sums up our research with respect to the use of constraint patterns in object-oriented class models. The main contribution reported here consists in the proposal of a new approach concerning the definition of these patterns, driven by the new requirements imposed by MDE on the use of assertions.

The remaining of this chapter is organized as follows. Section 3.1 details our motivation in approaching the field, by emphasizing the new purpose of assertions in model-driven development. Section 3.2 gives a brief overview of the most relevant work concerned with the definition of constraint patterns. Our approach is described in Section 3.3: the approached patterns are introduced in Subsection 3.3.1, together with their current solutions, which serve us as a comparison base; the core of our proposal is given in Subsection 3.3.2, followed by its validation in Subsection 3.3.3. Section 3.4 summarizes our contribution in this field and indicates directions of future work.

3.1 Motivation

The long-dreamed goal of software engineering, software development automation, is nowadays promised to be fulfilled by the MDE approach. However, this recently emerged paradigm can only deliver its promises when provided with means of developing detailed and rigorous models. In turn, the creation of such models requires graphical modeling languages to be complemented by appropriate assertion languages (such as OCL for the MOF-based family of modeling languages). We have previously discussed about the overall purpose and value of using assertions in software specification in Chapter 1 of this book. Following, we argue that the emergence of MDE has triggered a major shift with respect to the usage of models and assertions in software development.

Traditionally, modeling has been primarily targeted at facilitating problem understanding, assisting the client-developer communication, and guiding a mostly-manual implementation process. In this context, the purpose of writing model assertions (pre/post-conditions and invariants) was threefold. Firstly, they were meant to help in writing correct programs[1], their explicit definition being regarded as a precondition of their enforcement in software. Secondly, they were the "oracles" against which to mentally assess software correctness during program testing. Thirdly, they were used as documentation artifacts, along with the complemented models.

With the advent of MDE however, models are upgraded from "helpers" to "first class citizens" of software development. Consequently, the emphasis is now laid on the creation of comprehensive correct models, intended to be automatically turned into code. Moreover, models are used at various abstraction levels (user-model, metamodel, meta-metamodel), reflected in metamodeling architectures. These new requirements of MDE place the use of assertions into a new perspective. Ensuring model correctness requires model compilability checks and model testing; the former activity is based on the automatic evaluation of metamodel-level assertions (called Well-Formedness Rules or WFRs), while the later involves the automatic evaluation of user model-level assertions (called Bussines Constraint Rules or BCRs). Moreover, following a natural MDE process, the model-level assertions should be translated into program-level routines, along with the model itself. This provides for runtime assertion monitoring, enabling the automatic use of assertions in program testing (as opposed to the traditional mental reasoning), as well as an aid to debugging and the creation of fault tolerant systems[2].

We claim that these new means of using assertions should trigger a change with respect to the style in which assertion specification is done. While complying with the role of assertions in traditional software development requires them to be specified in the shortest, most intuitive manner, conformance to the requirements of MDE demands specifications to facilitate efficient error detection and diagnosis.

The widespread use of constraints in the context of MDE has naturally led to the identification of several constraint patterns. However, their currently available solutions in the literature do not seem to reflect the new purpose of assertions in the context of model-driven development. Moreover, there are cases when not even the problem itself is identified and stated properly. Starting from this state of facts, our aim in this chapter is that of contributing to the field with a new approach (one that is MDE-driven) concerning the definition of constraint patterns for object-oriented class models.

[1] A comprehensive definition of software correctness with respect to assertions may be found in [59].

[2] A comprehensive talk concerning the advantages of runtime assertion monitoring is given in [59].

3.2 Related Work

The concept of *constraint pattern* has been introduced into the field of object-oriented modeling by [19], in relation with object-oriented design patterns. The paper in question proposes a mechanism for associating OCL constraint patterns to design patterns, allowing instances of the former to be created at each instantiation of the latter. Driven by the aim of reducing the time, as well as the amount of errors occurring in the specification of OCL constraints on UML models, several papers have further approached the constraint patterns topic, by proposing specification methods and libraries of such patterns ([60], [36], [15, 16], [84, 82, 83]). Since throughout this chapter we will refer to the latter two approaches for comparison, we provide a brief overview of them in the following.

The work reported in [16, 15] is focused on the identification and specification of constraint patterns occurring in the UML behavioral description of software components. These patterns are referred as *OCL specification patterns*. The author of [16] identifies a set of nine such patterns and proposes a description scheme that enables a structured, uniform presentation of all pattern-related information. The following features are specified as part of the scheme for each pattern: name, list of parameters (typed by elements from the UML metamodel), parameter-related restrictions, type of the proposed constraint (invariant, pre or post-condition), context of the constraint (one of the parameters), as well as corresponding text (generic OCL expression depending on the specified parameters). Based on the information covered by this scheme, in [15], a formal pattern specification approach is proposed. This involves associating to each pattern a homonymous metamodel-level function, which takes the pattern parameters as arguments and returns an instance of the `Constraint` metaclass. The function in question is provided with an OCL pre/post-condition specification, which formalizes the information contained in the description scheme. The implementations of such functions are used as constraint generators in a prototype tool supporting the proposed approach.

The paper [84] and the thesis [82] offer a comprehensive, pattern-based solution to the problem of consistent design constraints' specification. The solution is provided as an MDE process consisting of a method and an associated tool that assists in the development of consistent constraint specifications. At the core of the proposed approach stand constraint patterns, defined as "parameterizable constraint expressions that can be instantiated to solve classes of specification problems" [82]. [84] introduces the philosophy of *model-driven constraint engineering*, in which instances of constraint patterns (playing the role of computation-independent models or CIMs) undergo model transformation steps that convert them into literal OCL constraints (platform-independent models or PIMs) or implementations of such constraints in particular programming languages (platform-specific models or PSMs). The paper also proposes a taxonomy of constraint patterns, differentiating among atomic/elementary and composite ones. Atomic patterns abstract basic constraints on a model, while composite patterns allow expressing arbitrarily complex restrictions, by integrating any number of constraints, either atomic or composed. In addition, the authors come with an extensible library of atomic constraint patterns. The semantics of elementary patterns is intuitively formalized in terms of parameterized OCL templates, enhanced by meta-constraints restricting the allowed parameter values. However, this OCL template approach can only approximate the semantics of composite patterns, due to the need of quantifying over arbitrary sets of predicates, which is a concept of higher-order logic and thus not covered by OCL. Nevertheless, this inconvenient is solved in [82], in which HOL-OCL [23] functions are used as a means to uniformly define the semantics of both elementary and composite constraint patterns.

3.3 A New Approach: MDE-Driven OCL Specification Patterns

Let us now remind the target of our work, as motivated earlier in this chapter. A review of the literature has shown us that the majority of OCL specifications accompanying class models (including the existing solutions to constraint patterns) tend to focus exclusively on the clearness of expressions. This is a requirement that is rather associated to the role of assertions in traditional modeling, as previously pointed out. In the context of MDE however, which lays emphasis on model correctness and automation, there are other requirements prevailing. Following the automatic verification of models and applications' correctness, the mere information that a system state is inconsistent or that a method pre/post-condition is not fulfilled is insufficient. Being able to identify the exact failure reasons is of utmost importance for error correction. This is the core idea governing the OCL specification approach that we promote. In accordance to it, we aim at providing new solutions for some of the existing constraint patterns. For ease of reference and comparison, we describe the patterns in question within the next section, together with their currently available solutions in the literature.

3.3.1 Approached Patterns. Existing Solutions

Let us first make clear the terminology employed throughout the remaining of this chapter. Being more recently dated, thus less known and used compared to their design counterparts, constraint patterns do not currently benefit from a generally-agreed naming or definition in the literature. For instance, they are referred as *OCL specification patterns* in [15] and as *constraint patterns* in [84, 82]. In addition, the general definition proposed in [82] and quoted by us in the previous section is rather directed towards the solution part of the pattern.

To take advantage of a commonly-established pattern intuition, we propose reasoning about constraint patterns by analogy to design patterns. From this perspective, a constraint pattern refers to both a constraint problem and its associated recommended solution.

Definition 3.1. A *constraint pattern* embodies a recurrent logical restriction imposed on class models, together with a recommended general specification for it.

The solution of an object-oriented design pattern is illustrated using the syntax of a particular object-oriented modeling language (e.g. UML, OMT). Analogously, the solution of a constraint pattern employs the syntax of the assertion language attached to the modeling language in which the constrained model is represented (e.g. OCL for the MOF family of languages and Ecore, or XOCL for XMF). Throughout this chapter, we will be working with MOF-based class models and OCL assertions. In this context, the OCL-based solutions given to constraint patterns will be referred as *OCL specification patterns* .

Definition 3.2. An *OCL specification pattern* denotes the recommended OCL-based solution of a constraint pattern.

We will be referring to three constraint patterns identified in the literature - *Attribute Value Restriction*, *Unique Identifier*, and *For All*. In the pattern taxonomy proposed in [82], the first two are classified as elementary constraint patterns, while the last one appears as a composite pattern. *Attribute Value Restriction* [82] (called *Invariant for Attribute Value* in [16]) is a basic pattern used to abstract various constraints on the value of a given class attribute. *Unique Identifier* [82] captures the situation in which an attribute (a group of attributes) of a class plays the role of an identifier for the class, i.e. the class' instances should differ in their value for that attribute (group). This is probably the best known constraint pattern, being referred under different names in the literature - *Semantic Key* in [16], *Primary Identifier* in [60], or simply *Identifier* in [36]. Finally, the *For All* constraint pattern [82] requires every object of a certain collection to fulfill a number of specified restrictions.

In the following, we present existing OCL specification patterns for the three mentioned constraint patterns. In order to keep the description consistent and intelligible, all[3] OCL specification patterns are given in terms of OCL parameterized templates, as described in [82]. Using this representation, each OCL specification pattern is a template (macro) that depends on a number of parameters, which are typed by elements from the UML/OCL metamodels. Replacing them with actual model elements generates a pattern instantiation. Moreover, to maintain consistency and allow an easy comparison of our work with existing related approaches, we will use the same representation means (OCL templates) when introducing our own solution proposals.

In [82], the following template representation is given for the OCL specification pattern associated to *Attribute Value Restriction*.

```
pattern AttributeValueRestriction(property:Property,
                                  operator,value:OclExpression)=
 self.property operator value
```

This template depends on three parameters (given in italics), namely *property* - which stands for the attribute that is to be constrained, *operator*, and *value* - which are used to restrict the attribute's value. Such a pattern may be instantiated to generate an invariant in the context of an UML class, by providing as actual parameters an attribute of the class, a concrete operator and a value expression.

Following, there is the OCL template for *Unique Identifier*, as proposed in [82]. The template employs one parameter, *property*, standing as a placeholder for any tuple of class attributes on which we want to impose the uniqueness constraint.

```
pattern UniqueIdentifier(property:Tuple(Property))=
 self.allInstances()->isUnique(property)
```

The specification given in [15] for the same pattern matches the following OCL template.

[3] As emphasized by [82], the *For All* pattern solution cannot be represented by a "pure" OCL template, due to its inherent higher-order nature. For intelligibility sake however, we provide and work with a reasonable OCL-template approximation.

```
pattern SemanticKey(class:Class, property:Property)=
 class.allInstances()->forAll(i1, i2 | i1 <> i2 implies
                              i1.property <> i2.property)
```

The template uses two parameters, *class* and *property*, the first standing for the class on whose instances we want to impose the uniqueness constraint, and the second for the attribute whose values should be unique. Both `UniqueIdentifier` and `SemanticKey` are intended to be reused by instantiation as class invariants.

Finally, we give below the OCL template-like description for the specification pattern associated to the *For All* constraint pattern.

```
pattern ForAll(collection:OclExpression,
               properties:Set(OclExpression))=
 collection->forAll(y | oclAND(properties, y))
```

The template-like description above uses two parameters. The first one, *collection*, denotes the collection of model elements which is iterated. The second parameter, *properties*, stands for the set of constraints that should be fulfilled by all objects in the collection. Each such constraint may be, in fact, an instantiation of some constraint pattern. Since the OCL standard does not include a construct for expressing the conjunction of an arbitrary number of boolean expressions, we introduce the *oclAND* notation in this purpose. Consequently, *oclAND*(*properties*, *y*) denotes the expression resulting from the conjunction of the boolean expressions obtained by replacing `self` with `y` in each of the constraints from *properties*.

3.3.2 Proposed Solutions

3.3.2.1 The *For All* Constraint Pattern

Let us consider the UML model in Figure 3.1, together with a business constraint rule stating that “All employees of a company should be aged at most 65”. We will refer to this particular constraint as *Complying with Retirement Age Limit (CRAL)* in the following.

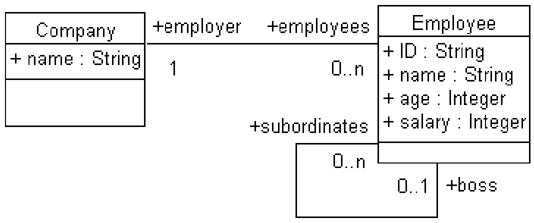


Fig. 3.1: A sample class model

Most of the OCL specifications encountered in the literature for such a constraint consist of an invariant of the following shape.

```
context Company inv CRAL_E:
 self.employees->forAll(emp:Employee | emp.age <= 65)
```

The specification above is quite compact and intelligible, due to the fact that the OCL `forAll()` operation directly corresponds to the mathematical logic's universal quantifier (∀). However, such an invariant shape is not the most useful in a testing/debugging-related context. The feedback it offers in case of failure is tool-dependent, consisting in either a simple `false` message, or an implementation-dependent tree showing the result of evaluating the age constraint on each employee of the company. In the former case, we have no useful hint regarding the identity of those employees which are over the age limit. In the latter, the task of searching within the employees tree may be disturbing and time-consuming. This turns the debugging of an object model containing a large number of employees into a difficult task.

As an answer to the above problem, we suggest using any of the following two specifications. These are convenient not only at modeling-time, but also at run-time, allowing a user to easily get the information he needs in fixing a potential error.

```
context Company inv CRAL_P1:
 self.employees->reject(emp:Employee | emp.age <= 65)->isEmpty()

context Company inv CRAL_P2:
 self.employees->select(emp:Employee | not(emp.age <= 65))->isEmpty()
```

`CRAL_E`, `CRAL_P1` and `CRAL_P2` are semantically equivalent. However, the newly proposed invariants have the advantage of allowing evaluations of the `reject()`/`select()` subexpressions, which return exactly the set of employees violating the age constraint. Evidence of this will be provided in Subsection 3.3.3, using the OCLE tool.

A basic reasoning on the kind of restriction imposed by *CRAL* leads to the conclusion that this particular constraint is, in fact, an instantiation of the *For All* composite constraint pattern described in Subsection 3.3.1. The constraint used by the composite is, at its turn, an instantiation of an atomic constraint pattern, namely *Attribute Value Restriction*. Furthermore, the OCL specification employed by the `CRAL_E` invariant (a widely-encountered style in the literature) is an instantiation of the `ForAll` specification pattern, as proposed in [82] (see Subsection 3.3.1). The latter uses an instantiation of the `AttributeValueRestriction` specification pattern proposed by the same reference. In light of these statements, the `CRAL_E` invariant writes as follows.

```
context Company inv CRAL_E:
 ForAll(self.employees, Set{AttributeValueRestriction(age,<=,65)})
```

Above, we have emphasized the benefits gained by replacing the `CRAL_E` specification with its `CRAL_P` counterparts. In order to be able to exploit those benefits in case of all constraints instantiating the *For All* composite pattern, we propose the following two equivalent OCL specification patterns for it.

```
pattern ForAll_Reject(collection:OclExpression,
                      properties:Set(OclExpression))=
 collection->reject(y | oclAND(properties, y))->isEmpty()
```

```
pattern ForAll_Select(collection:OclExpression,
                      properties:Set(OclExpression))=
 collection->select(y | not oclAND(properties, y))->isEmpty()
```

There are two necessary conditions (meta-constraints) which need to be fulfilled in order for the patterns above to generate syntactically correct OCL specifications by instantiation. Namely:
(FA1) *collection* should be a valid OCL expression which evaluates to a collection type;
(FA2) each of the *properties* should be a valid boolean OCL expression.

In this context, it is obvious that the `CRAL_P1` and `CRAL_P2` specifications are instantiations of the `ForAll_Reject` and `ForAll_Select` specification patterns respectively, as shown below.

```
context Company inv CRAL_P1:
 ForAll_Reject(self.employees,
              Set{AttributeValueRestriction(age,<=,65)})

context Company inv CRAL_P2:
 ForAll_Select(self.employees,
              Set{AttributeValueRestriction(age,<=,65)})
```

As previously stated, we claim that the use of the newly proposed OCL specification patterns (`ForAll_Reject` and `ForAll_Select`) has a significant impact on the efficiency of testing/debugging activities, by increasing it in case of large collections of objects.

3.3.2.2 The *Unique Identifier* Constraint Pattern - Invariant Solutions

There are basically two possible contexts for applying the *Unique Identifier* constraint pattern, although no emphasis has been put on distinguishing among them in the literature. One of them refers to the so-called “global” uniqueness - certain models or applications may require all possible instances of a class to differ in their value for a particular attribute. The other captures a “container-relative” uniqueness - a model/application constraint may state that each instance of a class accessible starting from a given “container”[4] should be uniquely identifiable by the value of a particular attribute, among all instances of the same class from within the same container. The existing OCL specification patterns for the *Unique Identifier* constraint pattern (those reproduced in Subsection 3.3.1) concern the “global” uniqueness case exclusively. Therefore, they should not be used for the “container-relative” one, as generally encountered in the literature. Moreover, the existing specification patterns have drawbacks, even when used in the appropriate context. In the following, we analyze both uniqueness contexts and propose appropriate OCL specification patterns for each.

“Global” uniqueness case (GUID): Let us consider a census application whose model contains a `Person` class. Suppose this class has an `ID` attribute and there is a business constraint stating that persons should be uniquely identifiable by their `ID`s. This is a classical case of a “global” uniqueness constraint.

[4] We use quotes since we only require a navigable association relationship, not necessarily a composition.

For such a constraint, the majority of OCL specification proposals in the literature have one of the following shapes.

```
context Person inv GUID_E1:
 Person.allInstances()->forAll(p, q | p <> q implies p.ID <> q.ID)

context Person inv GUID_E2:
 Person.allInstances()->isUnique(ID)
```

It may be easily noticed that these are instantiations of the `SemanticKey` and `UniqueIdentifier`[5] specification patterns, as reproduced in Subsection 3.3.1. Therefore, they can be written as:

```
context Person inv GUID_E1: SemanticKey(Person, ID)

context Person inv GUID_E2: UniqueIdentifier(Tuple{x=ID})
```

As given above, these specifications have two drawbacks, which have been confirmed by the tests that we have performed using OCLE.

1. The worst is that `GUID_E1` breaks the semantics of invariants, as promoted by Design by Contract. To acknowledge this, consider the case when at least two persons have the same `ID` value. Then, the evaluation of this invariant would return `false` for ALL persons (even for those having an unique `ID`). According to the semantics of an invariant, this should evaluate to `false` only for those instances violating the constraint that it formalizes. Such a semantics is clearly disregarded by `GUID_E1` invariant. This drawback is triggered by the use of `allInstances()` uncorrelated with the contextual instance.
2. Due to the use of `forAll()` and `isUnique()` respectively, the two specifications do not provide appropriate debugging support.

Since we have shown that the two specifications are, in fact, instantiations of the OCL specification patterns `UniqueIdentifier` and `SemanticKey`, it logically follows that the OCL patterns themselves are inappropriate. As a replacement of `GUID_E1` and `GUID_E2`, we propose the following invariant

```
context Person inv GUID_P:
 Person.allInstances()->select(p | p.ID = self.ID)->size()=1
```

from which we deduce the general OCL specification pattern that should be applied in case of “global” uniqueness constraints.

```
pattern inv_GloballyUniqueIdentifier(class:Class,attribute:Property)=
 class.allInstances()->select(i |
   i.attribute = self.attribute)->size() = 1
```

[5] The latter is rather an instantiation of the “specification intent” expressed by the `UniqueIdentifier` specification pattern. This is due to the fact that this pattern, as proposed by its authors and reproduced by us in Subsection 3.3.1, is mistaken, since it generates syntactically incorrect OCL code by instantiation. In order for it to generate compilable OCL code, it should be written using two parameters (not just one), similarly to its `SemanticKey` equivalent.

The pattern above is meant to be instantiated as an invariant in the context of *class*. The necessary conditions for ensuring syntactical correctness of the resulting OCL expression are thus the following:
(invGUID1) the pattern instantiation should be performed in the context of *class*;
(invGUID2) *attribute* should be among the attributes of *class*.

It is obvious that `GUID_P` is an instantiation of this pattern.

```
context Person inv GUID_P: inv_GloballyUniqueIdentifier(Person,ID)
```

The specification pattern that we have introduced above is both correct with respect to the semantics of invariants and useful from a debugging perspective. However, as stated by the UML 1.5 Specification [65] (pp. 6-19):

The use of allInstances has some problems and its use is discouraged in most cases. The first problem is best explained by looking at the types like Integer, Real and String. For these types the meaning of allInstances is undefined. ... The second problem with allInstances is that the existence of objects must be considered within some overall context, like a system or a model. ... ***A recommended style is to model the overall contextual system explicitly as an object within the system and navigate from that object to its containing instances without using allInstances.***

“Container-relative” uniqueness case (CUID): In accordance with the emphasized phrase from the above quotation, we claim that this is how the majority of uniqueness constraints should be imposed. Let us start from the model in Figure 3.1 and suppose there is a constraint requiring that employees of a company should be uniquely identified by their `ID`s. This is a classical case of an uniqueness requirement in the context of a “container”, the “container” being represented by a `Company` object.

A correct and useful specification for the considered constraint can be given in the context of the `Company`, as follows.

```
context Company inv CUID_P1:
 self.employees->reject(e |
  self.employees.ID->count(e.ID) = 1)->isEmpty()
```

This has a minor efficiency issue however, due to the repeated computation of the employees' id collection. In order to avoid it, we propose to isolate this computation by means of an OCL `let` statement.

```
context Company inv CUID_P2:
 let allIDs:Bag(String) = self.employees.ID in
 self.employees->reject(e | allIDs->count(e.ID) = 1)->isEmpty()
```

An analysis of `CUID_P2` points to a potentially new atomic constraint pattern requiring that “A given class attribute has exactly one occurrence in a given bag of elements of the same type”. We use the phrase *Unique Occurrence in Bag*, with the acronym *UOB*, to denote the newly discovered atomic constraint pattern. For *UOB*, we propose the following OCL specification pattern.

```
pattern UniqueOccurrenceInBag(bag:OclExpression,
                              class:Class, attribute:Property)=
 bag->count(self.attribute) = 1
```

Its associated meta-constraints state that:
(invUOB1) the pattern instantiation is supposed to be performed as an invariant in the context of *class*;
(invUOB2) *attribute* should be among the attributes of *class*;
(invUOB3) *bag* should be a valid OCL expression that evaluates to a Bag type;
(invUOB4) *attribute* and the elements from *bag* should have the same type.

Based on the statements above, it can be easily noticed that the proposed `CUID_P2` invariant specification uses an instantiation of our previously proposed OCL specification pattern `ForAll_Reject`, composed with an atomic `UniqueOcurrenceInBag` instance.

Therefore, we propose the following OCL specification pattern, that is to be applied in all cases of "container-relative" uniqueness.

```
pattern inv_ContainerRelativeUniqueIdentifier(
               container,contained:Class,
               navigationToContained:Property, attribute:Property)=
 let bag:Bag(OclAny) = self.navigationToContained.attribute in
 ForAll_Reject(self.navigationToContained,
              Set{UniqueOccurrenceInBag(bag,attribute)})
```

Its corresponding meta constraints are:
(invCUID1) the pattern instantiation should be performed as an invariant in the context of *container*;
(invCUID2) *navigationToContained* should be a reference in *container* having the type *contained*;
(invCUID3) *attribute* should be an attribute of *contained*.

It is possible to generalize the `inv_ContainerRelativeUniqueIdentifier` pattern, such that the uniqueness constraint, instead of applying to all contained elements, would rather apply to a conveniently selected subset. Below, we give the OCL pattern that we propose in this respect. Within it, *navigation* stands for the feature[6] used to access the objects on which we want to impose the uniqueness constraint, *properties* denotes the set of constraints used to filter them, *class* stands for their classifier and *attribute* for the id-like attribute.

```
pattern inv_GenContainerRelativeUniqueIdentifier(
      container,contained:Class, attribute:Property,
      navigation:Feature, properties:Set(OclExpression)) =
 let subset:Set(contained) = self.navigation->select(e |
                                 oclAND(properties,e)) in
 let bag:Bag(OclAny) = subset.attribute in
 ForAll_Reject(subset,Set{UniqueOccurrenceInBag(bag,attribute)})
```

The pattern instantiation is constrained by the following necessary conditions:
(invGCUID1) the pattern instantiation should be performed as an invariant in the context of *container*;
(invGCUID2) *navigation* should be a feature of *container* having the type *contained*;
(invGCUID3) *attribute* should be an attribute of *contained*;
(invGCUID4) each expression from *properties* should stand for a valid OCL boolean expression.

[6] We have replaced `Property` by `Feature` in order to be able to use additional operations (AOs) as well, not only role names.

An instantiation of this pattern will be provided within the validation section.

3.3.2.3 The *Unique Identifier* Constraint Pattern - Pre/Post-condition Solutions

Until now, the solutions provided in the literature for constraint patterns such as *Unique Identifier* have only been stated in terms of class invariants. And the evaluation of invariants is, indeed, the only available alternative for statical checks of constraint satisfaction by particular model instances. In the context of MDE however, models are designed as inputs of code-generation tools. Such tools use dedicated templates for generating comprehensive model implementations, including the code for the `get/set/add/remove` operations attached to attributes and references, as well as the code for checking the fulfillment of model assertions (pre/post-conditions and invariants). In accordance with the principles of Design by Contract, at run-time, invariant preservation should be ensured by any method implementation, when executed within its precondition. In case of a standard, automatically generated method implementation, the existence of an appropriate precondition is thus essential for guaranteeing invariant preservation.

Therefore, we argue that the solutions of constraint patterns (which until now have only been given in terms of class invariants) should be enhanced by the addition of appropriate OCL specification patterns for the preconditions of operations that might break the constraints in question. In the following, we provide such OCL specification patterns for the preconditions of model operations that could violate an unique identification constraint. In the process, we consider both the “global” and the “container-relative” types of uniqueness contexts, as distinguished in the previous subsection.

“Global” uniqueness case (GUID): Let us return to the example proposed within the “global” uniqueness context, namely that of a `Person` class whose instances should be uniquely identifiable by the values of their `ID` attribute. In absence of an appropriate precondition, a basic, automatically-generated `ID` setter could break the uniqueness constraint, if executed for a person with a parameter value equal to another person’s `ID`. In order to prevent it, we promote the use of the precondition specification shape enclosed by the `setID` operation contract below.

```
context Person::setID(value:String)
pre preSet_GUID:
 Person.allInstances()->reject(p | p.ID <> value)->isEmpty()
post postSet_GUID:
 self.ID = value
```

Generalizing, we propose the following precondition specification pattern, to be instantiated in the context of *class*`::set`*attribute*`()`. The generated precondition preserves the uniqueness of *attribute*’s values among all *class* instances.

```
pattern preSet_GloballyUniqueIdentifier(class:Class,
               attribute:Property, parameter:Parameter)=
 ForAll_Reject(class.allInstances(),
               Set{AttributeValueRestriction(attribute,<>,parameter)})
```

To ensure validity of the generated OCL expression, the following meta-constraints must hold:
(preGUID1) the instantiation is performed in the context of *class*`::set`*attribute*;
(preGUID2) *attribute* is an attribute of *class*;
(preGUID3) *parameter* is the only parameter of `set`*attribute*, having the same type as *attribute*.

“Container-relative” uniqueness case (CUID): In the “container-relative” case, the uniqueness constraint for the `ID`s of employees could be broken by both `Company::AddEmployee` and `Employee::setID` operations. To avoid it, we recommend using the precondition shapes enclosed by the corresponding contracts below.

```
context Company::addEmployee(value:Employee)
pre preAdd_CUID:
 self.employees->reject(e | e.ID <> value.ID)->isEmpty()
post postAdd_CUID:
 self.employees = self.employees@pre->including(value)

context Employee::setID(value:String)
pre preSet_CUID:
 self.employer.employees->reject(e | e.ID <> value)->isEmpty()
post postSet_CUID:
 self.ID = value
```

Generalizing, we propose the following precondition patterns:

```
pattern preAdd_ContainerRelativeUniqueIdentifier(
       container,contained:Class,
       navigationToContained:Property,
       attribute:Property, parameter:Parameter) =
 ForAll_Reject(navigationToContained,
    Set{AttributeValueRestriction(attribute,<>,parameter.attribute)})

pattern preSet_ContainerRelativeUniqueIdentifier(
       container,contained:Class,
       navigationToContainer,navigationToContained:Property,
       attribute:Property, parameter:Parameter) =
 ForAll_Reject(navigationToContainer.navigationToContained,
    Set{AttributeValueRestriction(attribute,<>,parameter)})
```

The instantiation of the first pattern above generates a precondition for the `add`*contained*`()` operation of *container*, meant to preserve the uniqueness of *attribute*’s values among all instances of *contained* from within *container*. These instances are accessible by means of the *navigationToContained* reference of *container*. The following meta-constraints should be fulfilled, so as to ensure the validity of the generated OCL expression.
(preAddCUID1) the pattern instantiation context is *container*`::add`*contained()*;
(preAddCUID2) *navigationToContained* is a reference in *container* of type *contained*;
(preAddCUID3) *attribute* is an attribute of *contained*;
(preAddCUID4) *parameter* is the only parameter of `add`*contained*, having *contained* as type;

The second pattern may be instantiated to generate a precondition for the `set`*attribute* operation of the *contained* class, with the purpose of preserving the same “container-relative”

uniqueness constraint. Following, there are the meta-constraints corresponding to its parameters.
(preSetCUID1) the pattern instantiation context is *contained*`::set`*attribute*`()`;
(preSetCUID2) *navigationToContained* is a reference in *container* of type *contained*;
(preSetCUID3) *navigationToContainer* is a reference in *contained* of type *container*, having mandatory-one multiplicity;
(preSetCUID4) *navigationToContained* is the opposite of *navigationToContainer*;
(preSetCUID5) *attribute* is an attribute of *contained*;
(preSetCUID6) *parameter* is the only parameter of `set`*attribute*, having the same type as *attribute*.

3.3.3 Tool Support and Validation

As already pointed out in Section 3.1, the primary goal of any modeling activity leaded under the umbrella of MDE should be the production of *correct* models. Except for the abstraction level, the question of determining the correctness of a model can be thought of by analogy to that of establishing the correctness of a program. Namely, ensuring program correctness requires carrying out two mandatory tasks, *compilability checks* and program *testing*, a successful accomplishment of the former coming as a precondition of the latter. Compilability checks are meant to establish the correctness of a program with respect to the programming language used to write it; specifically, they test whether the program conforms or not to the syntactical and static semantics' rules of the language in question. Program testing offers a means of verifying the program against its requirements specification, which may end in proving its incorrectness. The same two steps should be undertaken when judging the correctness of a model; their specifics in a modeling context are detailed in the following.

In modeling, *compilability checks* encompass both the model itself and its associated assertions, being regarded as a mandatory prerequisite of any model transformation task (code generation included). The model should be compilable with respect to the modeling language used to describe it, while its assertions should be compilable with respect to the constraint language employed. Failure to fulfill the first requirement may trigger inability to accomplish the second. Compilability of a model with respect to its modeling language is judged by its conformance to the abstract syntax and static semantics of the language. The abstract syntax of a modeling language is described by means of its metamodel, while the static semantics is given by the Well-Formedness Rules (WFRs). Conformance to the WFRs is checked by evaluating all such WFRs (or their programming language equivalents) on the model. Starting from the assumption that all WFRs are correct (since the modeling language must have been extensively tested prior to release), failure to fulfill any of the WFRs indicates a bug in the model. Writing the WFRs with model debugging support in mind (as promoted by our proposed specification patterns) considerably facilitates this task, thus speeding up the development process.

Model *testing* aims at checking whether the model conforms or not to the domain/reality that it represents and the rules that govern it. Such rules are referred as Business Constraint Rules (BCRs), being represented by means of model-level invariants (as opposed to WFRs which are given by metamodel-level invariants). Testing is performed using snapshots (domain model instantiations), which are meant to detect faults in the model itself (e.g. missing concepts, missing/wrong relationships, attributes having wrong types), as well as bogus model BCRs. Assuming that the model itself is correct, faults in the BCRs are identified by evaluating them on the test snapshots. The detection of any false-positive (wrong snapshot that is accepted, i.e. all BCRs evaluate to true) or false-negative (good snapshot that is denied, i.e. fails to fulfill a particular BCR) points to a logical bug in the BCR expressions. Designing them with debugging support in mind may ease the task considerably. This is imperative when the model to test is, in fact, a metamodel, since, given their reuse potential, metamodels require extensive testing on sizable models.

In the following subsections, we aim at giving proof of the testing/debugging potential of the proposed specification patterns, covering both compilability checks and model testing.

All MDE-related activities (model testing and debugging included) require tools supporting both the employed formalisms and the pursued goals. The advantages derived from using the specification patterns described in this work, instead of those already proposed in the literature, may be highlighted by means of an appropriate tool, such as OCLE (Object Constraint Language Environment) [52]. We have chosen this particular OCL tool over others, since it best supports the proposed specification approach.

3.3.3.1 Model Compilability Checks

In order to emphasize the benefits that our proposal brings in ensuring model compilability, let us start from the UML 1.5 metamodel excerpt in Figure 3.2 and from a sample WFR included in the UML 1.5 specification document [65]. The WFR in question concerns the name uniqueness of model elements within namespaces. In [65], the rule is located in the `Namespace` context, having the following informal statement "If a contained element, which is not an Association or Generalization has a name, then the name must be unique in the Namespace." Below, there is the OCL equivalent of this informal WFR, as provided by the same reference. As may be seen, it is stated as an instantiation of the classical `ForAll` OCL pattern.

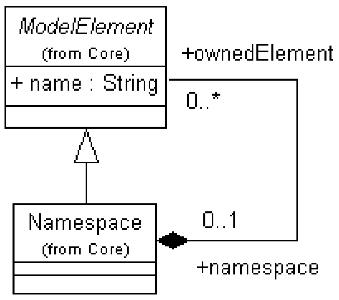


Fig. 3.2: UML 1.5 metamodel excerpt

```
self.allContents->forAll(me1, me2 : ModelElement |
  ( not me1.oclIsKindOf(Association) and
    not me2.oclIsKindOf(Association) and
    me1.name <> '' and me2.name <> '' and
    me1.name = me2.name
   )
   implies me1 = me2 )
```

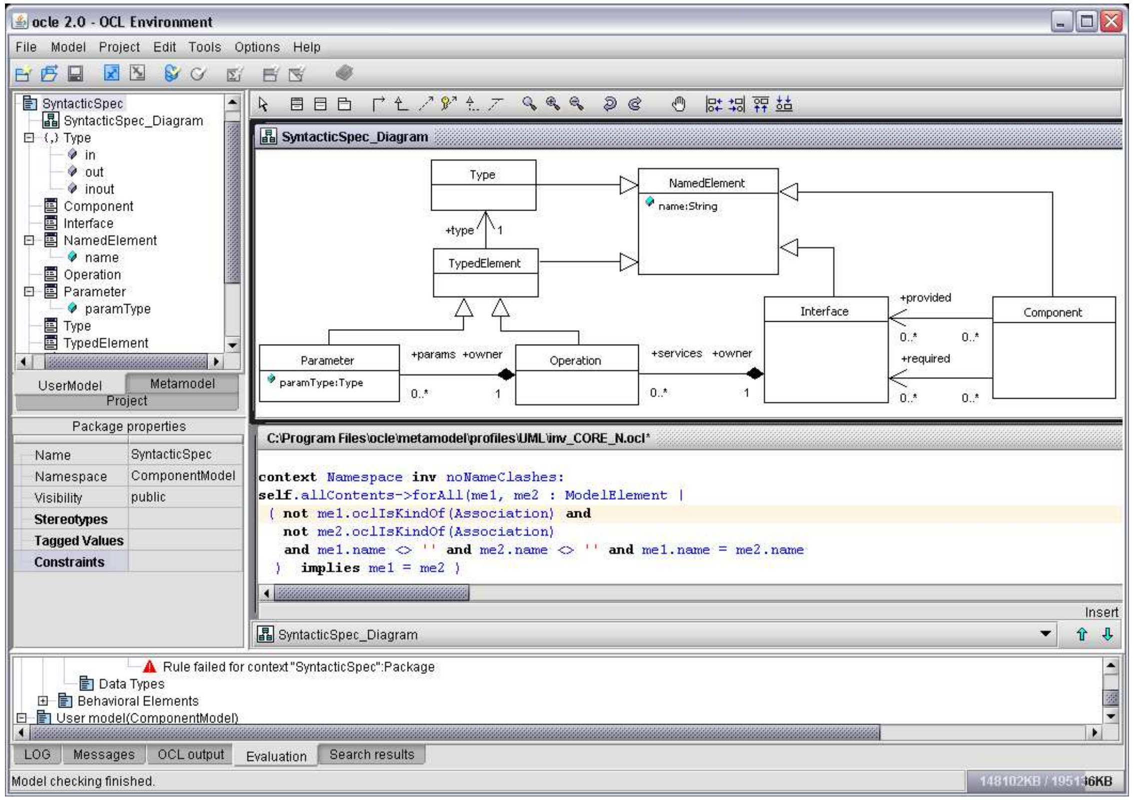
Assume that there is the need of creating an UML model for components, whose syntactic description is given as follows: "From a syntactic perspective, a component is a named entity that offers services through a set of provided interfaces, for whose accomplishment it needs to use the services provided by a set of required interfaces. Each such interface has itself a name and consists of a collection of operations. An operation is a typed entity that owns a number of parameters. Each parameter is itself a typed element, which additionally specifies data flow direction (input, output, or both)." The model part created for this syntactic description is illustrated within the model browser (top left) and diagram (top right) panels of the OCLE screenshot in Figure 3.3. We may assume that this is a part of a larger model covering all aspects related to the specification of components. The syntactic part of the model is rooted in the `SyntacticSpec` package.

OCLE allows checking the compilability of UML 1.5 models, by evaluating the meta-model WFRs against them. In order to ensure maximum flexibility, the WFRs are not hard-coded, but stored explicitely[7] as OCL expressions that may be conveniently edited.

Suppose that the OCL WFR used by OCLE for prohibiting names clashes within namespaces has been written as indicated by the UML 1.5 specification document. When checking the compilability of the component model (prior to code generation, for example), the tool reports that the WFR concerning name clashes is violated by the `SyntacticSpec` package[8], as indicated in the bottom evaluation panel. However, given the shape of the constraint, there are only two partial evaluations that could be performed on it in the attempt of discovering the model fault. First, there is the evaluation of the `allContents` additional operation, that basically returns the entire contents of the `SyntacticSpec` package; this does not offer extra debugging support, since it is also entirely visible in the model browser. Secondly, there is the evaluation of the `forAll` expression, which simply returns a `false` value, indicating constraint violation by `SyntacticSpec` - an already known information bearing no debugging help.

[7] In OCLE, WFRs are stored in dedicated ".ocl" ascii files.

[8] Any package is also a namespace.



Name	SyntacticSpec
Namespace	ComponentModel
Visibility	public
Stereotypes	
Tagged Values	
Constraints	

```
context Namespace inv noNameClashes:
self.allContents->forAll(me1, me2 : ModelElement |
 ( not me1.oclIsKindOf(Association) and
   not me2.oclIsKindOf(Association)
   and me1.name <> '' and me2.name <> '' and me1.name = me2.name
  )  implies me1 = me2 )
```

```
Rule failed for context "SyntacticSpec":Package
Data Types
Behavioral Elements
User model(ComponentModel)
```

Model checking finished.

Fig. 3.3: OCLE screenshot illustrating the sample component model

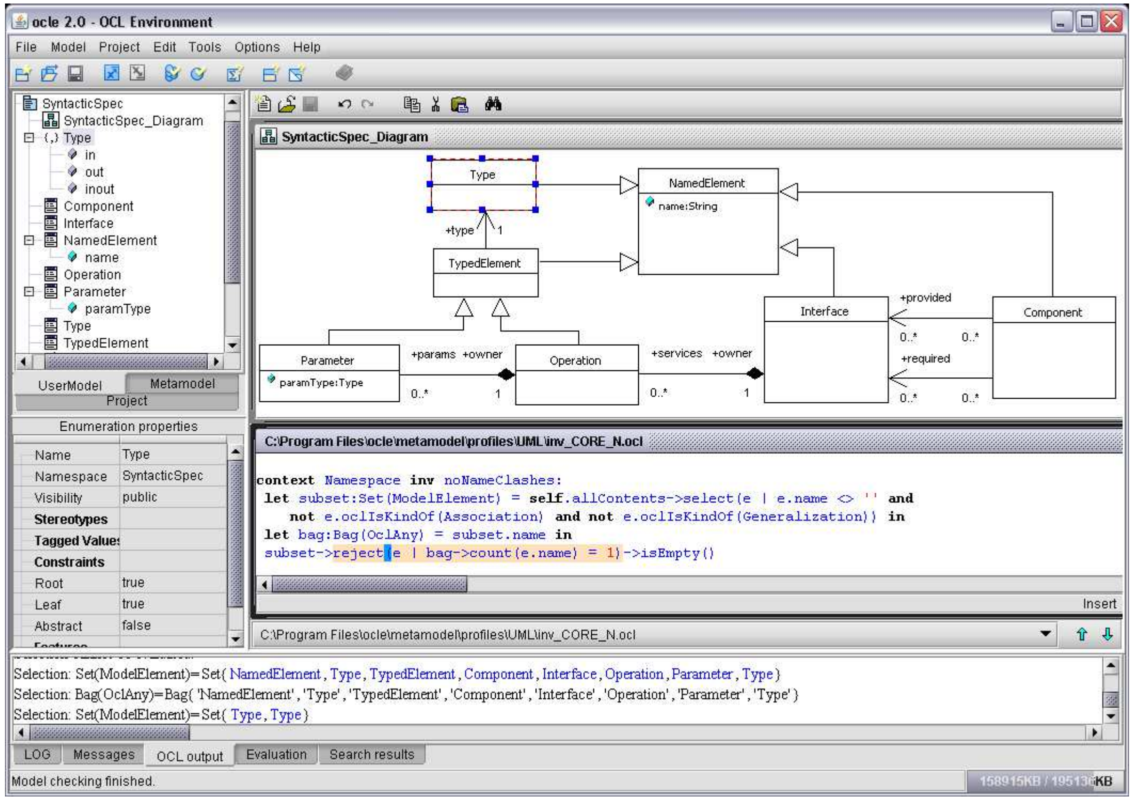
Yet, the WFR that we are dealing with can be stated as an instantiation of the generalized form of the “container-relative” *Unique Identifier* pattern given in Subsection 3.3.2.2, as shown below.

```
context Namespace inv noNameClashes:
 inv_GenContainerRelativeUniqueIdentifier(
            Namespace, ModelElement, name, allContents,
            Set{AttributeValueRestriction(name,<>,''),
                not self.oclIsKindOf(Association),
                not self.oclIsKindOf(Generalization)
               }
  )
```

The literal OCL expression resulting from instantiation is given in the middle-right panel of the OCLE screenshot from Figure 3.4.

By replacing the previous WFR definition with the new one within the metamodel constraints file, and rechecking the model, this will fail again, as expected, due to the violation of the WFR in question by the `SyntacticSpec` package. However, this time, the invariant shape enables subexpression evaluations that lead directly to the fault causing the failure, allowing to efficiently rectify it. The three lines in the bottom panel of the OCLE screenshot in Figure 3.4 correspond to the subexpression evaluations performed. The first gives the result of evaluating the `subset` helper variable, namely the subset of model elements from within the `SyntacticSpec` package to which the name uniqueness constraint applies. The second line provides the names of all those elements, as a result of evaluating the `bag` subexpression. Finally, the evaluation of the `reject` subexpression provides, on the third line, (hyperlinks to) the model elements causing the failure. As indicated by the latter, the model is faulty since it contains both a class and an enumeration having the same name, `Type`. The class has been introduced to represent the intuitive concept of type, while the enumeration is used to classify the values indicating possible data flow directions of parameters (input, output, or both). Switching to a less general, more meaningful name for the enumeration, such as `ParamDirection`, would eliminate the model error. In this respect, one of the hyperlinks made available by the last evaluation provides immediate access to the properties of the enumeration object, allowing to change its name so as to ensure compilability.

The component model employed by this proof of concepts is, in fact, a metamodel, that can be seen as an instantiation of MOF 1.4 (the metalanguage of UML 1.5). Moreover, both the UML 1.5 metamodel excerpt in Figure 3.2 and the WFR used have equivalents in the MOF 1.4 specification. Therefore, in this particular context, it would have been more natural to carry the compilability discussion above at a higher abstraction level (compilability of a metamodel with respect to its meta-metamodel). The fact that we have decreased the level, by treating the component metamodel as an ordinary UML 1.5 model and checking its compilability with respect to an UML 1.5 WFR, can be justified by tool constraints. Namely, OCLE, the only available tool supporting the approach that we promote, only works with an UML 1.5 repository in its current version.



Name	Type
Namespace	SyntacticSpec
Visibility	public
Stereotypes	
Tagged Values	
Constraints	
Root	true
Leaf	true
Abstract	false

SyntacticSpec_Diagram

C:\Program Files\ocle\metamodel\profiles\UML\inv_CORE_N.ocl

```
context Namespace inv noNameClashes:
 let subset:Set(ModelElement) = self.allContents->select(e | e.name <> '' and
    not e.oclIsKindOf(Association) and not e.oclIsKindOf(Generalization)) in
 let bag:Bag(OclAny) = subset.name in
 subset->reject(e | bag->count(e.name) = 1)->isEmpty()
```

C:\Program Files\ocle\metamodel\profiles\UML\inv_CORE_N.ocl

```
Selection: Set(ModelElement)=Set{ NamedElement , Type , TypedElement , Component , Interface , Operation , Parameter , Type }
Selection: Bag(OclAny)=Bag{ 'NamedElement' , 'Type' , 'TypedElement' , 'Component' , 'Interface' , 'Operation' , 'Parameter' , 'Type' }
Selection: Set(ModelElement)=Set{ Type , Type }
```

Model checking finished.

158915KB / 195136KB

Fig. 3.4: Approach validation in OCLE - Model compilability checks

3.3.3.2 Model Testing

To prove the advantages of our approach with respect to model testing, let us return to the model from Figure 3.1. Suppose that the domain experts have stated a business rule requiring that, within a company, each boss should have a better income than any of its subordinates, and the modelers have hastily coded it in OCL as follows:

```
context Employee inv bossHasBetterIncome:
 self.subordinates->reject(e |
       e.salary > self.salary)->isEmpty()
```

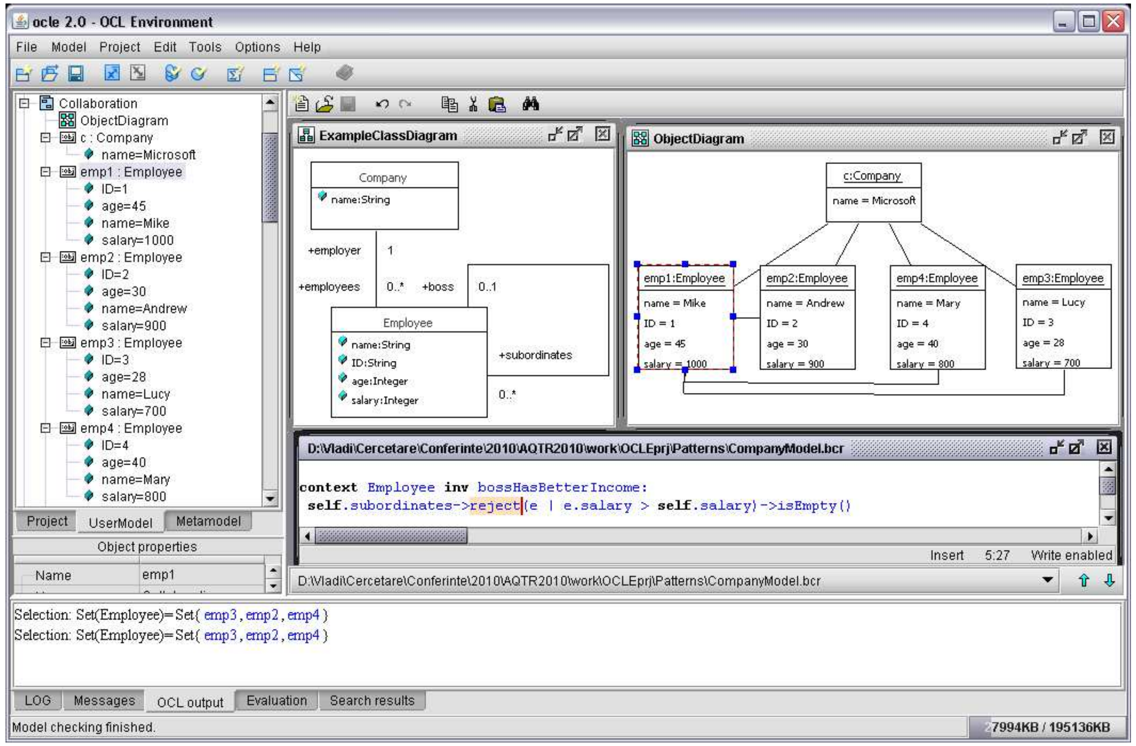
The OCL specification above is obviously an instantiation of the `ForAll_Reject` pattern, reading as:

```
context Employee inv bossHasBetterIncome:
 ForAll_Reject(subordinates,
    AttributeValueRestriction(salary,>,self.salary))
```

Assume that one of the snapshots used during the model testing phase is the one figured in the top right-most panel of the screenshot in Figure 3.5. The snapshot consists of a `Company` object with four `employees`, the employee named Mike being the boss of the others. Since the incomes of all subordinates are smaller than the one of the boss, the test is intended to be a positive one with respect to the above constraint (it is expected to pass). However, constraint evaluation fails for the boss employee, which turns the test into a false-negative (it crashes, although it shouldn't). As indicated by the bottom OCL output panel, the two partial evaluations performed for the `subordinates` reference and `reject` subexpression return the same set of employees. Therefore, all subordinates break the rule, which immediately leads to the assumption that the invariant may have been written on the reverse. And indeed, the swapping of `e` with `self` in the relational expression used by `reject` corrects the invariant, making the test succeed. If the bogus invariant had been stated as an instantiation of the classical `ForAll` pattern

```
context Employee inv bossHasBetterIncome:
 self.subordinates->forAll(e | e.salary > self.salary)
```

fault identification might have been slower, in the absence of any debugging hint. When a `forAll` fails, the result is the same (`false`), irrespective of whether the failure has been triggered by a single element in the collection or by all. Therefore, identifying the failure's cause generally involves a detailed and time-consuming examination of both snapshot and constraint. In this respect, our approach brings an efficiency improvement, by providing useful hints for error diagnosis.



```
context Employee inv bossHasBetterIncome:
 self.subordinates->reject(e | e.salary > self.salary)->isEmpty()
```

```
Selection: Set(Employee)=Set{ emp3 , emp2 , emp4 }
Selection: Set(Employee)=Set{ emp3 , emp2 , emp4 }
```

Fig. 3.5: Approach validation in OCLE - Model testing

3.4 Summary

The emergence of MDE has imposed the necessity and laid the groundwork for automatic correctness verifications of models and model-based generated applications. Such verifications rely on the use of model assertions. Assertions, such as pre/post-conditions and invariants, are needed to compensate for the narrow expressivity power of diagrammatic modeling languages. In traditional software development, which used to employ models primarily for documentation purposes, correctness and clearness were the only quality attributes required for assertions. In the context of MDE however, which strengthens the need for automatic model correctness checks, assertions should be designed so as to provide efficient support for error diagnosis.

The ever-growing use of assertions following the advent of MDE has led to the identification of several constraint patters, while the necessity of spending less time and avoiding syntax errors in their writing has motivated a few approaches aimed at automating the instantiation of these patterns [15, 16, 84, 82]. However, we argue that the constraint patterns' solutions provided until now in the literature fail in providing the level of debugging support requested for assertions in the context of MDE.

In this chapter, we have proposed a new approach (which we call *MDE-driven*) regarding the definition of constraint patterns for object-oriented modeling. It builds on two fundamental principles, namely:

1. In accordance with the model correctness-focused role of assertions within MDE, the solutions of constraint patterns should be designed so as to encapsulate increased debugging (error diagnosis) support;
2. In accordance with the finality of model-level assertions in MDE (translation into code) and with the principles of Design by Contract, the solutions of constraint patterns should be given not only in terms of invariants (as currently the case), but also in terms of matching preconditions.

We have rooted the proposed approach in a pair of solutions that we have provided for the composed *For All* constraint pattern. The semantical equivalence of our proposals with those existing in the literature has been confirmed by translation into a B abstract machine, whose proof obligations have been discharged by AtelierB. Further, we have provided appropriate solutions for the *Unique Identifier* constraint pattern. Unlike the existing approaches in the literature, we have distinguished among two possible contexts in which this pattern may occur ("global" vs. "container-relative"), giving suitable solutions for each case. In addition, the solutions for *Unique Identifier* have been given in terms of both invariants and preconditions. Our proposals have been illustrated using OCL, the standard constraint language for the MOF-based family of modeling languages; the OCL-based solutions to constraint patterns have been referred as *OCL specification patterns*.

We have validated our approach and we have emphasized its advantages compared to related work in the field using relevant cases studies and the OCLE tool. So as to maximize intelligibility and underline the relevance of our proposals, we have approached the issue of model correctness by analogy to program correctness. From this perspective, ensuring correctness involves both compilability checks and testing; we have proven the effectiveness of our approach in each of these two cases.

One may argue that the constraints generated by instantiating the proposed OCL specification patterns lack the clearness of the ones previously available in the literature. However, our approach has a great automation potential (either by instantiating the proposed solutions from scratch or by using them through automatic refactorings of old specifications), so this is by no means a limitation.

In fact, further work targets primarily at automating the use of the proposed approach in OCLE. In addition, we further aim at identifying new constraint and OCL specification patterns, along with bringing improvements to the existing ones.

Chapter 4
The Static Semantics of (Meta)Modeling Languages

While arguing our contribution from the previous chapter (a new approach concerning the definition of OCL specification patterns), we have presented it as an error diagnosing means, which serves the purpose of achieving model compilability. Within this chapter, we further elaborate on the model compilability issue, by emphasizing its compulsoriness, the state of facts in the field and the reasons beneath it, as well as our proposals for improving this state of facts.

Our contribution here is related to the set up of a framework supporting an accurate specification of the static semantics of (meta)modeling languages and enabling efficient model compilability checks. This contribution is twofold. First, we propose a set of underlying principles concerning the specification of a static semantics. Second, we provide enhancements to the definition of the static semantics of the UML metamodel and of three of the best known meta-metamodels - MOF, Ecore and XCore.

The chapter is organized as follows. Section 4.1 motivates our interest in approaching the static semantics of (meta)modeling languages, whose current specification deficiencies trigger problems in the area of model compilability. The set of principles that we propose at the basis of such a specification are given in Section 4.2. Section 4.3 details some of the enhancements that we have provided for the static semantics of UML/MOF, Ecore, and XCore. We conclude in Section 4.4, with a summary of the contributions brought by this chapter and ideas of future work on the topics approached within it.

4.1 Motivation

4.1.1 The Model Compilability Problem

As revealed in the previous chapter, the MDE paradigm has triggered a major change of focus in the field of software engineering: from *programs* and *programming languages* to *models* and *modeling languages*. As the artifacts driving a highly automated development process, a first mandatory requirement imposed on models in the context of MDE should be correctness with respect to their modeling language. Such a requirement relies on:

- rigorous definitions of modeling languages and
- the availability of modeling tools embedding appropriate verification functionalities.

The two conditions above are obviously co-dependent; the availability of accurate language specifications comes as a precondition for language-supporting tools, while tools themselves stand as prerequisites for evolving the language accuracy.

Similar to any other language, a modeling language can be defined as a three-tuple of the form (*abstract syntax*, *concrete syntax*, *semantics*) [33]. The *abstract syntax* defines the vocabulary of concepts employed by the language, together with their structural interrelationships. It has been traditionally given in terms of a class model - called *metamodel*[1], which can be visualized by means of class diagrams. Defining a metamodel is an analogous process to defining the BNF (Backus-Naur Form) grammar of a programming language. Moreover, similar to an ordinary context-free grammar, which is unable to capture static semantics' rules concerning typing or scoping, the graphical class diagram notation lacks the expressive power needed to lay down complex constraints that rule out illegal combinations of concepts in the language. Such constraints, known as *Well-Formedness Rules* (*WFRs*), define the *static semantics* of a modeling language. They are usually formalized as invariants on the metamodel, using OCL or an OCL dialect (e.g. XMF's XOCL).

As they enhance the class diagram descriptions, the major value of WFRs resides in the fact that they ensure a proper understanding of the modeling concepts' semantics. The existence of an informal (natural language) equivalent of each WFR is mandatory in this respect. Regarding the formal specification, its role is twofold. On the one side, it increases rigor and helps preventing potential ambiguities. On the other side, it lays the basis for automatically checking the compliance of models with respect to their modeling language.

Researches in the field have revealed that modeling and programming languages share more commonalities than differences [33]. The differences are mainly related to a higher abstraction level employed by modeling languages, as well as to the common use of graphical formalisms for representing their concrete syntax. The graphical formalism though is exclusively meant to support understandability, as all model processing tasks (serialization included) employ a textual syntax representation. Moreover, nowadays there are experts promoting the use of a textual concrete syntax even for model specification[2] [77], [45]. Acknowledging the truth that the graphical and textual concrete syntaxes are equivalent, that modeling and programming languages are closely similar, assists in accepting the fact that, similar to program compilability, full compliance of a model with its modeling language is a must.

Models that conform to the abstract syntax and static semantics of their modeling language are generally referred as *well-formed models* in the literature. However, we will use the phrase *compilable models* instead of *well-formed models*. The arguments are twofold. On

[1] Here, we use the term *metamodel* in its general acceptance, as denoting the abstract syntax model. According to the LDD vision however, a metamodel should capture the entire model of a language, covering also its concrete syntax and semantics.

[2] However, even in this case, the language used for model representation is complemented by a second language, targeted at model navigation and specification of constraints.

the one side, apart from WFRs, there may be other kinds of rules that a model has to comply with, such as methodological rules, metric rules, or business rules. A *well-formed model* should designate a model complying with all these rules, which is a stronger requirement than compilability alone. On the other side, the newly proposed phrase stresses on the similarity among modeling and programming languages, hence on the imperative nature of the compilability requirement.

Therefore, regardless of their size, models' compilability is a must if we want them manipulated by tools. While in case of small ones checks may be even manually performed, in case of medium-sized to large models the existence of appropriate verification instruments becomes a mandatory requirement.

Despite this, current practice shows that model compilability is rather a goal than a reality. This state of facts has both human and technological roots. On the one side, there is the unfortunate assumption that seems to be still governing the developers' community, according to which models are primarily meant to facilitate problem understanding and assist the client-developer communication, a rigorous model verification not being therefore an imperative. On the other, there are the shortcomings concerning the formalisms and tools involved in compilability assessments.

4.1.2 Diagnosing the State of Facts Regarding Model Compilability

As previously pointed out, there is a strong technological factor involved in what causes the current state of facts in the field of model compilability. Evidence of this is provided by posts or papers such as [9] and [25], reporting the fact that the existing UML tools provide either incomplete or poor implementations of the standard WFRs, thus allowing the creation of uncompilable models.

However, we argue that the problems with the aforementioned UML tools are only the visible tip of the iceberg. In fact, these problems are rooted in the absence of a general consensus within the modeling community with respect to the purpose and means of using assertions. Consequently, the real issues triggering the current state of facts in the area of model compilability are represented by the inadequate specification and deficient validation of the static semantics of (meta)modeling languages.

The statement made above is motivated by a detailed analysis that we have performed on the specification and use of assertions within the UML metamodel and three of the best known meta-metamodels (MOF [67], Ecore [78], and XCore [33]). This study has revealed that the goal of having a complete and correct static semantics' specification for each of them is far from being reached. The closest to this aim is Ecore, whose repository code contains a comprehensive set of WFRs implemented directly in Java. In case of the OMG MOF standard, a large number of OCL specifications used in describing the Core UML Infrastructure (which is part of MOF) are bogus. Unfortunately, various specification errors reported for the UML 1.x WFRs ([75], [30]) have not been fixed yet, being further inherited by the MOF 2.0 and UML 2.x documents. As for XCore, its static semantics specification does only contain two explicit XOCL WFRs.

The availability of an accurate, formal static semantics specification is vital in case of meta-metamodels, given their high reuse potential. Meta-metamodels stand at the top (level M3) of the metamodeling architectures proposed by all model-driven approaches, their abstract syntax being used in defining the metamodels of all possible modeling languages. There has to be possible to check/ensure the correctness of all these metamodels which are to be reused themselves by instantiation in thousands of modeling applications.

We argue that the solution to the above-mentioned problems consists in the adoption of a rigorous conceptual framework supporting an accurate definition of the static semantics of (meta)modeling languages and enabling efficient model compilability checks. Such a framework should build on a set of well-defined principles regarding the specification of a static semantics. Within this chapter, we aim at setting the bases of such a framework, by proposing a number of underlying principles. A proof of concepts regarding the application of these principles is provided by means of a number of WFRs for UML/MOF, Ecore and XCore. These WFRs are part of a larger set of rules that we have proposed with the aim of improving the static semantics of the (meta-)metamodels in question. The entire set can be found at [5].

4.2 Towards a Conceptual Framework Supporting Model Compilability - Principles of a Static Semantics Specification

As stated in the previous section, the compilability of a model is checked against its metamodel and associated WFRs; the metamodel defines the abstract syntax of the modeling language, while the WFRs enclose its static semantics. In order to fully serve its intended purpose of supporting efficient model compilability checks, there are a number of requirements that any set of WFRs should comply with.

The first one is *completeness*; the WFRs should entirely cover the static semantics rules of the language. This entails an intimate understanding of all metamodel-level concepts and how they may be suitably related.

A second mandatory requirement concerns the *availability of an OCL or OCL-like formalization of the entire set of rules*. At least two alternatives to this are offered by the existing approaches: the explicit implementation of rules within the metamodel repository code (the Ecore way) and an attempt at preserving their fulfillment at any time through appropriate implementations of the repository modifiers (the XCore way). However, an explicit OCL(-like) formalization is far more convenient compared to the latter two approaches. On the one side, OCL is the standard language for expressing such rules, the OCL assertions being, by nature, more compact and intelligible compared to their equivalents in a programming language. On the other, in the context of MDE, there should be tool-support available for translating OCL expressions into corresponding programming-language code (OCLE and [38] are two notable examples in this respect). As regarding the approach taken by XCore, its shortcomings will be detailed in Subsection 4.3.3.1.

In addition to the above, each WFR specification should itself fulfill a number of quality criteria. The following three are among the most important, the first two being also among the least addressed in the literature.

1. *Detailed, test-driven informal specification.* Preceding the formal WFR expression with a detailed and rigorous informal equivalent is the basic requirement for ensuring correct understandability of the rule. At its turn, the informal specification should be based on meaningful test snapshots needed for its validation (both positive and negative). By analogy to the programming approach known as *test-driven development*, this *test-driven specification* approach provides for a deeper reasoning with respect to the rules, with a positive effect on the correctness/comprehensiveness of their final statements. In fact, all good programming habits remain valid in the design of sizable OCL specifications.
2. *Testing-oriented formal specification.* The OCL WFRs should be stated so as to facilitate efficient error diagnosis in case of assertion failure. In this respect, the previous chapter argues on the use of appropriate OCL specification patterns. This quality requirement comes from acknowledging the ultimate purpose of models and assertions within a model-driven development process.
3. *Correct and efficient formal specification.* We qualify an OCL WFR as being incorrect in case it fails to satisfy one of the following two criteria. The first criterion concerns compilability, therefore conformance to the OCL standard; the second asks for a full conformance between the OCL specifications and their natural language counterparts.

Another aspect to consider when specifying WFRs refers to *choosing the most appropriate context and shape for each*. This involves understanding the differences between a WFR and a "classical invariant", as introduced by object-oriented programming (OOP) techniques. Specifically, in OOP, the semantics of invariants states that the invariant of a class should refer exclusively to relationships between the values of its attributes [59],[53]. In case the type of the attribute is a reference or a collection of references, the invariant is only allowed to constrain their existence and cardinality, being denied any access to the state of the objects attached to the references. This comes from the fact that objects should be autonomous and have exclusive control over their state. When specifying WFRs however, this rule is seldom obeyed. Generally, the invariant corresponding to a WFR refers to the state of the objects that are accessible by navigation starting from the contextual instance `self`. This semantic difference among WFRs and the "classical" OOP invariants influences the choice of their specification context, their complexity and evaluation.

Driven by the so-far stated principles, in the following section we summarize the state of facts regarding the static semantics specification of UML/MOF, Ecore and XCore and present specific proposals meant to improve it.

4.3 Enhancements to the Static Semantics of (Meta)Modeling Languages

4.3.1 Enhancements to the Static Semantics of UML/MOF

4.3.1.1 State of Facts. Related Work

OMG's UML is nowadays acknowledged as the *de facto* object-oriented modeling language. Moreover, its 1.4.2 release [66] has been adopted as an ISO standard. At its turn, MOF is the meta-metamodel standing at the core of the most popular MDE approach, which is OMG's MDA. Therefore, it is only natural that any attempt of improving the state of facts in the area of the static semantics of (meta-)modeling languages should start from the OMG specifications. Moreover, beginning with the 2.x releases, both UML and MOF share a common layer, as given by the UML 2.x Infrastructure [69].

Through the last decade, there have been various papers concerned with the adequacy of the WFRs from the OMG documents. We claim that [75], [46] and [30] are the closest to the approach presented in this chapter. [75] and [46] have actually been the first two papers providing comprehensive analysis of the UML WFRs (for the 1.3 release) and drawing an alarm with respect to the quality of their specification. They report a large amount of errors, mostly of syntactic and type-checking nature. In [30], the authors analyze various types of specification errors and propose solutions to fix the identified bugs. As the title suggests, the focus is on proposing "good practices" meant to support "a correct, clear and efficient specification". The consistency among the formal and informal specifications, the clearness of OCL expressions, the fact that evaluating OCL specifications instead of only compiling them is an imperative are among the proposed and exemplified practices.

Although most of the work signaling problems in the static semantics' definition of UML and MOF has focused on the uncompilability of WFRs with respect to OCL, a closer look at the standard specifications (both WFRs and Additional Operations (AOs)) reveals that, apart from compilability issues, the specifications in question enclose several other drawbacks, such as incompleteness, inconsistency, logical errors, as well as shortcomings caused by their deficient testing.

4.3.1.2 Proposed Enhancements

This section is aimed at providing a proof of concepts with respect to the ideas stated in Section 4.2, while contributing to the improvement of the static semantics of the UML and MOF metamodels. We take two case studies, centered around two relevant constraints for UML/-MOF: one related to composition, and the other to name clashes within namespaces. The chosen examples have not been randomly selected; they both represent core metamodeling issues. The fact that even their corresponding specifications are bogus is a strong argument towards the adoption of a rigorous WFRs specification framework, as promoted within this chapter.

The first case study reveals the incompleteness of the WFRs set enclosing the semantics of composition in both UML/MOF 1.x and 2.x, and emphasizes some inconsistencies among the informal statements and the OCL WFRs related to composition in UML/MOF 2.x. The proposed solutions stem from an analysis supported by the use of the *test-driven specification* principle. The possibility of expressing the same informal constraint in different contexts and under different shapes, as well as the criteria involved in choosing the right ones are also discussed and exemplified here.

The second case study uncovers three types of errors within the WFR and AOs prohibiting name clashes within namespaces: syntactic errors, logical ones, as well as faults coming from failure to provide the information required for error diagnosis in case the assertion gets violated. The solution proposed for the latter case involves the use of an appropriate OCL specification pattern.

To sum up, the considered case studies cover all aspects discussed in Section 4.2. Moreover, the solutions we provide contribute to improving the static semantics of the UML/MOF 1.x and 2.x metamodels.

On the UML/MOF Composition Relationship

Let us consider the UML composition relationship. As inferable from the OMG documents ([66], [69]) and papers such as [24], composition is a stronger form of association, whose semantics may be captured by the following constraints:

[C1] *Only binary associations can be compositions*;
[C2] *At most one end of an association may specify composition* (*a container cannot be itself contained by a part*);
[C3] *An association end specifying composition must have an upper multiplicity bound less or equal to one* (*a part is included in at most one composite at a time*);
[C4] *Since the composite has sole responsibility for the disposition of its parts, the parts should be accessible starting from the container* (*navigation from container to parts should be enforced*).

The above mentioned rules are equally important in defining the semantics of composition and should be all formalized at the metamodel level by means of appropriate WFRs.

In accordance with the *test-driven specification* principle, let us consider the example models from Figure 4.1.

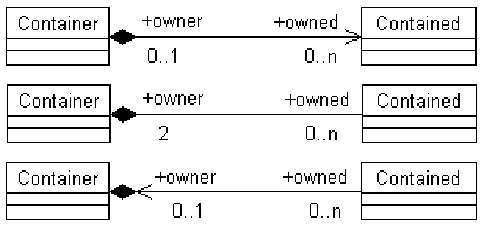


Fig. 4.1: Sample models involving composition

The first (from top to bottom) is correct with respect to the semantics of composition, as expressed by constraints [C1] to [C4]. The last two are both wrong; the second breaks the [C3] constraint (having an upper bound of 2 on the composition end), while the third violates the navigability constraint [C4] (allowing exclusively a part-to-container navigability).

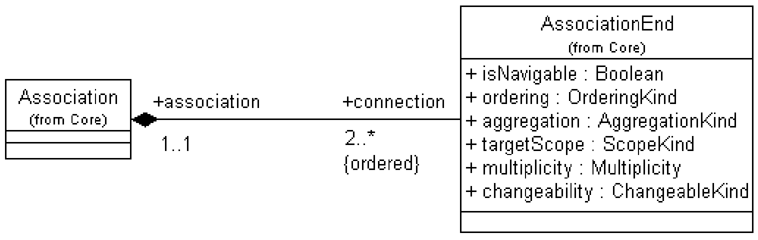

Association
(from Core)
+association
1..1
+connection
2..*
{ordered}
AssociationEnd
(from Core)
+ isNavigable : Boolean
+ ordering : OrderingKind
+ aggregation : AggregationKind
+ targetScope : ScopeKind
+ multiplicity : Multiplicity
+ changeability : ChangeableKind

Fig. 4.2: UML 1.4 metamodel excerpt illustrating `Association`s

As shown in Figure 4.2, in UML 1.x, an `Association` is defined by its two `AssociationEnds`. Composition can be specified by setting the `aggregation` enumeration attribute of `AssociationEnd` to `#composite`. With respect to enforcing the composition semantic rules [C1] to [C4], the specification only covers the first three of them. The OCL WFRs for [C1] and [C2] are stated in the context of `Association`, while the one for [C3] is written in the context of `AssociationEnd`, as follows:

```
self.aggregation = #composite implies self.multiplicity.max = 1
```

Listing 4.1: The UML 1.4 WFR for C3

The navigability constraint [C4] is missing from the UML 1.x specification, therefore the third sample model of Figure 4.1, although incorrect, would be reported as compilable. There are at least three different ways of writing this missing WFR in OCL, as shown below. Favoring one over another is a decision that depends on both language semantics and available tool facilities.

```
context AssociationEnd
 inv validCompositionNavigability1:
  self.aggregation = #composite implies
  self.association.connection->any(ae | ae <> self).isNavigable

context AssociationEnd
 inv validCompositionNavigability2:
  self.association.connection->exists(ae | ae <> self and
    ae.aggregation = #composite) implies self.isNavigable

context Association
 inv validCompositionNavigability3:
  self.connection->exists(ae | ae.aggregation = #composite) implies
  self.connection->any(ae | ae.aggregation <> #composite).isNavigable
```

Listing 4.2: Proposed WFR expressions for C4 in MOF and UML 1.x

The first two invariants from Listing 4.2 are both written in the context of `AssociationEnd`. If we were to judge from a classic invariants perspective, the second is better, since, in case of assertion failure, the objects which own the slot whose value has caused the failure (`isNavigable`) would be the ones reported as guilty. The first WFR reports the opposite ends. Nevertheless, with the aid of an OCL-supporting tool that allows the evaluation of subexpressions (such as OCLE), the other ends can be easily accessed.

The third invariant is written in the context of the `Association` metaclass. This specification is the only one fully complying with the UML 1.x composition semantics, stating that the ends of a composition association are both created and destroyed simultaneously with their owning association. According to this, the WFR in Listing 4.1 can be itself rephrased in the `Association` context, as follows.

```
context Association
 inv validCompositionUpperBound:
  self.connection->exists(ae | ae.aggregation = #composite) implies
  self.connection->any(ae |
    ae.aggregation = #composite).multiplicity.max = 1
```

Listing 4.3: Proposed WFR for C3 in MOF and UML 1.x

The WFR from Listing 4.3 and the last WFR from Listing 4.2 may also be combined within a single OCL expression, as shown below. However, this has the disadvantage of requiring partial evaluation in case of assertion failure, so as to identify precisely which expression in the conjunction has caused the failure.

```
context Association
 inv validCompositionUpperAndNavigability:
  self.connection->exists(ae | ae.aggregation = #composite) implies
  (self.connection->any(ae |
     ae.aggregation = #composite).multiplicity.max = 1 and
   self.connection->any(ae |
     ae.aggregation <> #composite).isNavigable)
```

Listing 4.4: Proposed WFR for both C3 and C4 in MOF and UML 1.x

As illustrated by Figure 4.3, the UML 2.x Infrastructure brings some changes in the definition of associations, changes that are also reflected in the MOF 2.0 specification. At the core of these changes stands the removal of the `AssociationEnd` metaclass, and its replacement with `Property`, "... associated with an `Association` via `memberEnd` attribute" [67] (pp. 66). Regarding navigability, [69] (pp. 112) states that: "An end property of an association that is owned by an end class or that is a navigable owned end of the association indicates that the association is navigable from the opposite ends, otherwise the association is not navigable from the opposite ends."

Unfortunately, concerning the semantics of composition, things seem to have worsened compared to the 1.x specifications. From those four constraints expressing the semantics of composition stated at the beginning of this section, only [C1] has a correct OCL equivalent within the specification documents. As for the others, [C4] seems to be missing, [C3] has a drawback that we will detail in the following, and [C2] appears in the MOF 2.0 specification rather as an informal precondition of the `create` operation from the `Reflection::Factory` package.

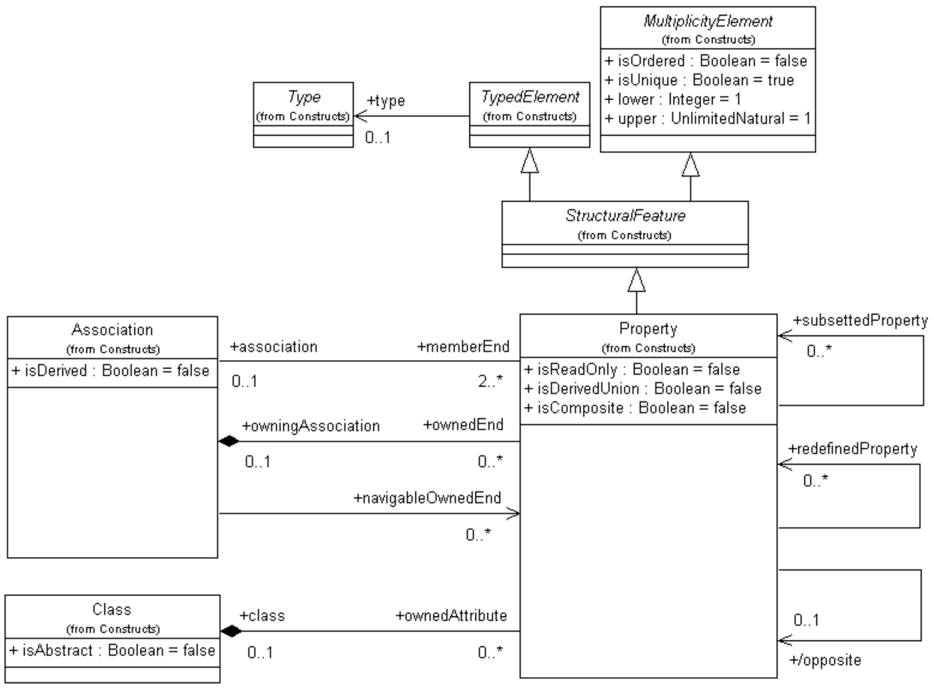


Fig. 4.3: MOF 2.0 and UML 2.3 metamodel excerpt

Regarding composition, [69] (pp. 113) states that “Composition is represented by the `isComposite` attribute on the part end of the association being set to `true`”. Given the fact that the word *composite* has a similar meaning to *container*, the previous statement is totally counter-intuitive. It basically reads as *A part in a composition is a composite/container*. In this respect, the OMG specifications should adopt a solution inspired by the EMF Ecore implementation, which has introduced the attributes `container` and `containment` with their natural interpretation.

Overpassing the language ambiguity problem, the OCL WFR corresponding to constraint [C3] found in [69] (pp.125) in the context of Property

```
isComposite implies (upperBound()->isEmpty() or upperBound() <= 1)
```

contradicts the above cited specification statement. If `isComposite` is `true`, then (in accordance with the above) the property plays the role of a part in a composition. Thus, this OCL expression constrains the upper bound of the part, instead of constraining the upper bound of its container.

Given the conflicting situation, we assume the textual statement at pp. 113 of [69], although counter-intuitive, as being the intended one. In this context, in the following, we propose appropriate OCL WFRs for each of the constraints [C2] to [C4].

The natural context for [C2] is represented by the `Association` metaclass. Its corresponding OCL invariant is given below.

```
context Association
 inv atMostOneCompositeEnd:
  self.memberEnd->select(p | p.isComposite)->size() <= 1
```

Listing 4.5: Proposed WFR for C2 in MOF and UML 2.x

The rules [C3] and [C4] can be stated both in context of `Association` and `Property`, as shown in Listings 4.6 and 4.7.

```
context Association
 inv validCompositionMultiplicity1:
  self.memberEnd->exists(p | p.isComposite) implies
  self.memberEnd->any(p | not p.isComposite).upper = 1

context Property
 inv validCompositionMultiplicity2:
  self.isComposite and self.association->notEmpty()
  implies
  self.association.memberEnd->any(p | p <> self).upper = 1
```

Listing 4.6: Proposed WFRs for C3 in MOF and UML 2.x

```
context Association
 inv validCompositionNavigability1:
  self.memberEnd->exists(p | p.isComposite) implies
  self.memberEnd->any(p | p.isComposite).isNavigable()

context Property
def: isNavigable() : Boolean =
(self.class->notEmpty()) xor
(self.owningAssociation->notEmpty() and
 self.owningAssociation.navigableOwnedEnd->includes(self))

context Property
 inv validCompositionNavigability2:
  self.isComposite and self.owningAssociation->notEmpty()
  implies
  self.owningAssociation.navigableOwnedEnd->includes(self)
```

Listing 4.7: Proposed WFRs for C4 in MOF and UML 2.x

On Forbidding Name Clashes within Namespaces

The rule prohibiting name conflicts within namespaces is among the most important WFRs, therefore, in the following, we will argue on its specification. In [69], a namespace is defined as follows: “A namespace is an element in a model that contains a set of named elements that can be identified by name.” It logically follows that the coexistence under the same namespace of at least two elements having identical names should be forbidden. The type of the elements is irrelevant. In fact, this is the only constraint specified in the `Namespace` context of `Core::Abstractions` (see [69], pp. 73). Its corresponding informal specification states that "*All the members of a Namespace are distinguishable within it*." Below is the corresponding formal specification.

```
context Namespace
 inv distinguishableName: membersAreDistinguishable()
```

The `membersAreDistinguishable()` operation is formally defined as:

```
context Namespace
 def: membersAreDistinguishable():Boolean =
  self.member->forAll( memb | self.member->excluding(memb)
      ->forAll(other | memb.isDistinguishableFrom(other, self)))
```

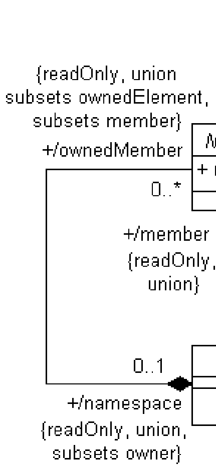


Fig. 4.4: The elements defined in the `Namespace` package

Let us assume that the namespace contains a large number of elements. If this additional operation evaluates to `false`, it is important to discover the identity of those elements producing the failure. With this aim, we propose instead the specification below, which is an instantiation of the `ForAll_Reject` OCL specification pattern introduced in the previous chapter.

```
context Namespace
 def: membersAreDistinguishable():Boolean =
  self.member->reject(memb |
   self.member->excluding(memb)->reject(other |
    memb.isDistinguishableFrom(other, self))->isEmpty())->isEmpty()
```

Both the standard specification and the proposal above employ the AO `isDistinguishableFrom(p1,p2)`. This operation is firstly defined within the `NamedElement` context, being redefined in the `BehavioralFeature` context. As stated in the [69] (pp.

72), the query "... determines whether two NamedElements may logically co-exist within a Namespace. By default, two named elements are distinguishable if (a) they have unrelated types or (b) they have related types but different names."

```
context NamedElement
 def: isDistinguishableFrom(n:NamedElement,ns:Namespace):Boolean =
  if self.oclIsKindOf(n.oclType) or n.oclIsKindOf(self.oclType)
   then ns.getNamesOfMember(self)->intersection(
     ns.getNamesOfMember(n))->isEmpty()
   else true
  endif
```

A simple analysis of this last additional operation specification suggests that this query is wrong. In both EMOF and CMOF, `Class` and `Enumeration` are unrelated types. Let `Colour` be the name of two instances, one of the `Class` metaclass (`self`), and the other of the `Enumeration` metaclass (`n`). In this case, the `if` condition evaluates to `false`, and the entire `if` statement to `true`. This evaluation result, allowing the two instances to co-exist in the same namespace, is obviously an invalid one.

As concerning the query `getNamesOfMember(m:NamedElement)`, the document [69] states that "The query `getNamesOfMember()` gives a set of all of the names that a member would have in a `Namespace`. In general, a member can have multiple names in a `Namespace` if it is imported more than once with different aliases. Those semantics are specified by overriding the `getNamesOfMember` operation. The specification here simply returns a set containing a single name, or the empty set if no name."

```
context Namespace
 def: getNamesOfMember(element:NamedElement): Set(String)=
  if member->includes(element)
   then Set{}->including(element.name)
   else Set{}
  endif
```

The above specification is not compilable, because the type of `Set{}` is `Set(OclUndefined)` and not `Set(String)`. In order to fix the bug, `Set{}` must be replaced with `oclEmpty(Set(String))`.

Behavioral features are named elements. In this case, the coexistence relationship requests that each two `BehavioralFeature`s have different signatures. Therefore, the query `isDistinguishableFrom()` specified in the `NamedElement` context must be overriden. In [69], the corresponding formal specification is:

```
context BehavioralFeature
 def: isDistinguishableFrom(n:NamedElement,ns: Namespace):Boolean =
  if n.oclIsKindOf(BehavioralFeature)
  then if ns.getNamesOfMember(self)->intersection(
        ns.getNamesOfMember(n))->notEmpty()
       then Set{}->include(self)->include(n)->isUnique(bf |
         bf.ownedParameter->collect(type))
       else true
       endif
  else true
  endif
```

This specification is not compilable because the type of `Set{}->include(self)->include(n)` is `Set(NamedElement)`, thus `bf.ownedParameter` cannot be computed since `bf` is a `NamedElement`. Following, is the correct specification.

```
context BehavioralFeature
 def: isDistinguishableFrom(n:BehavioralFeature,
                            ns:Namespace):Boolean =
  if ns.getNamesOfMember(self)->intersection(
    ns.getNamesOfMember(n))->notEmpty()
  then oclEmpty(Set(BehavioralFeature))->including(self)->including(n)
    ->isUnique(bf | bf.ownedParameter->collect(type))
  else true
  endif
```

4.3.2 Enhancements to the Static Semantics of Ecore

4.3.2.1 State of Facts

Compared to the other meta-metamodels that we have studied, namely MOF and XCore, Ecore has a special status, that may be described by the following:

- Ecore is, beyond any doubt, the best known EMOF (Essential MOF) implementation. However, Ecore does not match EMOF exactly. On the one side, the approach taken with Ecore is more pragmatic and implementation-oriented. On the other side, starting with EMF 2.3, Ecore includes constructs for modeling with generics [54]; this is considered to be a departure from EMOF, which does not currently provide such support;
- Due to the framework it ships with (EMF), Ecore is definitely the most tested meta-metamodel;
- The Ecore repository includes a set of WFRs that allow validating the metamodels that instantiate it. These rules are implemented within the EcoreValidator class. However, although the code does contain comments, these do not reflect all implementation decisions. Especially in case of those rules used to check the correctness of parameterized types, the code complexity is increased and the lack of detailed comments and examples is disturbing. The fact that, as stated in [54], “The design of Ecore’s support for generics closely mirrors that of Java itself” is expected to help in this respect. Still, the tests that we have run have shown that there are differences among the two, regarding both the declaration of generic types and their correct instantiation;
- The Ecore implementation witnesses the fact that the value of meta-metamodel level WFRs has been acknowledged. However, even though EMF integrates an OCL plugin (MDT-OCL [43]) and there is a functional approach available enabling the automatic translation of OCL assertions into Java code [38], we have not found any OCL equivalent of the implemented constraints;
- The paper [47] proposes some OCL WFRs that may be used in validating the Ecore generics; this is actually the only paper concerning the OCL formalization of Ecore WFRs that we have found in the literature. However, even though they are a good starting point and

comparison base, the OCL specifications described there are far from complete and not entirely correct.

4.3.2.2 Proposed Enhancements

Driven by the previously reported state of facts and in accordance to one of the principles exposed in Section 4.2, regarding the necessity of an OCL(-like) formalization of a static semantics, we have defined, tested and validated in OCLE a comprehensive set of OCL WFRs for the Ecore meta-metamodel. The entire set of rules can be found at [5]. Within this subsection, we discuss some WFRs related to the Ecore generics. Choosing these particular constraints for exemplification purposes is due to both their complexity level (since they are non-trivial WFRs) and the fact that they allow a close comparison with related work described in [47].

The Ecore Generics

Figure 4.5 shows that part of the Ecore meta-metamodel that ensures the generic modeling support it provides. As previously mentioned, this has been introduced starting with EMF 2.3, the newly added concepts being `ETypeParameter` and `EGenericType`. We briefly explain and exemplify these concepts in the following.

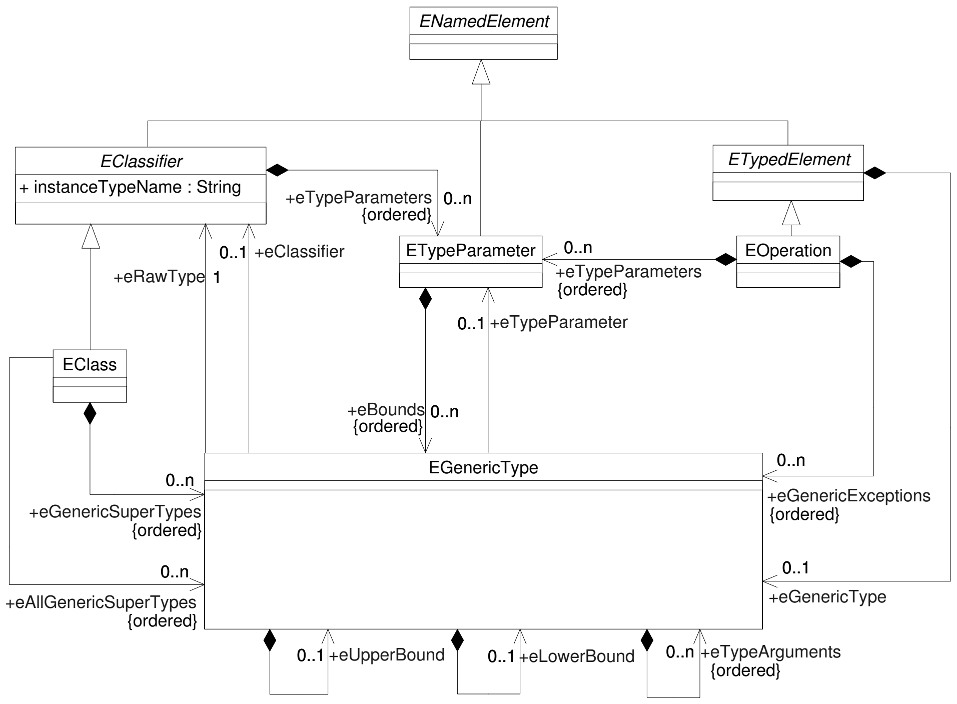


Fig. 4.5: Ecore generics

Similar to Java, Ecore supports generic type and operation declarations, as well as generic type instantiations (also known as parameterized types).

An `ETypeParameter` instance stands for a type parameter used by either a generic classifier or a generic operation declaration. That particular `ETypeParameter` is contained by its corresponding `EClassifier` or `EOperation` instance. This is denoted by the composition relationships `EClassifier` - `ETypeParameter` and `EOperation` - `ETypeParameter` from Figure 4.5, which are mutually exclusive (`xor` constraint). As an example, the Java generic type declaration `interface Collection<T>` would be modeled in Ecore by means of an `EClass` instance named `Collection`, having its `interface` attribute set to `true`, and whose `eTypeParameters` sequence contains a single `ETypeParameter` instance, named `T`.

Type parameters may have bounds, as indicated by the composition relationship between `ETypeParameter` and `EGenericType`. For a Java type declaration such as

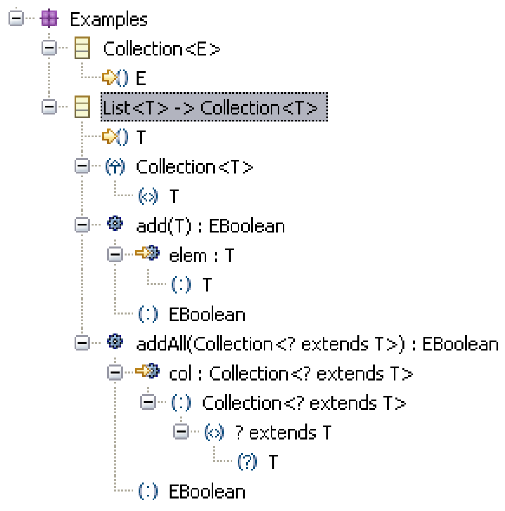
```
1 class OrderedList<T extends Comparable<T>> { ... }
```

the `eBounds` sequence owned by the type parameter `T` contains a single `EGenericType` instance, namely `Comparable<T>`.

An `EGenericType` instance may denote one of the following: a type parameter reference, a (generic) type invocation, or a wildcard. This is reflected by its associations to `ETypeParameter` and `EClassifier`, respectively. The two associations are mutually exclusive; there is a WFR specifying that an `EGenericType` instance cannot be simultaneously associated to both an `eTypeParameter` and an `eClassifier`. In case it has an `eTypeParameter`, then it is a type parameter reference, if it has an `eClassifier`, then it is a (generic) type invocation, and when both are missing, it is a wildcard. An `EGenericType` instance denoting a generic type invocation may specify type arguments (see the `eTypeArguments` role name); in case it does not specify any type arguments, then it is used as a raw type, the reason being that of ensuring compatibility with the previous, non-generic EMF releases. Wildcards may specify a lower or an upper bound (see the corresponding unary compositions of `EGenericType`). To exemplify all these, let us consider the following Java interface definition:

```
2 interface List<T> extends Collection<T>
3 {
4   boolean add(T elem);
5   boolean addAll(Collection<? extends T> col);
6   ...
7 }
```

The listing above contains a generic type declaration for `List`. In the equivalent Ecore model (Figure 4.6), this would be modeled by means of an `EClass` instance named `List`, which contains an `ETypeParameter` instance with the name `T`. The newly declared type specifies a generic supertype, `Collection<T>`. The latter is modeled using an `EGenericType` instance that corresponds to a generic type invocation with a type argument; the referred classifier is `Collection` and the contained type argument `T`. At its turn, this type argument is an `EGenericType` instance that corresponds to a type parameter reference, the referenced type parameter being the `ETypeParameter` instance `T`. The fourth line of the listing contains another `EGenericType` instance that corresponds to a type parameter reference, only this time it is used not as a type argument, but as the type of the `EParameter` instance `elem`. The type of `col` in line 5 denotes a generic type invocation; the referenced classifier is again `Collection` and the type argument is `? extends T`. The latter is an `EGenericType` instance that corresponds to an upper bounded wildcard; it has no `eClassifier` or `eTypeParameter` and it specifies an `eUpperBound`, namely `T`.



```
Examples
  Collection<E>
    E
  List<T> -> Collection<T>
    T
    Collection<T>
      T
    add(T) : EBoolean
      elem : T
        T
      EBoolean
    addAll(Collection<? extends T>) : EBoolean
      col : Collection<? extends T>
        Collection<? extends T>
          ? extends T
            T
      EBoolean
```

Fig. 4.6: Ecore model for `List<T>` (EMF tree-editor screenshot)

Therefore, `EGenericType` instances can play various roles in an Ecore model, each kind of usage being constrained by corresponding WFRs. Such an instance can be exactly one of the following:

1. A generic supertype of a class, as shown by the composition relationship `EClass-EGenericType`; `Collection<T>` in line 2 of the listing above is such an example;

2. The type of a typed element (attribute, reference, operation, parameter), as shown by the composition relationship between `ETypedElement` and `EGenericType`; an example is `T` in line 4 above;

3. A bound of a type parameter, as shown by the composition relationship `ETypePa-rameter-EGenericType`; `Comparable<T>` in line 1 above is such an example;

4. One of the type arguments of a generic type invocation, fact denoted by the unary composition relationship of `EGenericType` owning the `eTypeArguments` role; `T` from `Collection<T>` in line 2 above is a good example;

5. The upper or lower bound of a wildcard, as shown by the other two unary compositions of `EGenericType`; `T` from `Collection<? extends T>` in line 5 above is an example of an upper bound usage of a generic type;

6. An exception type, fact denoted by the composition between `EOperation` and `EGenericType`.

On a WFR for Generics in Ecore

Equipped with this knowledge regarding the Ecore generics, we seek to provide an OCL specification for the following informal WFR: "*Assuming that a generic type denotes a type parameter reference, the referenced type parameter must be in scope and must not be a forward reference. The type parameter is in scope if its container is an ancestor of this generic type within the corresponding Ecore containment tree*". In accordance with the *test-driven specification* principle, we give a few examples in the following, so as to ensure a deeper understanding of the rule and to set up some test cases for its validation. Assuming a closer familiarity of the reader with Java than Ecore, we start with the Java equivalent of each chosen example, followed by OCLE and EMF snapshots for the corresponding Ecore model.

As a first example, let us consider the Java declaration `class Cls1<P, R extends P>`. The generic type declaration for `Cls1` uses `P` and `R` as type parameters, the latter being upper bounded by the former. This is a valid generic declaration since the referenced type parameter `P` is in scope and is not a forward reference (`P` being declared prior to `R`). The equivalent Ecore model consists of an `EClass` instance named `Cls1` which contains two `ETypeParameter` instances named `P` and `R` (Figures 4.7 and 4.8 show the corresponding EMF and OCLE snapshots). The type parameter `R` has a bound, which is an `EGenericType` instance that references the type parameter `T`. The OCLE snapshot shows explicitly the `EGenericType` instance used as a bound (`GT_P`) and its link to the referenced type parameter. Within the EMF tree, the bound appears as a direct descendent of the type parameter it bounds, being labeled with the name of the referenced type parameter.

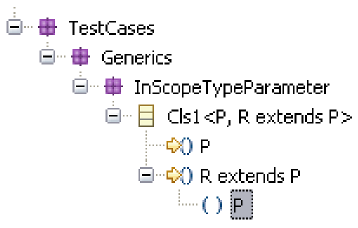


Fig. 4.7: Ecore model for Example 1: `Cls1<P, R extends P>` (EMF snapshot)

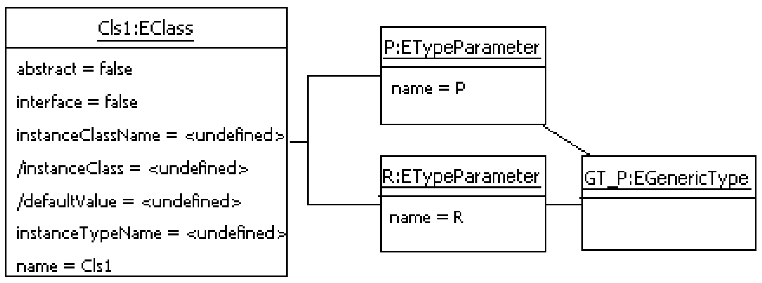


Fig. 4.8: Ecore model for Example 1: `Cls1<P, R extends P>` (OCLE snapshot)

For the second example, consider the Java declarations `class Cls2<Q>` and `class Cls3<S, T extends Q>`. The second one is obviously not valid, since the bound of `T` references type parameter `Q`, which is out of scope. The corresponding OCLE snapshot is shown in Figure 4.9; its EMF equivalent is missing, since the framework constrains a referenced type parameter to be chosen from the list of those in scope. Therefore, it is impossible to model such a case using the EMF tree-like editor. However, this erroneous situation could still occur if the model were loaded from an XMI file instead of being created directly with the editor.

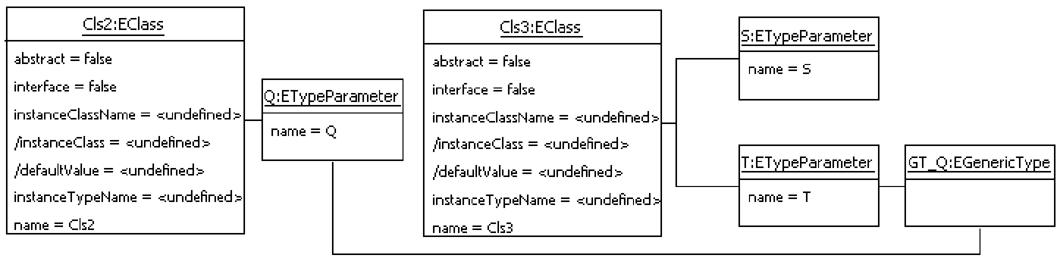

Cls2:EClass
abstract = false
interface = false
instanceClassName = <undefined>
/instanceClass = <undefined>
/defaultValue = <undefined>
instanceTypeName = <undefined>
name = Cls2
Q:ETypeParameter
name = Q
Cls3:EClass
abstract = false
interface = false
instanceClassName = <undefined>
/instanceClass = <undefined>
/defaultValue = <undefined>
instanceTypeName = <undefined>
name = Cls3
S:ETypeParameter
name = S
T:ETypeParameter
name = T
GT_Q:EGenericType

Fig. 4.9: OCLE snapshot for Example 2: `Cls2<Q>, Cls3<S, T extends Q>`

In both examples described above, the container of each type parameter has been a classifier. Let us now consider the following Java generic operation declaration `<V> void Op(V param)`. The equivalent Ecore model has at its root an `EOperation` instance, `Op`, whose `eTypeParameters` sequence contains only the type parameter `V`. `Op` owns a single parameter, `param`, whose type is an `EGenericType` instance that references the type parameter `V`. The EMF and OCLE snapshots are illustrated in Figures 4.10 and 4.11. This is, again, a valid model with respect to the WFR under consideration.

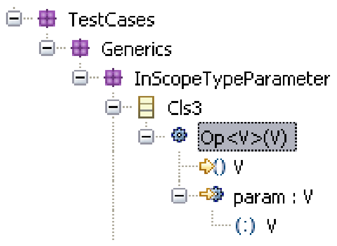


Fig. 4.10: Ecore model for Example 3: `<V> void Op(V param)` (EMF snapshot)

The fourth and last example we take is again a generic class declaration, of the form `class Cls4<T1, T2 extends T3, T3>`. Such a declaration is not valid, since the bound of `T2` performs a forward referencing of the type parameter `T3`. The equivalent snapshots are given in Figures 4.12 and 4.13.

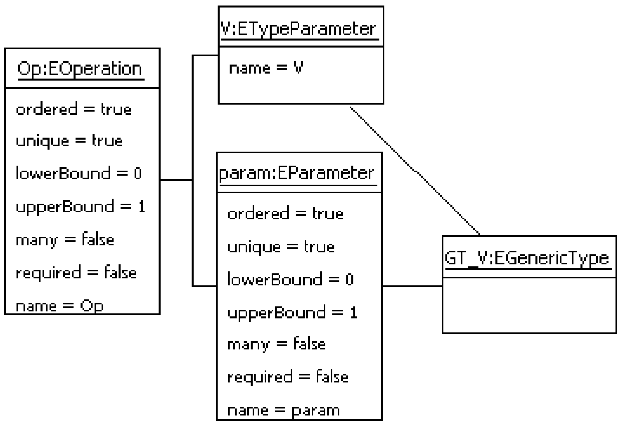


Fig. 4.11: Ecore model for Example 3: `<V> void Op(V param)` (OCLE snapshot)

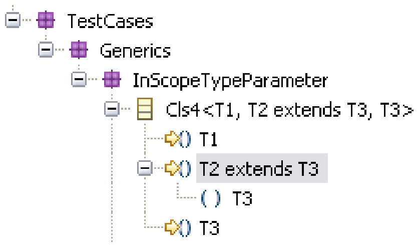


Fig. 4.12: Ecore model for Example 4: `Cls4<T1, T2 extends T3, T3>` (EMF snapshot)

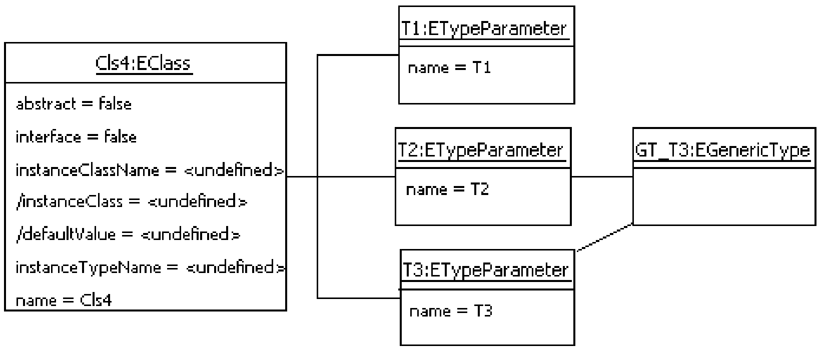


Fig. 4.13: Ecore model for Example 4: `Cls4<T1, T2 extends T3, T3>` (OCLE snapshot)

The constraints in Listing 4.8 formalize the WFR stated at the beginning of this subsection. The OCL specification has been splitted in two invariants defined for the `EGenericType` context, namely `InScopeTypeParameter` and `NotForwardReference`; as their names indicate, the former enforces the type parameter referenced by a generic type to be in scope, while the latter checks for forward referencing. As in programming, the splitting of large constraints into smaller pieces is a good modeling practice. This way, the constraints become easier to write and their comprehensibility is enhanced. Even more, this also provides valuable support in localizing exactly and in real time the cause of a constraint violation during model compilability checks.

```
context EGenericType
 -- The referenced type parameter must be in scope, i.e.,
 -- its container must be an ancestor of this generic type ...
 inv InScopeTypeParameter:
  self.isTypeParameterReference() implies
  self.ancestors()->includes(self.eTypeParameter.eContainer())

context EGenericType
 -- ... and must not be a forward reference.
 inv NotForwardReference:
  (self.isTypeParameterReference() and
   self.isUsedInATypeParameterBound())
  implies
  (let refParameter : ETypeParameter = self.eTypeParameter
   let boundedParameter : ETypeParameter = self.boundedTypeParameter()
   let paramSeq:Sequence(ETypeParameter)=
     (if refParameter.eContainer().oclIsKindOf(EClassifier)
      then refParameter.eContainer().oclAsType(
                   EClassifier).eTypeParameters
      else refParameter.eContainer().oclAsType(
                   EOperation).eTypeParameters
      endif)
   let posRefParameter : Integer = paramSeq->indexOf(refParameter)
   let posBoundedParameter : Integer =
     (if paramSeq->includes(boundedParameter)
      then paramSeq->indexOf(boundedParameter)
      else -1
      endif)
   in
   ( (posBoundedParameter <> -1) implies
     ( (posRefParameter < posBoundedParameter) or
       ( (posRefParameter = posBoundedParameter) and
         (not boundedParameter.eBounds->includes(self))
       )
     )
   )
  )
```

Listing 4.8: Proposed OCL WFRs for `EGenericType` prohibiting invalid type parameter references

We do not insist on the OCL specification for `InScopeTypeParameter` (lines 1 to 6 of Listing 4.8), since it carefully matches the comments it goes along with. However, we detail the three query operations it makes use of, namely `isTypeParameterReference()`, `eContainer()` and `ancestors()`. Their OCL definitions are provided in Listing 4.9.

```
context EGenericType
 def: isTypeParameterReference() : Boolean =
       not self.eTypeParameter.isUndefined()

context EObject
 def: ancestors() : Set(EObject) =
      let empty : Set(EObject) = Set{} in
      if self.eContainer().isUndefined() then empty
      else Set{self.eContainer()}->union(
           self.eContainer().ancestors())
      endif

context EObject
 def: eContainer() : EObject = oclUndefined(EObject)

context EGenericType
 def: eContainer() : EObject =
  let cls=EClass.allInstances->any(c |
            c.eGenericSuperTypes->includes(self))
  let param=ETypeParameter.allInstances()->any(p |
            p.eBounds->includes(self))
  let te=ETypedElement.allInstances()->any(t | t.eGenericType = self)
  let gt1=EGenericType.allInstances()->any(g |
            g.eTypeArguments->includes(self))
  let gt2=EGenericType.allInstances()->any(g | g.eLowerBound = self)
  let gt3=EGenericType.allInstances()->any(g | g.eUpperBound = self)
  let op=EOperation.allInstances()->any(o |
           o.eGenericExceptions->includes(self))
  in
  (if not cls.isUndefined() then cls
   else if not param.isUndefined() then param
        else if not te.isUndefined() then te
             else if not gt1.isUndefined() then gt1
                  else if not gt2.isUndefined() then gt2
                       else if not gt3.isUndefined() then gt3
                            else if not op.isUndefined() then op
                                 else oclUndefined(EObject)
                                 endif
                            endif
                       endif
                  endif
             endif
        endif
   endif)

context ETypeParameter
 def: eContainer() : EObject =
  let classifier = EClassifier.allInstances()->any(c |
49                      c.eTypeParameters->includes(self))
50   in
51   (if not classifier.isUndefined() then classifier
52    else EOperation.allInstances()->any(o |
53                      o.eTypeParameters->includes(self))
54    endif)

56 context EPackage
57  def: eContainer() : EObject = self.eSuperPackage

59 context EClassifier
60  def: eContainer() : EObject = self.ePackage

62 context EStructuralFeature
63  def: eContainer() : EObject = self.eContainingClass

65 context EOperation
66  def: eContainer() : EObject = self.eContainingClass

68 context EParameter
69  def: eContainer() : EObject = self.eOperation
```

Listing 4.9: Query operations used by `InScopeTypeParameter`

The core query here is `ancestors()`, which computes all parents of an arbitrary object from within the Ecore containment tree to which the object belongs. The returned set should include the object's direct container, the direct container of the latter, and so on. In case the particular object is the root of the tree, then the empty set is returned. In order to provide its intended functionality, `ancestors()` makes use of `eContainer()`, which returns the direct container of an arbitrary object.

The `eContainer()` operation is given a default definition for the root of the Ecore modeling hierarchy, `EObject`, which is then overriden in all its descendants, according to the composition relationships they are involved in (see Figure 4.14). The default implementation returns `oclUndefined(EObject)` (Listing 4.9, lines 13-14) and most of the overridings simply perform a one-step navigation of a composition relationship (Listing 4.9, lines 56-69). However, the composition relationships which involve `ETypeParameter` and `EGenericType` are uni-directional in the Ecore model, therefore the OCL expressions for `eContainer()` in these two particular cases (Listing 4.9, lines 16-54) are more complex and less efficient, due to the unavoidable calls to `allInstances()`. `ETypeParameter`, for instance, is involved in two composition relationships (with `EClassifier` and `EOperation`, see Figure 4.5 for reference), both of which are unidirectional, navigable only from container to part (although we did not manage to find the rationale for this design decision). Therefore, the direct container of a type parameter is always either a classifier or an operation. However, this container cannot be accessed through a simple navigation. The only way to identify it involves searching that particular type parameter within the `eTypeParameters` collections of all classifiers and operations that belong to the current model (which explains the use of `allInstances()` in lines 48, 52 of Listing 4.9). The operation or classifier which includes the searched type parameter within its `eTypeParameters` collection is its direct container. There will definitely be at most one such classifier or operation, since the considered relationships are compositions. Therefore, the use of the undeterministic `any()` in lines 48, 52 of Listing 4.9 is completely safe; it should produce the same result no matter the tool used. The overriding of `eContainer()` for `EGenericType` can be justified in a similar manner, only this time the number of composition relationships involved, therefore the complexity, is greater.

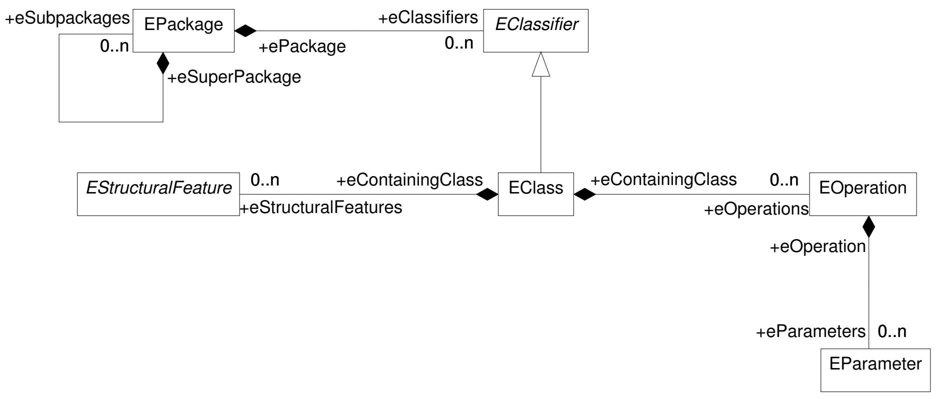


Fig. 4.14: Ecore containment relationships

The evaluation of `InScopeTypeParameter` using OCLE has ended successfully for the first, third and fourth of the test examples considered above, while failing for the second, in accordance with the results given by the EMF EcoreValidator and the Java compiler. In case of the first test example, the constraint is evaluated for the `EGenericType` instance `GT_P`, which references the `ETypeParameter` instance `P`. The computed set of `ancestors()` of `GT_P` contains the `ETypeParameter` instance `R` (its direct container, of which it is a bound) and the `EClass` instance `Cls1` (the direct container of `R`). Also, the direct container (`eContainer()`) of `P` is `Cls1`. Since the latter belongs to the `ancestors()` set, the constraint evaluates to true on `GT_P`. The evaluation results for the second and fourth examples (`false` and `true`, respectively) can be explained in a similar way. In case of the third one, the invariant is evaluated on the `EGenericType` instance `GT_V`, which references type parameter `V`. The `ancestors()` of `GT_V` are its direct container, `param`, (`GT_V` being the `eGenericType` of `param`) and the `EOperation` instance `Op` (the direct container of `param`); the `eContainer()` of `V` is `Op`, therefore the required inclusion takes place, which successfully ends the evaluation.

In order to simplify the reading, we will use the phrase “generic type” instead of “`EGenericType` instance” from here on.

The invariant that completes the proposed WFR’s OCL definition, `NotForwardReference`, rules out all generic types which reference type parameters that are in scope, but are declared afterwards. This situation can only occur when the generic type is contained in a type parameter bound, either of a classifier or of an operation. The employed query operations, `isUsedInATypeParameterBound()` and `boundedTypeParameter()`, are defined in Listing 4.10.

A generic type is said to be used in a type parameter bound if and only if there is an `ETypeParameter` instance among its ancestors. This is expressed by means of line 4 of Listing 4.10 above. If that particular `ETypeParameter` instance is its direct container, then the generic type identifies itself with the bound, otherwise it is contained at a certain level in this bound. If a generic type is involved in defining the bound of a type parameter, then this will be the only `ETypeParameter` instance among its ancestors (since no containment, direct or not, is possible among type parameters). Therefore, the use of `any()` within the OCL expression which returns the type parameter in whose bound the current generic type is involved (line 9, Listing 4.10) is safe.

```
1  context EGenericType
2   def: isUsedInATypeParameterBound() : Boolean =
3   -- checks whether self is used in defining a type parameter bound
4    self.ancestors()->exists(o | o.oclIsTypeOf(ETypeParameter))

6  context EGenericType
7   def: boundedTypeParameter() : ETypeParameter =
8   -- returns the type parameter in whose bound self is used
9    self.ancestors()->any (o |
10     o.oclIsTypeOf(ETypeParameter)).oclAsType(ETypeParameter)
```

Listing 4.10: Query operations used by `NotForwardReference`

Resuming to the OCL definition of `NotForwardReference` (lines 8-37, Listing 4.8), we should make clear that a forward reference can only happen when a generic type, let us call it `GT`, references a type parameter, let us call it `T`, which is declared at a later time compared to the moment of use of `GT`. Since type parameters can only be declared within a generic classifier or operation definition, and assuming that the referenced type parameter is in scope (out of scope type parameters are ruled out by the first invariant), it follows that `GT` can only be involved in defining a bound for a type parameter owned by the same classifier or operation that owns `T`. This explains line 12 from the definition of `NotForwardReference`. Following this, the invariant computes the referenced type parameter (`refParameter`), the bounded type parameter (`boundedParameter`), and the sequence of all parameters owned by the direct container of `refParameter` (`paramSeq`). For the invariant to evaluate to `true`, `refParameter` should be declared prior to `boundedParameter` in `paramSeq` (line 31, Listing 4.8), or the two should be the same type parameter (line 32, same listing). In the latter case, however, it is prohibited for a type parameter to bound itself (line 33). Therefore, a situation such as the following `class Cls5<P1, P2 extends P2>` is not allowed, since `P2` bounds itself. Still, a declaration of the kind `Cls6<P3, P4 extends Cls<P4> >`, in which `P4` is involved in defining its own bound, is valid.

From the test examples above, the one intended to capture forward referencing was the fourth. There, the `NotForwardReference` invariant will be evaluated for the generic type `GT_T3`, which bounds type parameter `T2` and references type parameter `T3`. The sequence of all parameters having the same container as the referenced one evaluates to `Seq{T1,T2,T3}`, from which it is obvious that the position of the referenced type parameter (3) is greater than the one of the bounded parameter (2). Therefore, the boolean expression in line 31 of Listing 4.8 evaluates to `false`, and so does the whole invariant.

4.3.2.3 Related Work

As already mentioned in the beginning of this subsection, the only benchmarks we have for comparing our work with are the EMF implementation of the EcoreValidator and the paper [47].

The EMF EcoreValidator

We have already made clear in Section 4.2 which are the advantages derived from using OCL, instead of a programming language, in formalizing WFRs. In addition, in Subsection 4.3.2.1, we have pointed out some of the drawbacks of the current EMF implementation of the Ecore WFRs. Among them, there have been mentioned some discrepancies between the Java specification of generics and the corresponding WFRs implemented by the EMF EcoreValidator. We will take one such example in the following, so as to justify a new OCL WFR that we propose for the Ecore generics and that should also be implemented by the EcoreValidator.

Concerning the correct declaration of generic types and methods, the Java Language Specification [48] (pp. 50) states the following constraints: "*Type variables have an optional bound, T & I_1 ... I_n. The bound consists of either a type variable, or a class or interface type T possibly followed by further interface types I_1, ..., I_n. ... It is a compile-time error if any of the types I_1 ... I_n is a class type or type variable. The order of types in a bound is only significant in that ... and that a class type or type variable may only appear in the first position.*"

Therefore, a generic type declaration of the kind

```
class GenericClass1 <T1 extends InterfaceA & ClassB>
```

where `InterfaceA` is an interface type and `ClassB` is a class type, gives the following compile-time error in a Java environment "The type ClassB is not an interface; it cannot be specified as a bounded parameter". However, by modeling the exact same type in EMF and validating it, the validation completes successfully.

In a similar manner, the declaration

```
class GenericClass2 <T1, T2 extends T1 & InterfaceA>
```

generates the Java compile-time error "Cannot specify any additional bound InterfaceA when first bound is a type parameter", while its equivalent Ecore model validates successfully under EMF.

This is due to the fact that the `EcoreValidator` class does not include code for checking the above mentioned constraints. Therefore, we propose the following OCL WFR for the Ecore generics:

```
context ETypeParameter
 inv ValidBounds:
  -- If a type parameter has bounds and the first bound is a
  -- type parameter reference, then there are no other bounds.
  (self.eBounds->notEmpty() and
   self.eBounds->first().isTypeParameterReference()
   implies self.eBounds->size() = 1
  )
  and
  -- If there are at least two bounds, then all
  -- except (maybe) the first one should refer to interface types.
  (self.eBounds->size() >= 2
   implies Sequence{2..self.eBounds->size()}->reject(i |
     self.eBounds->at(i).hasInterfaceReference())->isEmpty()
  )
```

Listing 4.11: Proposed OCL WFR for `ETypeParameter` prohibiting invalid type parameter bounds

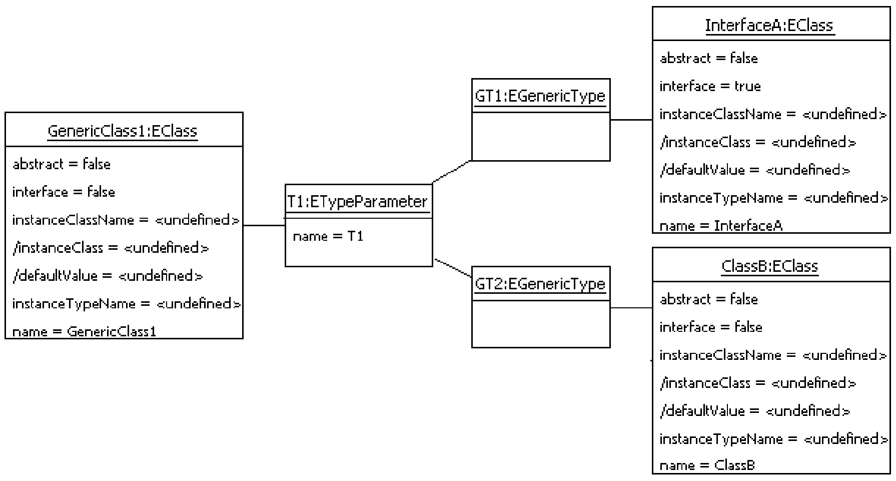

InterfaceA:EClass
abstract = false
interface = true
instanceClassName = <undefined>
/instanceClass = <undefined>
/defaultValue = <undefined>
instanceTypeName = <undefined>
name = InterfaceA
GT1:EGenericType
GenericClass1:EClass
abstract = false
interface = false
instanceClassName = <undefined>
/instanceClass = <undefined>
/defaultValue = <undefined>
instanceTypeName = <undefined>
name = GenericClass1
T1:ETypeParameter
name = T1
GT2:EGenericType
ClassB:EClass
abstract = false
interface = false
instanceClassName = <undefined>
/instanceClass = <undefined>
/defaultValue = <undefined>
instanceTypeName = <undefined>
name = ClassB

Fig. 4.15: OCLE snapshot corresponding to `GenericClass1 <T1 extends InterfaceA & ClassB>`

The above WFR makes use of the following query operations:

```
context EGenericType
  def: hasClassifierReference() : Boolean =
    not self.eClassifier.isUndefined()

  def: hasClassReference() : Boolean =
    self.hasClassifierReference() and
    self.eClassifier.oclIsTypeOf(EClass)

  def: hasInterfaceReference() : Boolean =
    self.hasClassReference() and
    self.eClassifier.oclAsType(EClass).interface

  def: isTypeParameterReference() : Boolean =
    not self.eTypeParameter.isUndefined
```

Listing 4.12: Query operations used by `ValidBounds`

By evaluating the proposed WFR on the OCLE snapshots given in Figures 4.15 and 4.16, the obtained result is `false` in both cases, in accordance with the Java specification.

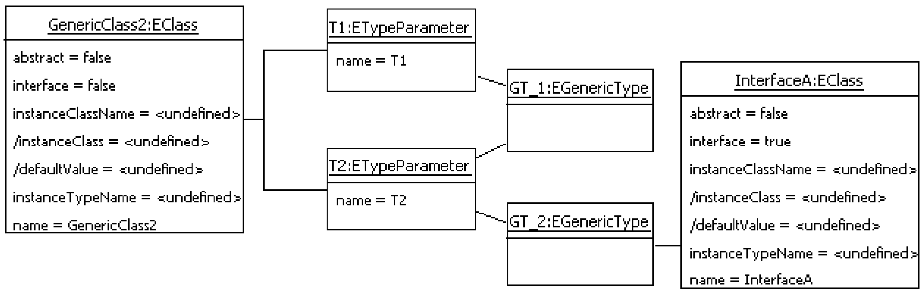

GenericClass2:EClass
abstract = false
interface = false
instanceClassName = <undefined>
/instanceClass = <undefined>
/defaultValue = <undefined>
instanceTypeName = <undefined>
name = GenericClass2
T1:ETypeParameter
name = T1
T2:ETypeParameter
name = T2
GT_1:EGenericType
GT_2:EGenericType
InterfaceA:EClass
abstract = false
interface = true
instanceClassName = <undefined>
/instanceClass = <undefined>
/defaultValue = <undefined>
instanceTypeName = <undefined>
name = InterfaceA

Fig. 4.16: OCLE snapshot corresponding to `GenericClass2 <T1, T2 extends T1 & InterfaceA>`

The Approach Taken in [47]

In the following, we will focus on comparing our work with the one described in [47]. This paper aims at stating a set of OCL constraints that allow checking whether (1) a given generic type declaration or (2) a corresponding instantiation with type arguments (a so-called parameterized type) are well-formed or not. For further reference and comparison, we provide in Listing 4.13 the OCL code for the `consistentTypeParameters` WFR, meant to accomplish the first goal above[3].

```
context EClassifier
 inv consistentTypeParameters:
  allDifferent(eTypeParameters.name) and
  eTypeParameters->forAll(tp | tp.isConsistent(eTypeParameters))

context ETypeParameter::isConsistent(
          tpsInScope:Collection(ETypeParameter)):Boolean
 def: self.name <> '' and (self.eBounds->isEmpty() or
      self.eBounds->forAll(tr |
          tr.isConsistentTypeReference(tpsInScope)))

context EGenericType::isConsistentTypeReference(
          tpsInScope:Collection(ETypeParameter)):Boolean
 def: not isWildcard() and
      ( (self.isReferenceToTypeParameter() and
         tpsInScope->includes(self.eTypeParameter))
        xor
        (self.isReferenceToClassifier() and
         self.eClassifier.isValidTypeInvocation(self.eTypeArguments))
      )
```

[3] The OCL specification used for accomplishing goal number 2 is omitted, since it is not directly comparable with the proposed WFRs presented in this section.

```
22 context EGenericType::isReferenceToTypeParameter():Boolean
23  def: eClassifier->isEmpty() and
24       not eTypeParameter->isEmpty() and eTypeArguments->isEmpty()

26 context EGenericType::isReferenceToClassifier():Boolean
27  def: not eClassifier->isEmpty() and eTypeParameter->isEmpty()
```

Listing 4.13: The `consistentTypeParameters` constraint from [47]

Its corresponding informal specification, as can be deduced from the paper, would be the following: "*The type parameters of any classifier should have non-empty, distinct names. The bounds of a type parameter (if any) can reference either a type parameter or a classifier (they cannot be wildcards). If the bound references a type parameter, then the referenced parameter should be in scope; if it references a classifier, it should be a valid type invocation (either non-generic or generic, possibly raw).*"

The set of constraints described in [47] can be analyzed with respect to both its declared purpose and our final goal of defining a complete set of OCL WFRs for Ecore in general, and Ecore generics in particular.

Regarding the first criterion, there are certain shortcomings concerning these constraints, that we mention briefly in the following:

1. The proposed constraints are incomplete with respect to their intended purpose. On the one side, those meant to check the well-formedness of a generic type declaration only constrain the bounds of a type parameter to reference parameters from within the same type declaration, without prohibiting forward references (lines 15-16, Listing 4.13). However, forward referencing is not allowed, neither in EMF not in Java (whose generics' model has inspired the one in Ecore). On the other side, for the WFRs that check the correct instantiation of a generic type definition (which are not reproduced here), only a skeleton is given. The OCL expressions for `captureConversion(...)` and `isSuperTypeOf(...)` (which are the core queries of these WFRs, both in matter of complexity and functionality) are missing from the paper and no reference to them is provided. As a consequence, it is impossible to evaluate on snapshots the correctness or efficiency of those WFRs;
2. There is some redundancy in the OCL specification. The `isConsistentTypeReference(...)` query operation (starting in line 12 above) states that a generic type used in a parameter bound (1) should not be a wildcard and (2) should reference either a classifier or a type parameter. The latter implies the former, so it should be enough to only impose (2) as a constraint. Moreover, the `isConsistent(...)` operation (starting in line 6) requires any type parameter to have a nonempty name. However, in Ecore, `ETypeParameter` inherits `ENamedElement`, and the latter owns a WFR that checks the well-formedness of its `name` attribute (well-formed implies not empty);
3. The use of `forAll` should be avoided, in accordance to the OCL specification patterns proposed by us in the previous chapter.

With respect to defining a complete set of WFRs for the Ecore generics, those described in [47] are only a small subset. They are only focused on the definition and instantiation of generic classifiers; generic operations are not taken into account. Moreover, even if a given generic type is a valid instantiation of a certain generic classifier, depending on its usage, it may be further constrained. There are various ways of using such a generic type (we have detailed on that in the previous subsection), with several constraints that result thereof. Here are some examples: a generic type used as a generic supertype should have a classifier that refers to a class; there may not be two different instantiations of the same generic classifier among the generic supertypes of a class; the classifier of a generic type that types an attribute should be a data type instance, while the one used for a reference should be a class instance, and so on.

The proposed Ecore WFRs included in this chapter are part of a broader set of OCL WFRs that we have defined [5], aimed at covering all constraints that apply to the Ecore concepts, generics included. Their expressions have been written following OCL specification patterns that provide for an easy debugging in case of an evaluation-time error. Lines 13-14 of Listing 4.11 are a proof of this. Unefficient OCL constructs have only been used when there has been no other option (see the discussion related to the use of `allInstances()`), and the safety of using undeterministic constructs such as `any()` has been justified whenever the case. All WFRs have been tested and validated using OCLE.

The OCL WFRs that we have defined for generics take into account both generic classifier and generic operation declarations, as well as all previously mentioned usages of a generic type. From a completeness perspective, our WFR from Listing 4.8 is stronger than its equivalent part from Listing 4.13, since it checks for forward referencing, and this aligns it with both the EMF implementation and the Java specification; the one proposed in Listing 4.11 is missing from the EMF implementation, while being enforced by the Java specification.

4.3.3 Enhancements to the Static Semantics of XCore

4.3.3.1 State of Facts

XCore is the bootstraping kernel of XMF, a MOF-like metamodeling facility focused on capturing all aspects of a language definition - abstract syntax, concrete syntax and semantics. Unlike MOF though, XMF is completely self-defined and provides platform-independent executability support by means of an executable OCL dialect named XOCL.

The official XMF reference [33] acknowledges the value of WFRs and promotes their use in defining the static semantics of modeling languages. Still, the document does not describe (neither informally, nor formally) any WFR for the XCore meta-metamodel. As regarding the XMF implementation, this does only include two explicit XOCL constraints, specified in the context of the `Element` and `Object` classes, respectively. Apart from these, there seem to be also a number of other constraints which are only inferable from the XOCL code corresponding to the XCore modifiers.

We argue that this XMF approach, that omits the explicit definition of WFRs, trying to preserve model consistency only by means of a suitable implementation of modifiers, has a number of drawbacks.

- In case of an executable language such as XMF, which also provides an interpreter console, one can never assure that model changes will be performed exclusively by calling the corresponding modifiers in the prescribed order. Direct assignments or different call sequences are also possible, leading to potentially invalid models.

- As emphasized in [59], this approach may be seen as an alternative to the use of preconditions. As opposed to preconditions however, it induces an increased code complexity, with a negative effect on reliability.
- Complex constraints generally involve multiple classes and the necessity of "adjusting" the code of several modifiers. Overlooking to check for the rule in any of these modifiers may lead to incorrect models. Instead, writing explicit WFRs is simpler, clearer, and less error-prone.
- Trying to preserve model consistency at all stable times may not be the best solution always. Underspecification, for instance, may be desirable in particular circumstances.

Writing explicit WFRs is a prerequisite in enforcing them. Even with the approach taken, the XMF implementation does not cover some of the elementary WFRs that are compulsory for object-oriented concepts, such as avoiding name conflicts among features of the same class/classifier or the proper management of contained-container dependencies.

4.3.3.2 Proposed Enhancements

As a solution to the above-mentioned problems, we have proposed a set of XOCL WFRs for the XCore meta-metamodel, which we have tested on relevant model examples. The entire set of rules, together with the corresponding tests, can be consulted at [5]. Below, we only discuss two relevant examples, related to name uniqueness and containment respectively.

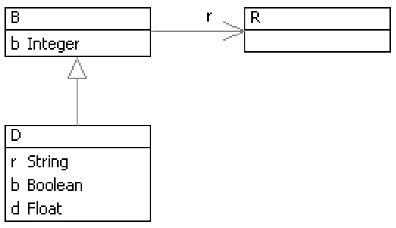


Fig. 4.17: An XCore model exhibiting name conflicts

On Avoiding Name Conflicts Among Owned and Inherited Members

As previously stated, one of the WFRs not covered by the XMF implementation concerns the name conflict among an attribute owned by the current class and attributes inherited from its ancestors. This is a fundamental object-oriented modeling constraint, being enforced by object-oriented programming languages as well. Such a conflict should arise in case of class `D` from Figure 4.17, which defines the attributes `b` and `r`, having identical names with an inherited attribute and reference, respectively.

In order to identify such invalid models, we propose a WFR with the following informal statement: *There should not be any name conflicts among the attributes owned and inherited* *by a class*. Listing 4.14 provides its formal XOCL equivalent. The referenced part of the XCore metamodel is illustrated in Figure 4.18.

```
context Attribute
 @Constraint uniqueName
  let allAtts = self.owner.allAttributes() then
      sameNameAtts = allAtts->excluding(self)->select(att |
                       att.name.asSymbol() = self.name.asSymbol())
  in sameNameAtts->isEmpty()
  end

fail
  let sameNameAtts = self.owner.allAttributes()->excluding(self)->
        select(att | att.name.asSymbol() = self.name.asSymbol()) then
      msg = "Attribute name duplication! " +
            "Inherited/owned attributes of " +
            self.owner.toString() + " with the same name: "
  in @While not sameNameAtts->isEmpty() do
       let att = sameNameAtts->sel
       in msg := msg + att.owner.toString() + "::" + att.toString()
                   + "; ";
          sameNameAtts := sameNameAtts->excluding(att)
       end
     end;
     msg
  end
end
```

Listing 4.14: Proposed XOCL WFR prohibiting name conflicts among owned and inherited attributes of a class

Apart from the constraint itself, XMF allows the specification of a `fail` clause, whose body is intended to provide model debugging information in case of assertion failure. This facility is in accordance with the OCL specification patterns that we have proposed in the previous chapter.

On the XCore Containment Relationship

The proper management of containment (composition) relationships is a fundamental issue in metamodeling. This topic has been also approached in Subsection 4.3.1.2, in the context of UML. As shown by the metamodel excerpt in Figure 4.18, XCore represents containments explicitly, by providing the `Contained` and `Container` abstract metaclasses in this purpose. Below, we give their description, as taken from the XMF Mosaic documentation [3].

> "A *container* has a slot contents that is a table[4]. The table maintains the contained elements indexed by keys. By default the keys for the elements in the table are the elements themselves, but sub-classes of container will modify this feature accordingly. Container provides operations for accessing and managing its contents."
> "A *contained* element has an owner. The owner is set when the contained element is added to a container. Removing an owned element from a container and adding it to another container will change the value of owner in the contained element."

[4] According to the metamodel, this rather seems to be the description for `IndexedContainer`, due probably to a lack of synchronization between metamodel and documentation.

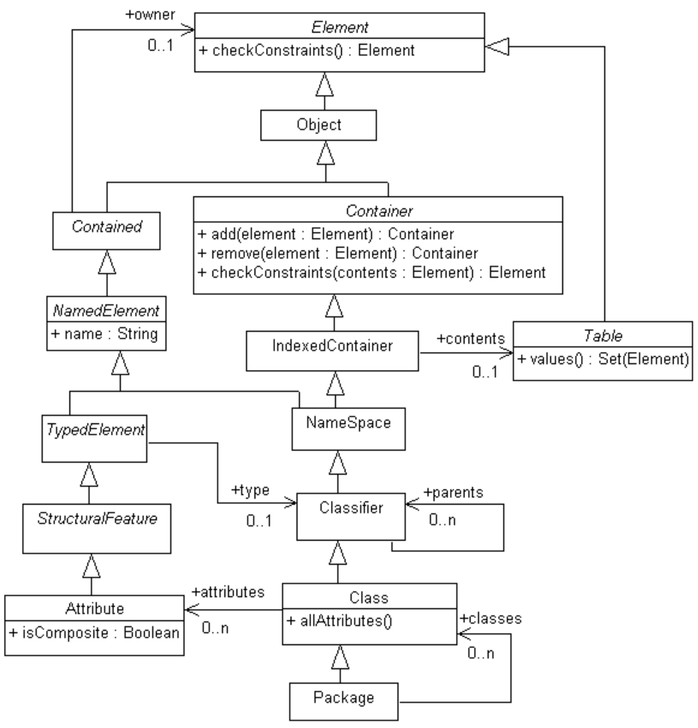


Fig. 4.18: An excerpt of the XCore meta-metamodel

According to the commonly-agreed semantics of containments, we argue that there are two fundamental rules that any XCore model should fulfill in this respect. These rules correspond to the UML composition constraints [C3] and [C2] respectively, as stated in Subsection 4.3.1.2.

[C1’] *A part should belong to a single container at a given time.*
[C2’] *A container cannot be itself contained by one of its parts.*

As in case of other constraints, the enforcement of the ones above was meant to be covered in XMF by an appropriate implementation of operations in the descendants of `Container` and `Contained`. Moreover, in order to preserve models’ validity, these operations are expected to be called in a particular sequence during model editing tasks. As a consequence, the models created using the model/diagram editors of the XMF tool (XMF-Mosaic) are correct with respect to these rules. However, the models edited using the interpreter console (where there is freedom with respect to the type and sequencing of the editing operations) may reach invalid states, which are impossible to detect in the absence of explicitly stated WFRs.

In order to exemplify this for the rule [C1’], let us start from a sample XCore model containing an empty package named `Test1` (which has been assigned to a global variable `t1`), and the following sequence of XOCL commands executed within the XMF interpreter console.

```
p1 := Package("P1");
t1.add(p1);
p2 := Package("P2");
t1.add(p2);
c := Class("C");
p1.add(c);
```

The lines above modify our initial model by creating two new packages, `P1` and `P2`, which are added as subpackages of `Test1`, and a class, `C`, which is added to package `P1`. As a consequence, class `C` will have `P1` as its `owner`, while `P1` will have `C` as the only element within its `contents` table.

Suppose `C` has been mistakenly added to `P1`, when, in fact, it should have been added to `P2`. Issuing the following command in the console

```
p2.add(c);
```

apparently solves the problem, since the owner of `C` is changed to `P2`, and `C` is added to the contents table of `P2`. However, `C` still belongs to the contents table of `P1`, from which it should have been removed prior to its addition to `P2`. Therefore, in the current state, the model is invalid with respect to rule [C1'], as `C` simultaneously belongs to two different containers (`P1` and `P2`). A visual proof of this is given by the model browser on the left of the XMF-Mosaic screenshot from Figure 4.19, illustrating the state of the model as reached after the execution of the above commands.

Still, even if the model is obviously wrong, the lack of an appropriate WFR makes the call to `checkConstraints()` on `Test1` report this package and its entire contents as valid. The XOCL WFR that we propose below offers a solution to this problem.

All `Contained` *instances that belong to the contents table of an* `IndexedContainer` *should have that container as owner.*

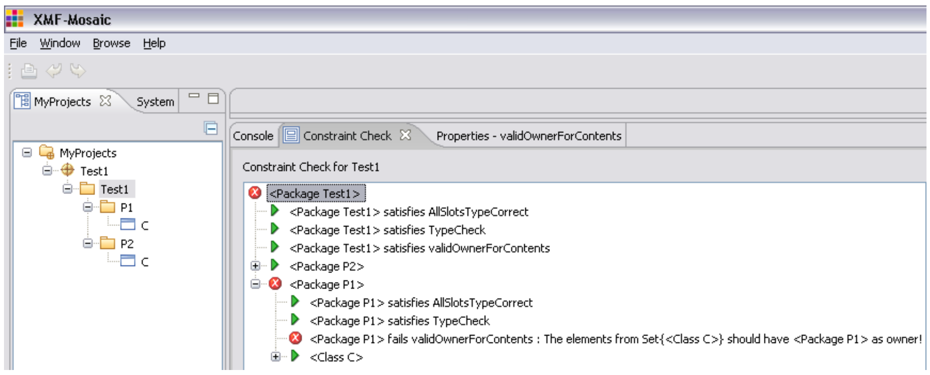
```
context IndexedContainer
 @Constraint validOwnerForContents
  self.contents.values()->select(v | v.oclIsKindOf(Contained) and
         v <> null)->reject(v | v.owner = self)->isEmpty()

  fail "The elements from " +
       self.contents.values()->select(v | v.oclIsKindOf(Contained)
         and v <> null)->select(v | v.owner <> self).toString() +
       " should have " + self.toString() + " as the owner!"
 end
```

Listing 4.15: Proposed XOCL WFR for containment constraint C1'

As shown by the right-hand side of the screenshot in Figure 4.19, the constraint checking performed after the addition of the above constraint to `IndexedContainer` reports the `P1` package as invalid with respect to this particular constraint. In fact, the proposed constraint captures anomalies of a more general nature than just parts simultaneously belonging to at least two different containers (e.g. parts belonging to the `contents` table of a container and having no `owner` set at all).

Regarding the rule [C2'] above, the necessity of introducing a corresponding explicit WFR can be argued by means of the following example. Let us assume the existence of an XCore



```
MyProjects
  Test1
    Test1
      P1
        C
      P2
        C
```

Constraint Check for Test1

```
<Package Test1>
  <Package Test1> satisfies AllSlotsTypeCorrect
  <Package Test1> satisfies TypeCheck
  <Package Test1> satisfies validOwnerForContents
  <Package P2>
  <Package P1>
    <Package P1> satisfies AllSlotsTypeCorrect
    <Package P1> satisfies TypeCheck
    <Package P1> fails validOwnerForContents : The elements from Set{<Class C>} should have <Package P1> as owner!
    <Class C>
```

Fig. 4.19: XMF-Mosaic screenshot

model consisting of a single empty package named `Test2` (that has been assigned to a global variable `t2`). Furthermore, assume that there is the necessity of creating under `Test2` a hierarchy of three subpackages, say `P1`, `P2`, and `P3`, each included in the previous one. This basic model editing task can be accomplished in XMF by means of the following sequence of commands.

```
p1 := Package("P1");
t2.add(p1);
p2 := Package("P2");
p1.add(p2);
p3 := Package("P3");
p2.add(p3);
```

However, the misuse of `p1` instead of `p3` as the argument of the latter call above has the effect of creating a circular containment between packages `P1` and `P2`, each of them becoming the `owner` of the other. Yet, in the absence of an explicit WFR prohibiting this, a call to `Element::checkConstraints()` on any of them reports no problem at all.

As a solution to this, we propose the XOCL WFR below, which applies to all indexed containers, except for the `Root` namespace (in XMF, `Root` is the global namespace in which everything is contained, itself included).

No `IndexedContainer` *different from the* `Root` *namespace can be owned by one of its parts.*

```
context IndexedContainer
 @Constraint notOwnedByPart
  (self <> Root and self.oclIsKindOf(Contained)) implies
  self.contents.values()->select(v | self.owner = v)->isEmpty()

  fail "This container is owned by each of its parts from " +
        self.contents.values()->select(v | self.owner = v).toString()
 end
```

Listing 4.16: Proposed XOCL WFR for containment constraint C2’

4.4 Summary

Within this chapter, we have approached the topic of *model compilability*, which we have defined as full compliance of a model to the abstract syntax and static semantics of its modeling language. So as to emphasize its significance, the problem has been presented by analogy to program compilability. We argue that the ability to certify models' compilability is the first precondition for attaining the automation goals envisioned by model-driven development. In turn, this relies on the availability of a thorough definition of the static semantics of modeling languages.

We report on a detailed analysis that we have performed with respect to the definition of the static semantics of the UML metamodel and of the MOF, Ecore, and XCore meta-metamodels. This has revealed various types of problems in each case. Our motivation for approaching these issues is given by the fact that a faulty definition of the static semantics of modeling languages seriously affects their correct reuse by instantiation. The problem is even more severe for metamodeling languages, given their increased reuse potential.

We have attributed the encountered problems to the lack of a rigorous conceptual framework supporting an accurate definition of the static semantics of (meta)modeling languages and enabling efficient model compilability checks. As a solution, we have contributed to the creation of such a framework, by identifying a set of underlying principles. Guided by these principles, we have further contributed to the improvement of the static semantics definition of UML/MOF, Ecore, and XCore, by providing a set of OCL WFRs (XOCL in case of XCore) for each of them. The entire set of proposed rules, as well as the tests used for their validation are available at [5]. Within this chapter, we have only presented some relevant examples for each (meta-)metamodel. In case of UML/MOF and XCore, we have approached the rules related to name uniqueness and composition, which are both core (meta)modeling issues. In case of Ecore, we have approached some rules regarding generics, due to both their complexity level and innovative nature (Ecore being the only of the three meta-metamodels supporting generics). All of our proposals have been compared to related work on the topics in question, so as to emphasize their advantages.

For further work we plan to extend the set of added/corrected rules, so as to encompass the whole UML 2.3 metamodel. Moreover, we aim at identifying and formalizing a core set of constraint patterns occurring in the specification of the static semantics of metamodeling languages. Investigating the problems of consistency and redundancy in a given set of constraints (WFRs in particular) is also considered (possibly, by translation into the B language [14] and use of the associated theorem provers).

Chapter 5
Domain Specific Modeling Languages (DSMLs)

In the field of omponent-based software engineering (CBSE), one of the major topics still waiting for a resolution is the lack of component models (either academic or industrial) supporting a full contractual specification of software components - a core requirement for ensuring safe component reuse.

Within this chapter, we report on a contribution intended to set the bases of a framework able to support an appropriate contractual specification of software components, with a special emphasis on semantic contracts. This contribution is described in Section 5.1. Our motivation is made clear in Subsection 5.1.1, followed by a detailed presentation of a domain specific modeling language (DSML) ensuring the backbone of our proposal, in Subsection 5.1.2. The core contribution concerns the method proposed for representing components' semantic contracts within the language metamodel. A modeling example using the proposed language is illustrated in Subsection 5.1.3. The ending subsection, 5.1.4, describes a simulation approach regarding the execution of component services, which relies on the previously proposed method for representing semantic contracts.

5.1 ContractCML - A Contract-Aware Component Modeling Language

5.1.1 Motivation and Related Work

Component-based software development can be seen as encompassing two complementary activities: the development of reusable software components (development *for* reuse) and their assembling into component-based applications (development *by* reuse). The former activity is supposed to deliver new components as black-box entities that encapsulate data and functions and interact with the environment exclusively by means of their provided and required interfaces. Therefore, an effective and safe reuse of such a component would require it to be accompanied by a specification describing precisely its functionality in terms of provisions and requirements, but without entering the details of how this functionality is implemented. Such a specification is to be regarded as a *contract* between a component and its clients [20].

Four contract levels have been identified to apply to software components [20], namely: *basic*, *behavioral*, *synchronization*, and *quality-of-service*. A first level basic contract introduces the so-called *syntactic specification* [37] of a software component; this merely includes signatures of required and provided services. The second level behavioral contract enriches the previous one, by adding a *semantic specification* [37] of services; this comprises a precise functional description of each service, including legal conditions under which it should be invoked (pre-conditions), as well as expected effects of its execution (post-conditions). The availability of such a semantic specification is demanded by all component reuse-related activities, namely *component retrival* - finding a component that serves our needs, *component composition* - integrating it correctly within an assembly, and *component substitution* - replacing a component with another, that provides compatible behavior. The second level contracts specify behavior in terms of individual services, regarded as atomic operations executing in a sequential context. As opposed to this, level three synchronization contracts describe the global behavior of component objects. This includes dependencies between services pertaining to a component, such as sequence, parallelism or shuffle, in a distributed concurrent environment. Finally, level four contracts cover non-functional component properties (e.g. maximum response delay, average response, precision of results) and are usually negotiable.

Although the compulsoriness of a semantic specification for software components is unanimously acknowledged, the only form of specification employed by dedicated industrial component models like EJB, COM, or CCM remains the syntactic one [37]. Some improvements have been brought by academic component models, some of which have introduced facilities for describing component behaviors. The Fractal and SOFA component models, for example, use behavior protocols in this purpose [74]. Nevertheless, this kind of specification rather fits level-three contracts, while missing level-two semantic information. Nothing can be said about the effects of invoking a service from an interface, except for what might be assumed from its own name, or the names and types of its parameters.

As the four types of contracts provide complementary information, a thorough component specification should include them all. Moreover, since we see them as interdependent (accomplishing a certain contract level stands as a precondition for the ones above it), skiping an intermediary level would result in essential loss of information.

As implied by the above-described state of facts, there is the need of a homogeneous framework supporting the specification, validation and evaluation of component contracts at all the four mentioned levels. In this context, within this section we introduce ContractCML (Contract Component Modeling Language), a domain specific modeling language (DSML) that we propose as the backbone of such an approach. ContractCML is a hierarchical component modeling language (as opposed to a flat one), since it allows defining and managing not only primitive components, but also composed ones. But most important, as its name shows, it is a contract-aware component modeling language, allowing the specification of component-related contracts. At the moment, it covers the first two levels, but its extensible architecture facilitates the adding of new levels in a non-invasive way. In order to integrate level two semantic specification capabilities into the language, we have used a Design by Contract approach, complemented by a model weaving technique.

Using Design by Contract in order to specify the semantics of an interface's services is not a novel idea. This principle has entered the object-oriented world due to Bertrand Meyer [58] and the Eiffel language; later, it has been applied to CBSD by Catalysis [39] and UML Components [28] methodologies. In fact, the pattern followed by ContractCML in order to ensure a semantic specification of interfaces has its roots in the proposal made by Cheesman and Daniels in [28]. This Catalysis-derived approach advocates the use of Unified Modeling Language (UML), Object Constraint Language (OCL), and Design by Contract for specifying and implementing components and component applications. However, as acknowledged by its authors, the lack of appropriate component modeling concepts within UML 1.x, as well as the poor OCL support within CASE tools at the time, have been identified as the main difficulties in automating the methodology. Fortunately, today, with the advent of MDE, the state of facts in the field of modeling/constraint languages and their associated tool-support has much improved.

There is a twofold motivation behind our decision of defining a domain specific modeling language for describing components and their contracts. Firstly, since DSMLs are tailored to particular problem domains, the models that use them are easier to understand and manage compared to those created with a general purpose modeling language (e.g. UML). Secondly, our choice enables us to benefit from a powerful tool support. Indeed, EMF [40], GMF [42], oAW [11] and XMF-Mosaic [3] are MDE meta-tools providing rich functionalities. Namely, these meta-tools assist users in specifying, testing and validating DSMLs at all language levels: abstract syntax, concrete syntax, and semantics. Moreover, having a validated DSML, the above-mentioned frameworks provide support in developing dedicated (domain specific) modeling tools. In our case, such a tool would provide functionalities for specifying components and assembling them in a graphical manner, placing a strong emphasis on component interoperability issues.

Component interoperability has been defined as "the ability of two or more components to communicate and cooperate despite differences in their implementation language, the execution environment, or the model abstraction" [32]. Checking component interoperability is an imperative, since it allows system designers to decide whether two components can be interconnected, or whether one component can be safely replaced by another. Since components are defined as black boxes that communicate with the environment only by means of their interfaces, component interoperability turns into an interface compatibility issue. This interface compatibility involves at least two aspects: a syntactic one (signature matching) and a semantic one (specification matching). Given a component modeling language, signature matching can be easily enforced by means of WFRs expressed at an architectural level. Semantic compatibility, however, is more delicate. There are two possible ways to check it: mathematical proof or model simulation. The former can ensure compatibility, but involves advanced mathematical knowledge and requires strong prover tools. The later (similar to program testing) does not guarantee compatibility, but may provide a good probability of it, with less effort. Therefore, following the description of the ContractCML metamodel and its illustration by means of an example, we present an approach for the simulation of ContractCML component services. The approach in question relies on our proposal for representing the semantic contracts and on the use of an executable constraint language.

5.1.2 The ContractCML Metamodel

5.1.2.1 Metamodel Overview

The ContractCML metamodel has been designed in a modular style, starting with basic syntactic component concepts, on which semantic and architectural aspects have been added. This has resulted in a loosely coupled, highly extensible architecture. The metamodel packages, as well as their inter-dependencies, are illustrated in Figure 5.1. We provide an overview of each one's contents in the following, in a natural order imposed by their dependencies.

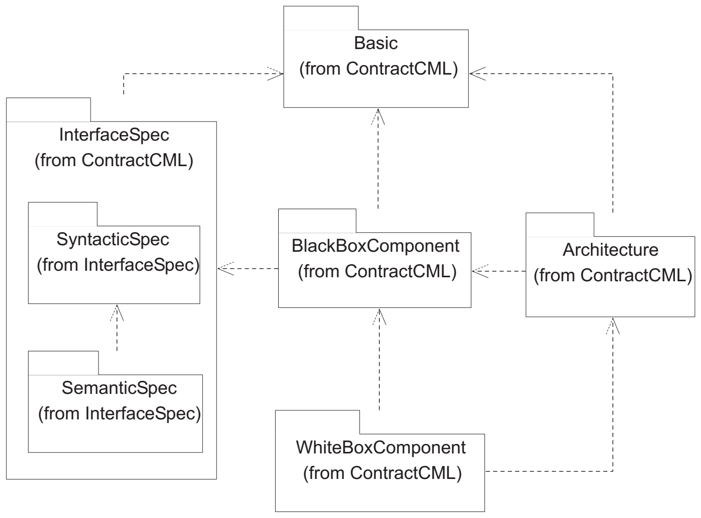


Fig. 5.1: ContractCML metamodel packages

The `Basic` package, on which all the other packages rely, includes elementary modeling concepts (see Figure 5.2). All the metaclasses within it are abstract. Among them, `ModelElement` and its descendant, `NamedElement`, are inherited by all ContractCML metaclasses, and by all metaclasses that are identifiable through a name, respectively. `Type` and `TypedElement`, both derived from `NamedElement`, are also among the basic concepts. A typed model element is directly associated to a single type, as shown by the corresponding multiplicity in Figure 5.2. The sole WFR defined for this package (concerning the validity of names for named elements) is illustrated in Listing 5.1.

```
-- [WFR] Any NamedElement instance should have a non-empty name
context NamedElement
 inv notEmptyName:
  not self.name.oclIsUndefined() and self.name <> ''
```

Listing 5.1: WFRs for `ContractCML::Basic`

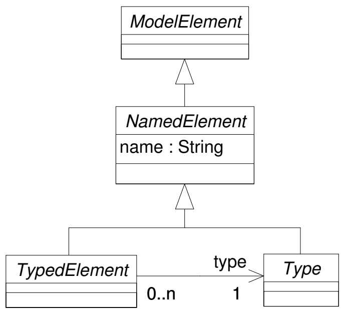


Fig. 5.2: The `ContractCML::Basic` package

Depending only on `Basic`, the `InterfaceSpec` package groups the metaclasses related to the specification of interfaces in two subpackages - syntactic and semantic. Their contents will be detailed in the following subsections.

Relying on the interface specification concepts, the `BlackBoxComponent` package (Figure 5.3) includes metaclasses that offer a client's view on a component. From this per-

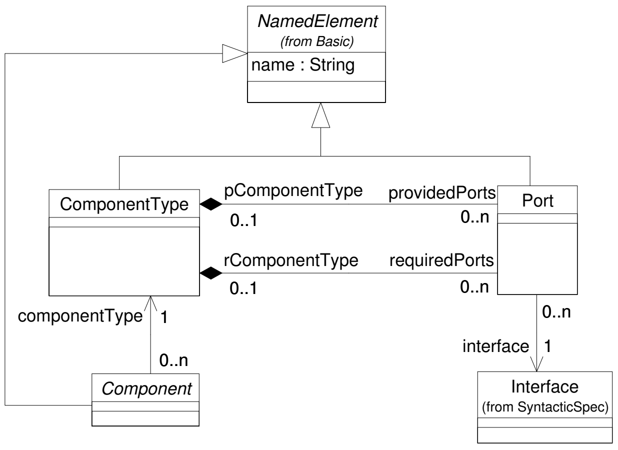


Fig. 5.3: The `ContractCML::BlackBoxComponent` package

spective, each `Component` is identified by a name and has a corresponding `ComponentType`. The latter, which is itself a named element, can be thought of as the collection of all ports provided and required by the component. A `Port` defines an interaction point with the environment and is typed by an interface. A component is always associated to a single component type, but the same component type can be implemented differently by distinct providers, resulting in distinct components. The well-formedness rules applying to the metaclasses in this package, as well as the additional operations defined are given in Listing 5.2.

```
-- [WFR] A port is either required or provided
--       by a certain component type, not both.
context Port
 inv providedXORrequired:
  self.isProvided() xor self.isRequired()

-- [AO] Checks whether a port is provided
context Port::isProvided():Boolean
 body: self.rComponentType.oclIsUndefined()

-- [AO] Checks whether a port is required
context Port::isRequired():Boolean
 body: self.pComponentType.oclIsUndefined()

-- [WFR] A given Interface is provided/required
--       by a single Port of the same ComponentType
context ComponentType
 inv sameInterfProhibited:
  self.allInterfaces()->select(interf:interfacespec::syntacticspec::
   Interface | self.allPorts()->select(p:Port |
   p.interface = interf)->size() <> 1)->size() = 0

-- [AO] All interfaces of a ComponentType instance
context ComponentType::allInterfaces():Set(Interface)
 body: self.allPorts().interface->asSet()

-- [AO] All ports of a ComponentType instance
context ComponentType::allPorts():Set(Port)
 body: self.providedPorts->union(self.requiredPorts)

-- [AO] All interfaces provided by a ComponentType instance
context ComponentType::providedInterfaces():Set(Interface)
 body: self.providedPorts.interface->asSet()

-- [AO] All interfaces required by a ComponentType instance
context ComponentType::requiredInterfaces():Set(Interface)
 body: self.requiredPorts.interface->asSet()
```

Listing 5.2: WFRs and AOs for `ContractCML::BlackBoxComponent`

Having a set of components regarded as black boxes, a client can create component assemblies. The `Architecture` package illustrated in Figure 5.4 bundles concepts used for describing such assemblies. An `Architecture` owns a set of `ComponentInstance`s and a set of `Binding`s between them. A component instance is simply a particular occurrence of a component, to which a name is assigned. Any binding has two `BindingEnd`s, each one pointing to a certain port. All bindings from inside an architecture are of type `AssemblyBinding`; assembly bindings link a required port of one component instance to a compatible provided port of another component instance from within the same architecture. Ports' compatibility is defined in terms of their associated interfaces, and must encompass both syntactic and semantic compatibility. Since both these aspects are covered by our metamodel, the basis for checking component interoperability within an architecture is ensured.

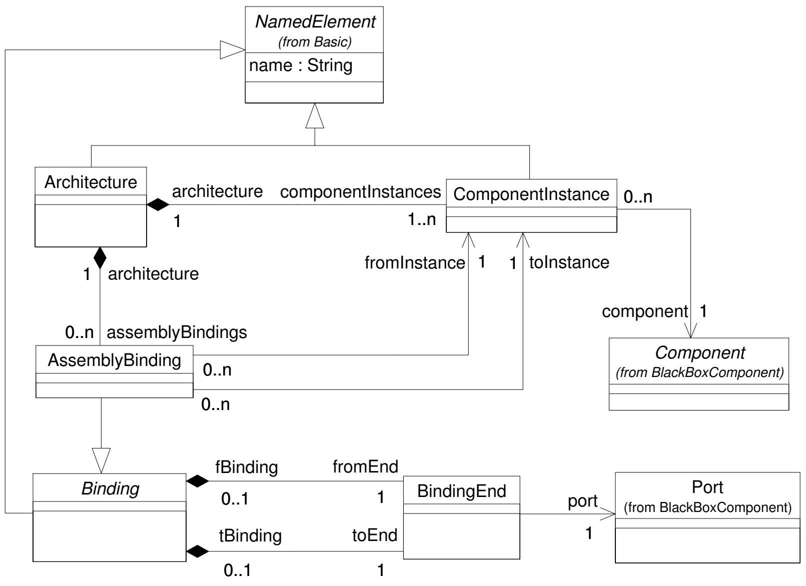


Fig. 5.4: The `ContractCML::Architecture` package

The semantics of binding ends and assembly bindings are captured by the WFRs in Listing 5.3.

```
-- [WFR] A BindingEnd is either from or to, at least one, not both
context BindingEnd
 inv fromXORto:
  self.fBinding.oclIsUndefined() xor self.tBinding.oclIsUndefined()

-- [WFR] The two component instances linked by an assembly binding
--       should belong to the same architecture as the binding
context AssemblyBinding
 inv withinArchitecture:
  self.fromInstance.architecture = self.architecture and
  self.toInstance.architecture = self.architecture

-- [WFR] Core semantics of assembly-type bindings
context AssemblyBinding
 inv validAssemblyBinding:
  self.fromInstance.requiredPorts()->includes(self.fromEnd.port) and
  self.toInstance.providedPorts()->includes(self.toEnd.port)

-- [WFR] An assembly binding cannot be cyclic
--       (the linked component instances should be different)
context AssemblyBinding
 inv nonCyclicAssembly:
  self.fromInstance <> self.toInstance

-- [AO] Provisions of a certain component instance
```

```
context ComponentInstance::providedPorts:Set(Port)
 body: self.component.componentType.providedPorts

-- [AO] Requirements of a certain component instance
context ComponentInstance::requiredPorts:Set(Port)
 body: self.component.componentType.requiredPorts
```

Listing 5.3: WFRs and AOs for `ContractCML::Architecture`

Having the black box and architectural concepts defined, the `WhiteBoxComponent` package overpasses the client's perspective and offers a deeper, architectural view on components (see Figure 5.5). The abstract `Component` metaclass defined within `BlackBoxCompo-`

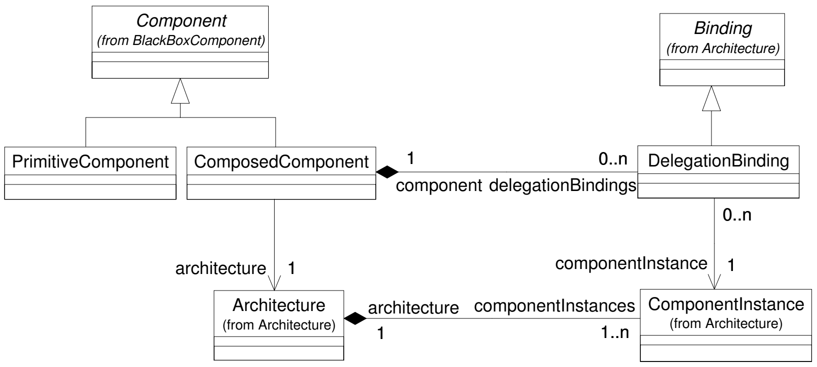


Fig. 5.5: The `ContractCML::WhiteBoxComponent` package

`nent` is specialized into `PrimitiveComponent` and `ComposedComponent`. A composed component has an associated architecture and owns a collection of `DelegationBinding`s. A delegation binding always connects ports of the same nature, either provided or required. It may delegate from a provided port of a composed component to a port provided by one of the component instances owned by its architecture or from a required port of a component instance within its architecture to a port required by the enclosing composed component. The semantics of delegation bindings, which we have just expressed in natural language, is enclosed by the WFRs in Listing 5.4. As opposed to a composed component, a primitive one has no associated architecture; it is either implemented directly in a programming language or its architecture is simply unknown.

```
-- [WFR] Any delegation binding owned by a composed component links
--       the owner composed component to one of the component
--       instances within its architecture
context DelegationBinding
 inv withinComposedComponent:
  self.component.architecture.componentInstances->includes(
                                  self.componentInstance)

-- [WFR] Core semantics of delegation-type bindings
context DelegationBinding
```

```
inv validDelegationBinding:
 ( self.component.componentType.providedPorts->includes(
                                 self.fromEnd.port) and
   self.componentInstance.providedPorts()->includes(self.toEnd.port)
 ) or
 (
   self.componentInstance.requiredPorts()->includes(
                                 self.fromEnd.port) and
   self.component.componentType.requiredPorts->includes(
                                 self.toEnd.port)
 )
```

Listing 5.4: WFRs and AOs for `ContractCML::WhiteBoxComponent`

5.1.2.2 Basic Contracts. Syntactic Specification of Interfaces

The elements within `SyntacticSpec`, one of the two `InterfaceSpec` packages, describe component interfaces from a syntactic perspective. By means of these elements, ContractCML is allowed to express basic contracts. Such contracts stand at the basis of component type checking and component interoperability verifications.

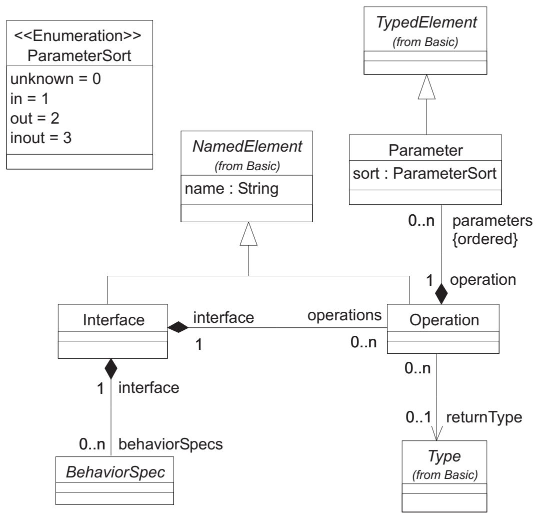


Fig. 5.6: Syntactic specification of interfaces

As suggested by Figure 5.6, an `Interface` is identified by a name and consists of a collection of operations, each `Operation` having itself a name, an ordered list of parameters and, possibly, a returned type. As a typed model element, a `Parameter` has an associated name and type; in addition, its `sort` attribute specifies the dataflow direction: input, output, or both. The `unknown` enumeration literal is meant to support partial or incomplete specification, covering the case when such sort information is not available. The syntactic compatibility of operations and interfaces (in terms of exact matching) can be ensured by the use of the additional operations given in Listing 5.5.

```
-- [AO] Checking whether two operation signatures match exactly
context Operation::exactSignatureMatch(op:Operation):Boolean
 body:  self.name = op.name and
  self.returnType = op.returnType and
  self.parameters->size() = op.parameters->size() and
  Sequence{1..(self.parameters->size())}->reject(i:Integer |
   self.parameters->at(i).type = op.parameters->at(i).type and
   self.parameters->at(i).sort = op.parameters->at(i).sort)->isEmpty()

-- [AO] Checking whether two interface signatures match exactly
context Interface::exactSignatureMatch(interf:Interface):Boolean
 body:  self.name = interf.name and
  self.operations->size() = interf.operations->size() and
  self.operations->reject(op:Operation | interf.operations->
           exists(o:Operation | o.exactSignatureMatch(op)))->isEmpty()
```

Listing 5.5: AOs for checking the syntactic compatibility of operations and interfaces

As already mentioned previously, publishing a syntactic specification of all provided and required interfaces of a component is mandatory in order to make a system that uses it work. However, in order to ensure that it works right, delivering the intended functionality, behavioral information regarding the respective component should be also provided. Such behavioral information may be either interface-specific (capturing aspects regarding the functionality of an interface independently of other interfaces belonging to the same component) or global (involving several component interfaces). The `BehaviorSpec` metaclass abstracts such interface-specific behavioral information. Concrete behavioral specifications (either at level two - expressing the semantics of services by means of pre/post-condition pairs, or at level three - constraining the order of service calls), should subclass `BehaviorSpec` and be described in separate packages. This approach ensures a good metamodel manageability and extensibility.

5.1.2.3 Behavioral Contracts. Semantic Specification of Interfaces

Built on `SyntacticSpec`, the `SemanticSpec` package adds second-level contract support to our component language. It follows a Design by Contract (DBC) approach in order to provide a semantic specification of interfaces. The metaclasses represented in Figure 5.7 and the relationships among them have been derived from the specification concepts introduced in [28].

A `DBCSpec` is a kind of behavioral specification that may be attached to an interface. It consists of a collection of operation specifications (one for each service/operation exposed by the interface), together with an interface information model (or state model). As defined in [28], the information model corresponding to an interface is an abstraction of that part of a component's state that affects or may be affected by the execution of operations in the interface. It does not expose implementation details. It is merely an abstraction that helps in defining operations' behavior.

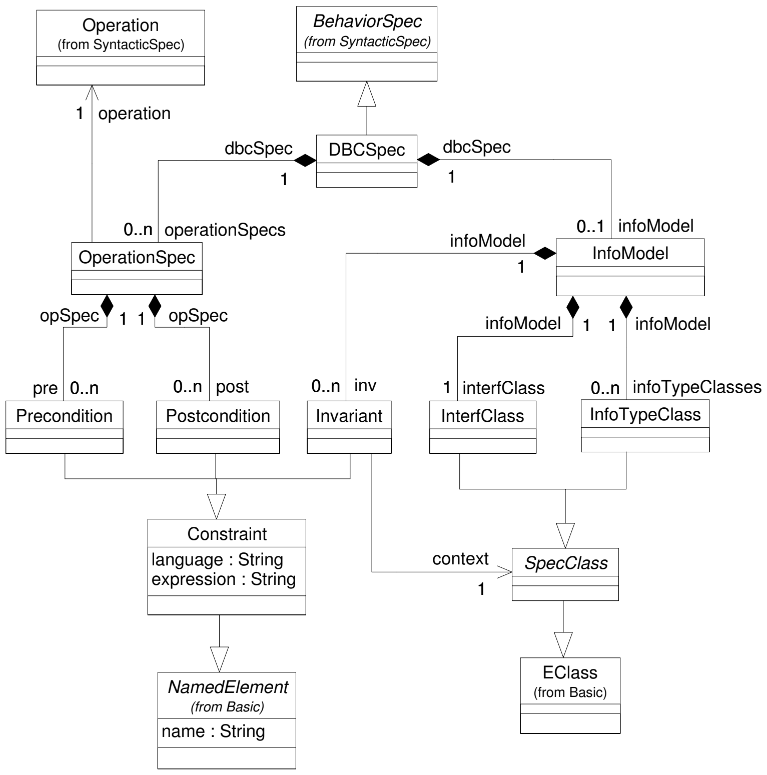


Fig. 5.7: Semantic specification of interfaces

The `OperationSpec` metaclass allows representing the behavior of its associated `Operation`, stated in terms of pre/post-condition pairs. The correctness of such an association is captured by the WFR From Listing 5.6. A `Precondition` is a predicate expressed in terms of the input parameters and the state model; a `Postcondition` is also a predicate, involving both input and output parameters, as well as the state just before the invocation and immediately after. Preconditions and postconditions are special types of `Constraint`s. Such a constraint is identified by a name and consists of an expression stated in a certain language, either natural or formal. Since we want these constraints to be managed by tools, our approach uses a formal language to specify them, namely OCL.

```
-- [WFR] Any OperationSpec from within a DBCSpec attached to
--       an Interface should specify the semantics of an operation
--       that actually pertains to that interface
context OperationSpec
 inv correctOpSpec:
```

```
let ops:Set(Operation)=self.dbcSpec.interface.operations in
ops->includes(self.operation)
```

Listing 5.6: A WFR for the `SemanticSpec` package

At this point, let us recall some details regarding the approach taken in [28] for the semantic specification of interfaces, so as to motivate some of the choices we have made at this level. The methodology above-mentioned is UML-based. An interface specification consists of the interface itself, its information model, its operations' specifications (signatures and pre/post-conditions), plus any additional invariants on the information model. The interface is represented as a class, stereotyped as `<<interface type>>` and having operations corresponding to all interface services. The information model is actually a class model (classes and relationships between them), all classes within it having the `<<info type>>` stereotype. Moreover, the interface class has a composition relationship with at least one of the information model classes. This approach allows expressing constraints (either operations' pre/post-conditions or information model invariants) in a very natural way.

Transposing these concepts into our metamodel raises some issues. First of all, as already stressed, ContractCML is a domain specific modeling language and therefore, apart from a few basic general modeling concepts, its metamodel includes only component-related concepts. However, in order to be able to represent interface information models as described above, we need appropriate support at the metamodel level (namely, metaclasses corresponding to classes, their structural and behavioral features, relationships, etc.). Defining our own metamodel package to use in this purpose would not be a proficient solution, since there are such metamodels available, which could be reused. Second of all, there is the problem of linking a ContractCML `Interface` instance to an UML-like `Class` instance by means of a composition relationship, due to the fact that the two metaclasses belong to different metamodels. The following subsection details the approach that we have taken as an answer to these problems.

5.1.2.4 A Model Weaving Approach to Representing Information Models

We have taken a model weaving approach in order to provide a solution to the previously mentioned issues. Model weaving can be regarded as a special kind of model transformation, the latter being generically defined as the process of creating an output model based on one or more input models [49]. By model weaving, different (but related) models can be composed into a consistent whole. Two types of weaving can be distinguished: symmetric and asymmetric. An asymmetric weaving works with a base model and one or several aspect models which it integrates into the base in a user controllable way, as opposed to a symmetric one, where there is no designated base model [49].

XWeave [49] is an asymmetric model weaving tool based on the EMF Ecore meta-metamodel. It is able to work both at metamodel and model level; in the former case, it weaves metamodels which are instances of Ecore, in the later, it weaves models which are instances of Ecore metamodels. Weaving is based either on name matching in the base and aspect metamodels, or on explicit pointcut definitions using the oAW expression language.

Conceptually, providing ContractCML with information models' representation capabilities may be done by taking it as the base metamodel and integrating it with an aspect metamodel from which class models can be instantiated. We have considered Ecore itself as the aspect, since it satisfies our constraints: it is an Ecore-based metamodel (thus enabling us to use XWeave) and its instances are class models. Moreover, the ContractCML metamodel can be itself represented as an instance of Ecore, making the integration using XWeave feasible.

In order to benefit from the weaving as intended, we have considered the following representation for information models within the ContractCML metamodel (see Figure 5.7): Similar to [28], an `InfoModel` consists of a collection of classes. These are all instances of `InfoTypeClass`, except for one, which is an `InterfClass` instance. The latter is the class corresponding to the interface itself and has at least a composition relationship with an information-type class. We will refer to it as *the main class of the information model* in the following. All the classes that compose the information model are used exclusively for interface specification purposes, fact denoted by their specialization from the abstract `SpecClass`. In turn, `SpecClass` inherits from `EClass`, which is a metaclass that we have added within the `Basic` package (see Figure 5.8 (b)). An information model may also contain a number of invariant constraints, such an `Invariant` being always written in the context of a specification-type class.

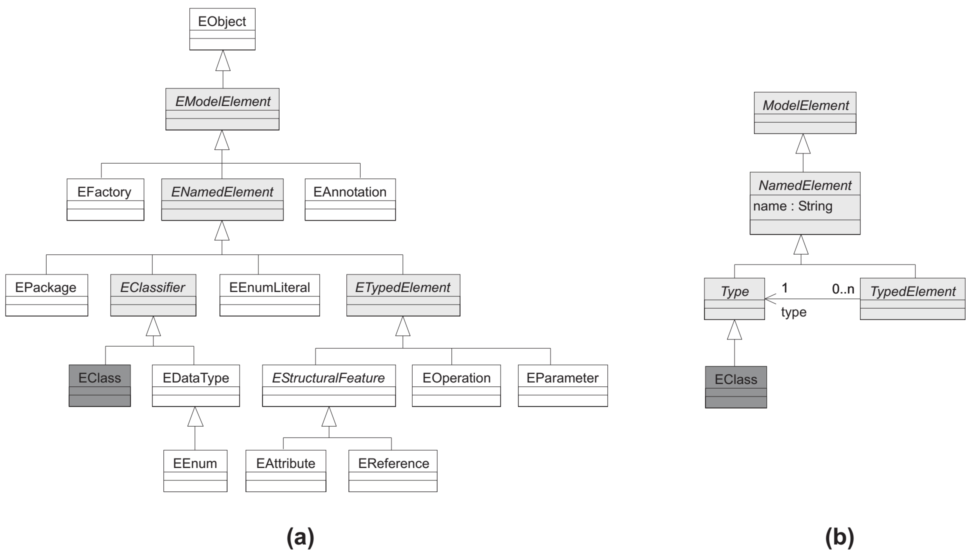


Fig. 5.8: `Ecore` (a) and `ContractCML::Basic` (b) concepts correspondence

The `EClass` introduced within the `Basic` package was thought as a means to achieve model weaving with Ecore, following a name matching weaving algorithm. However, since the `Basic` package in its entirety contains concepts which have Ecore semantic equivalents, a correct weaving should take all these equivalences into account. All metaclasses from `Basic`, together with their Ecore correspondents are highlighted in Figure 5.8. The equivalent concepts most often differ by a letter (the initial “E” from Ecore), except from `EClass` and `Type`, the latter corresponding to the Ecore’s `EClassifier`. Unfortunately, the current XWeave release does not include functionalities for expressing links between equivalent concepts and weavings based on those links. Instead, weaving is based on exact name matching. As a consequence, we had to rename the light-grayed Ecore metaclasses, so as to match their ContractCML correspondents. Of course, this renaming did not affect Ecore itself, but solely a copy that we have used for weaving purposes. The oAW workflow used to perform the weaving is given in Listing 5.7.

```
<workflow>
 <!-- set up EMF for standalone execution -->
 <bean class="org.eclipse.mwe.emf.StandaloneSetup">
  <platformUri value=".."/>
 </bean>

 <!-- load base model and store it in slot 'base' -->
 <component class="org.eclipse.mwe.emf.Reader">
  <useSingleGlobalResourceSet value="true"/>
  <uri value="platform:/resource/ContractCML/src/test/
                                   ContractCML.ecore"/>
  <modelSlot value="base"/>
 </component>

 <!-- call a workflow that reads the aspect model and stores it -->
 <!-- in slot 'aspect', then wove the aspect into the base -->
 <cartridge file="org::openarchitectureware::util::xweave::
                                                    wf-weave.oaw"
  aspectFile="platform:/resource/ContractCML/src/test/Ecore.ecore"
  baseModelSlot="base"
  overwrite="false"/>

 <!-- write the woven model to file -->
 <component class="org.eclipse.mwe.emf.Writer">
  <uri value="platform:/resource/ContractCML/src/test/
                                  ContractCML.ecore"/>
  <useSingleGlobalResourceSet value="true"/>
  <modelSlot value="base"/>
  <cloneSlotContents value="true"/>
 </component>
</workflow>
```

Listing 5.7: oAW wokflow used to perform the weaving

5.1.3 ContractCML Modeling Example

So as to provide a modeling example using ContractCML, let us consider a simplified variant of the hotel reservation system case study used in [28] and [32]. This offers the possibility to show some concrete syntax elements of our language, as well as to emphasize the advantages of having a component modeling language that includes semantic specification facilities.

Figure 5.9 shows two primitive components, `ReservationSystem` and `Hotel`, which have the `ReservationMgr` and `HotelMgr` component types, respectively. The `Reser-`

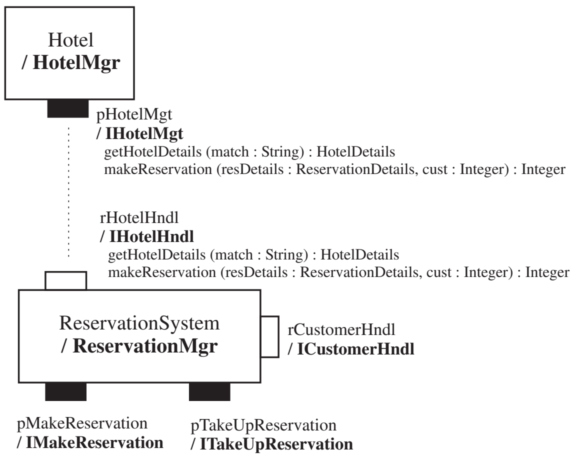

Hotel
/ HotelMgr
pHotelMgt
/ IHotelMgt
getHotelDetails (match : String) : HotelDetails
makeReservation (resDetails : ReservationDetails, cust : Integer) : Integer
rHotelHndl
/ IHotelHndl
getHotelDetails (match : String) : HotelDetails
makeReservation (resDetails : ReservationDetails, cust : Integer) : Integer
ReservationSystem
/ ReservationMgr
rCustomerHndl
/ ICustomerHndl
pMakeReservation
/ IMakeReservation
pTakeUpReservation
/ ITakeUpReservation

Fig. 5.9: ContractCML modeling example

`vationMgr` component type provides, by means of two ports named `pMakeReservation` and `pTakeUpReservation`, the `IMakeReservation` and `ITakeUpReservation` interfaces. In order to be able to provide the services exposed by these interfaces, it has two requirements on the environment, by means of two required ports, `rHotelHndl` and `rCustomerHndl`, of type `IHotelHndl` and `ICustomerHndl`.

We have considered two possible services contained in the `IHotelHndl` interface required by the reservation system component: one of them returns details regarding the hotel whose name is given as a parameter, while the other registers a reservation, based on some details and a customer identifier. The `HotelMgr` component type provides an `IHotelMgt` interface, having the same operation signatures as those required by the reservation system component. In this context, the question is whether the `IHotelMgt` interface can safely replace `IHotelHndl` within a component assembly. As shown by the operation signatures, syntactic compatibility is ensured. What remains to be studied then is the semantic compatibility of these interfaces, based on their information models and operation specifications.

The information model for `IHotelHndl`, as well as the format of structured data used by its operations (`ReservationDetails` and `HotelDetails`), are depicted in Figure 5.10. The specification for the `makeReservation` service of `IHotelHndl` is expressed based on this model. It consists of a precondition stating that the hotel for which a reservation is required exists, and a postcondition that guarantees that a new reservation having the provided information has been registered. The corresponding OCL expressions, written in the context of `IHotelHndl` and based on its information model, are given below.

```
--IHotelHndl’s makeReservation precondition
hotels->exists(h | h.id = resDetails.hotel)
--IHotelHndl’s makeReservation postcondition
```

```
(reservs - reservs@pre)->size = 1 and
let r:Reservation = (reservs - reservs@pre)->asSequence()->first() in
r.hotel.id = resDetails.hotel and r.customer.id = cust and
r.startDate = resDetails.startDate and
r.endDate = resDetails.endDate and
r.claimed = false and result = r.resRef
```

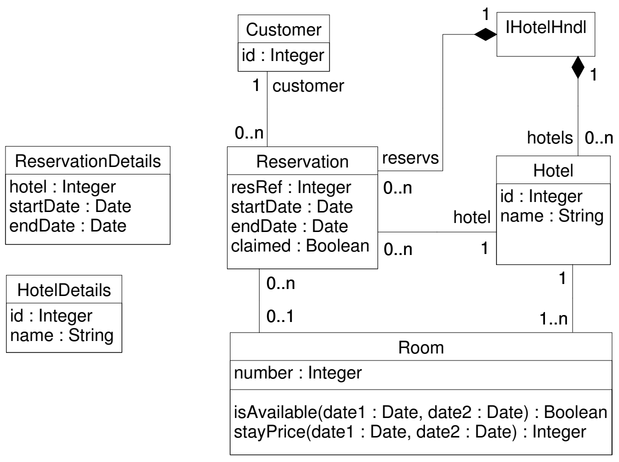


Fig. 5.10: IHotelHndl information model

The interface provided by `HotelMgr`, `IHotelMgt`, takes into account hotels with different kinds of rooms. This is shown by its information model in Figure 5.11, which includes a `RoomType` class, togheter with the corresponding associations. The remaining information-type classes and associations are identical to those of the `IHotelHndl` model. With respect to the data used by the operation signatures, `HotelDetails` has the same structure, while `ReservationDetails` is enriched so as to store room type information.

The precondition of `makeReservation` in `IHotelMgt` has the same expression as its `IHotelHndl` correspondent, but the postcondition is different, since it has to take into account room types. Its OCL expression is given below.

```
--IHotelMgt's makeReservation postcondition
(reservs - reservs@pre)->size = 1 and
let r:Reservation = (reservs - reservs@pre)->asSequence()->first() in
r.hotel.id = resDetails.hotel and r.customer.id = cust and
r.startDate = resDetails.startDate and
r.endDate = resDetails.endDate and
r.roomType.name = resDetails.roomType and
r.claimed = false and result = r.resRef
```

The `getHotelDetails` operation is specified identically for both interfaces.

Based on the above interface specifications, a reasoning regarding their semantic compatibility can be conducted. Can `IHotelMgt` safely replace `IHotelHndl`? The answer has to

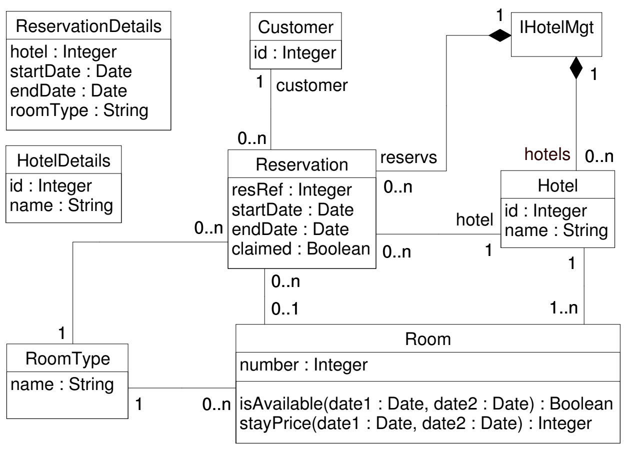


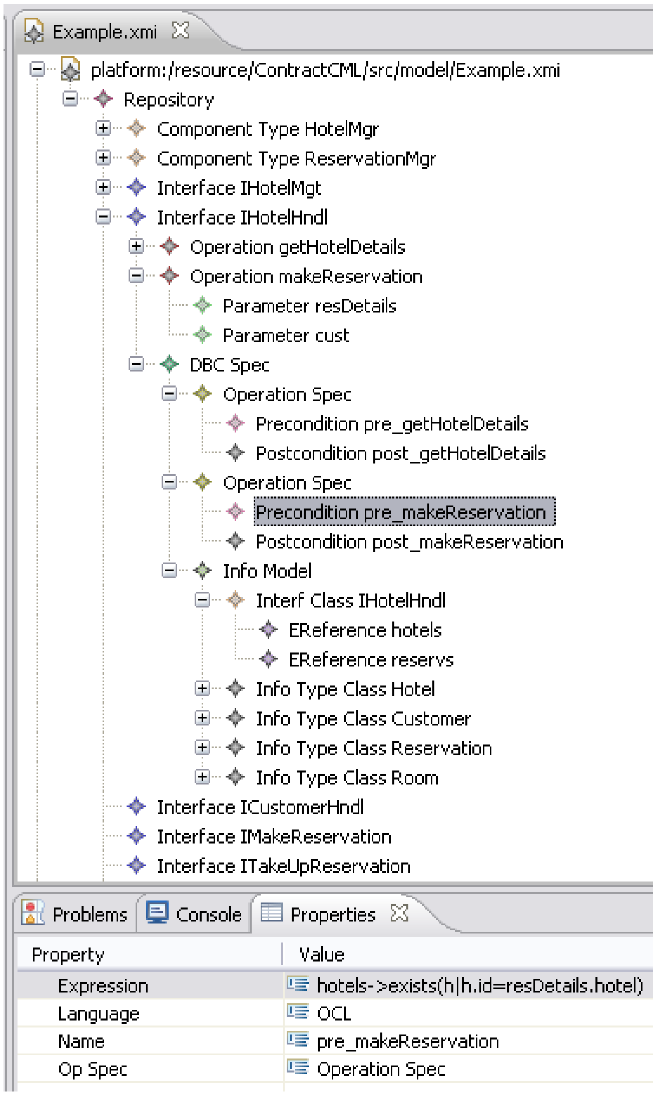
Fig. 5.11: IHotelMgt information model

take into account both information models and operation specifications. By comparing these, the following conclusions can be derived: The `IHotelMgt` information model its richer than its `IHotelHndl` correspondent. Moreover, the `IHotelMgt makeReservation` postcondition is stronger than the one from `IHotelHndl`, the preconditions being equivalent. Therefore, since the `getHotelDetails` specifications are identical, one can easily infer that the provided interface offers more than the required one, thus the linking of the two components within an architecture is legal. This is a case of “plug-in” interface compatibility, that could be established by a dedicated theorem prover tool. Details regarding the theoretical aspects of interface compatibility can be found in [32].

A tree-like representation of our example model, using the EMF tree editor, is shown in Figure 5.12. This editor only displays the name of model elements, the remaining properties being accessible by means of the Properties tab.

5.1.4 Component Models’ Simulation Using ContractCML

As motivated earlier, ContractCML was thought as the backbone of a framework meant to support a rigorous specification of software components, as well as their semantically correct assembling into component-based applications. Through the present subsection, we intend to further contribute to such a framework, by proposing a simulation method for ContractCML component services. As emphasized previously, adding simulation capabilities to our language is highly beneficial, since it allows us to test component assemblies and perform component interoperability checks early in a system’s lifecycle. The proposed method relies on our approach for representing components’ semantic contracts. The simulation takes place in the context of the XMF-Mosaic framework, being based on an XCore representation of the ContractCML metamodel and the use of XOCL.



Example.xmi

```
platform:/resource/ContractCML/src/model/Example.xmi
  Repository
    Component Type HotelMgr
    Component Type ReservationMgr
    Interface IHotelMgt
    Interface IHotelHndl
      Operation getHotelDetails
      Operation makeReservation
        Parameter resDetails
        Parameter cust
      DBC Spec
        Operation Spec
          Precondition pre_getHotelDetails
          Postcondition post_getHotelDetails
        Operation Spec
          Precondition pre_makeReservation
          Postcondition post_makeReservation
        Info Model
          Interf Class IHotelHndl
            EReference hotels
            EReference reservs
          Info Type Class Hotel
          Info Type Class Customer
          Info Type Class Reservation
          Info Type Class Room
    Interface ICustomerHndl
    Interface IMakeReservation
    Interface ITakeUpReservation
```

Problems | Console | Properties

Property	Value
Expression	hotels->exists(h|h.id=resDetails.hotel)
Language	OCL
Name	pre_makeReservation
Op Spec	Operation Spec

Fig. 5.12: EMF tree-editor screenshot

5.1.4.1 Proposed Simulation Method

As emphasized by [28], the main purpose of an interface information model is to support the specification of level two usage contracts for software components. However, by employing an executable OCL-like language, such as XOCL, it becomes possible to use the information model for simulation purposes as well. Namely, we may simulate a service belonging to the provided interface of a component by defining an XOCL body for its homonymous operation located in the main class of the interface information model, and then executing it. Simulation of a provided interface service will be thus performed in terms of the interface's information model. Still, this basic technique only works for components that have no requirements on the environment. In order to realistically simulate those having both provisions and requirements, it has become necessary to enrich the ContractCML metamodel, enabling it to represent not only usage contracts, but also *realization contracts*.

Unlike usage contracts, which are for clients, realization contracts are for the component realizers/implementors [28]. A realization contract attached to a component type contains information regarding the way in which its provided services should be designed in terms of the required ones. In [28], these interactions are described using UML collaboration diagrams. In ContractCML, we allow expressing such contracts by means of a special type of class, called `RealizationClass`, an instance of which may be attached to a component type (Figure 5.13).

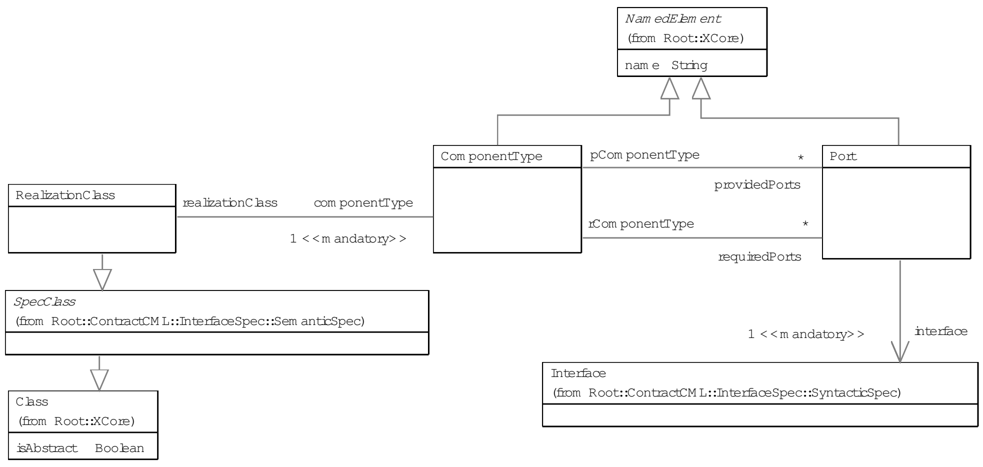


Fig. 5.13: ContractCML support for realization contracts

Any `RealizationClass` instance attached to a component type has to fulfill two mandatory constraints, both of them being formalized as metamodel WFRs (see Listing 5.8):

1. *it has to inherit all main classes from the information models of interfaces exposed through the component type's provided ports*, and
2. *it has to hold a reference to each main class from the information model of each required interface*.

```
context RealizationClass
 @Constraint rightInherit
  let interfClasses = Set{} in
   @For p in self.componentType.providedPorts do
    let interfClass = p.interface.behaviorSpecs->select(s |
        s.isKindOf(ContractCML::InterfaceSpec::SemanticSpec::
```

```
        DBCSpec))->sel.infoModel.interfClass in
     interfClasses := interfClasses->including(interfClass)
    end
   end;
   interfClasses->reject(c | self.parents->includes(c))->isEmpty()
  end
 end

context RealizationClass
 @Constraint rightConfig
  let reqInterfClasses = Set{} in
   @For p in self.componentType.requiredPorts do
     let reqInterfClass = p.interface.behaviorSpecs->select(s |
         s.isKindOf(ContractCML::InterfaceSpec::SemanticSpec::
         DBCSpec))->sel.infoModel.interfClass in
       reqInterfClasses := reqInterfClasses->including(reqInterfClass)
     end
   end;
   reqInterfClasses->reject(c | self.attributes->exists(a |
                                a.type = c))->isEmpty()
  end
 end
```

Listing 5.8: XOCL WFRs applying to RealizationClass

The realization contracts themselves are then expressed by overriding the inherited methods (which perform simulation in terms of the interface information models) such that they delegate calls to the owned references. In case a particular component type has no required interfaces (acting as a leaf in a component architecture), then no overriding is needed, the simulation of a provided service being performed at the level of the information model attached to the interface to which the service belongs.

Apart from representing realization contracts, we have added a `Simulator` class at the metamodel level. In order to simulate any of the services provided by a particular component instance, it is necessary to create such a `Simulator` object. It is assumed that the component instance is part of an architecture in which its requirements are provided by other component instances and so on. The `Simulator` constructor takes the component instance as an argument, which it uses in order to instantiate what we have called a `simulationBase`. The latter is, in fact, a completely and recursively configured instance of the `RealizationClass` corresponding to the argument's component type. We give in the following the XOCL body of the `Simulator` operation returning the simulation base. All the navigations involved can be tracked in figures 5.13 and 5.4.

```
@Operation getSimulationBase(inst:ComponentInstance):XCore::Element
  let arch = inst.architecture;
      compType = inst.component.componentType then
      cls = compType.realizationClass then res = cls()
  in @For reqPort in compType.requiredPorts do
       let portName = reqPort.name;
           binding = arch.assemblyBindings->select(b |
             b.fromEnd.port = reqPort and
             b.fromInstance = inst)->sel then
           instance = binding.toInstance then
           reqInstance = getSimulationBase(instance)
       in res.set(portName,reqInstance)
       end
     end;
     res
  end
end
```

Having such a base configured, simulation of a service provided by the component instance is achieved by calling the `simulate` operation on its corresponding `Simulator` instance. The service itself, as well as the sequence of values for its arguments should be passed as parameters.

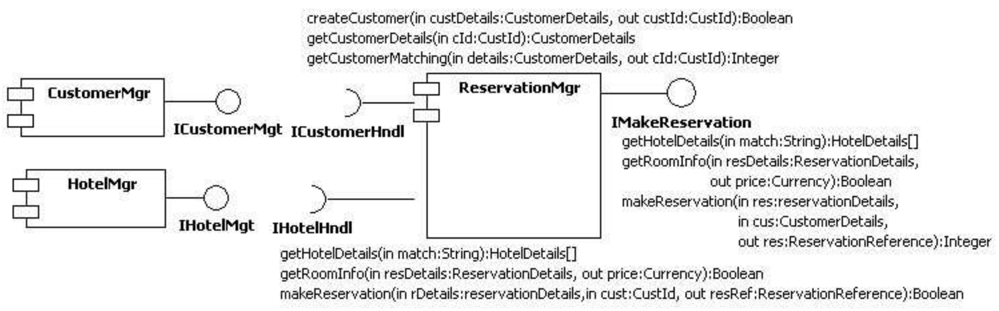
```
@Operation simulate(service:Service,
                    arguments:Seq(Element)):XCore::Element
  let cls = self.simulationBase.of() then
      op = cls.allOperations()->select(o |
        o.name.asSymbol() = service.name.asSymbol())->sel
  in op.invoke(self.simulationBase,arguments,null)
  end
end
```

5.1.4.2 Validation

The proposed approach has been validated using an extended version of the Reservation System case study introduced in Subsection 5.1.3 (see Figure 5.14). It involves a system-level component, `ReservationMgr`, and two business-level components, `CustomerMgr` and `HotelMgr`. In order to deliver the services from its provided interface `IMakeReservation`, the system component requires two other interfaces, namely `ICustomerHndl` and `IHotelHndl`. The `CustomerMgr` business component provides an `ICustomerMgt` interface, having services identical in signature to those owned by `ICustomerHndl`, while `HotelMgr` offers an `IHotelMgt` interface, whose service signatures match exactly those of `IHotelHndl`.

Our aim is to check whether the three components can interoperate correctly, namely whether `ICustomerMgt` can be safely used instead `ICustomerHndl` and `IHotelMgt` instead of `IHotelHndl`, respectively. As mentioned above, syntactic compatibility of the corresponding interfaces is ensured. We assume that each of the five interfaces (irrespective of it being provided or required) has an associated semantic usage contract describing precisely the meaning of its services. In addition, we require that `ReservationMgr` makes available a realization contract showing the interactions between its provided and required services. With these assumptions, the interoperability check is performed by assembling the three components into an architecture, simulating the services provided by the system component, and analyzing the outputed results with respect to the semantics specified for those services (i.e. the expected results).

We further summarize the undertaken modeling steps. The Reservation System has been modeled using an XCore-based version of ContractCML, within the XMF-Mosaic framework. Each interface of the system corresponds to a ContractCML `Interface` object, for



```
createCustomer(in custDetails:CustomerDetails, out custId:CustId):Boolean
getCustomerDetails(in cId:CustId):CustomerDetails
getCustomerMatching(in details:CustomerDetails, out cId:CustId):Integer
```

```
getHotelDetails(in match:String):HotelDetails[]
getRoomInfo(in resDetails:ReservationDetails,
            out price:Currency):Boolean
makeReservation(in res:reservationDetails,
                in cus:CustomerDetails,
                out res:ReservationReference):Integer
```

```
getHotelDetails(in match:String):HotelDetails[]
getRoomInfo(in resDetails:ReservationDetails, out price:Currency):Boolean
makeReservation(in rDetails:reservationDetails,in cust:CustId, out resRef:ReservationReference):Boolean
```

Fig. 5.14: Reservation System case study

which we have defined both a syntactic and a semantic specification. The former declares the services composing the interface, together with their arguments and return types, while the later describes the semantics of these services with the aid of an interface information model. The information model corresponding to `ICustomerMgt` is depicted in Figure 5.15.

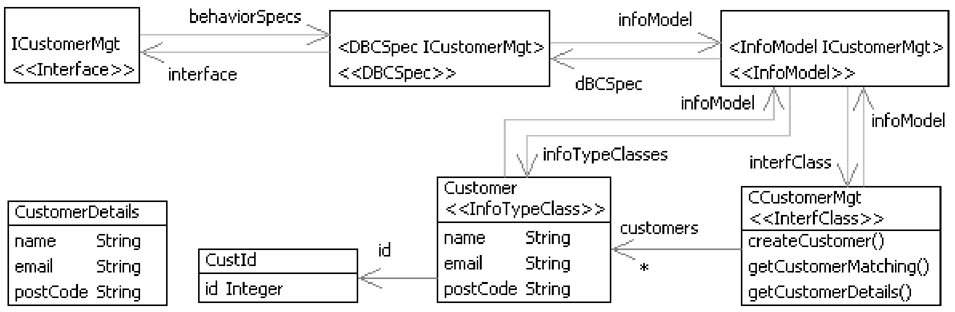


Fig. 5.15: Interface information model for `ICustomerMgt`

The main class of this information model is `CCustomerMgt`, which owns an operation for each `ICustomerMgt` interface service. This way, the service's semantics can be expressed by defining an XOCL body for its corresponding operation. The XOCL statements composing the body change/query the information model, thus abstracting/simulating the way in which the real component's state would be affected by the execution of that service from the interface. (Except for the operation bodies, it should be possible to automate the creation of main classes of information models). Below, we list the body of the `createCustomer()` operation, which dynamically (at simulation-time) creates a new `Customer` object based on provided customer details, adds it to the `customers` collection owned by `CCustomerMgt`, and returns its customer id.

```
@Operation createCustomer(custDetails:CustomerDetails,
                          custId:CustId):Boolean
 if custDetails.name=null or custDetails.email=null or
    custDetails.postCode=null
 then false
 else let newCust = Customer(); cid = CustId()
      in newCust.name := custDetails.name;
         newCust.email := custDetails.email;
         newCust.postCode := custDetails.postCode; cid.id := 0;
         if not self.customers = null
         then @For c in self.customers do
                if c.id.id > cid.id then cid.id := c.id.id end
              end;
              cid.id := cid.id + 1; newCust.id := cid;
              self.customers := self.customers->including(newCust)
         else
           cid.id := 1; newCust.id := cid;
           self.customers := Set{newCust}
         end;
         custId.id := cid.id; true
      end
 end
end
```

For each of the components in Figure 5.14, we have created a corresponding ContractCML `ComponentType` instance. A `ComponentType` is defined by its provided and required ports, a port being typed by an interface. For each `componentType`, we have modeled an associated realization contract, expressed by means of a `RealizationClass` instance. The rules obeyed by such a class have been stated in the previous section (see Listing 5.8). Similar to the main classes of information models, realization classes can also be automatically generated (except for the operation bodies). The realization class corresponding to the system component is shown in Figure 5.16.

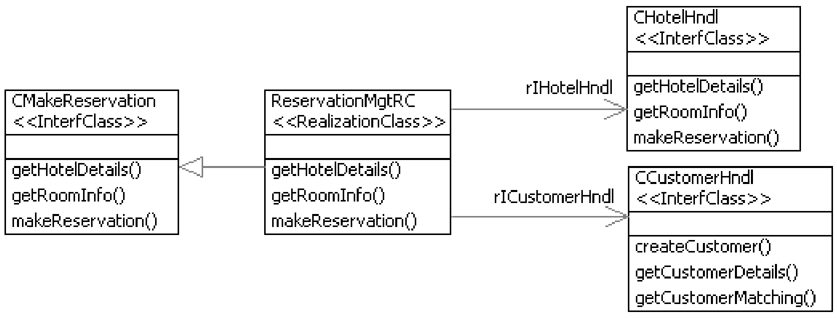


Fig. 5.16: Realization class for the `ReservationMgt` component type

The `makeReservation` operation from `CMakeReservation` (main class of the `IMakeReservation` information model) has been defined to simulate performing a reservation by acting solely on the `IMakeReservation` information model. Inside `CReservationMgtRC` (realization class for `ReservationMgt` component type), this method is overriden, so that its functionality is accomplished by delegating some of the tasks to the required interfaces. This is shown by the listing below. The overridings for `getHotelDetails()` and `getRoomInfo()` are even simpler, delegating directly to the homonymous methods from the required `IHotelHndl` interface.

```
@Operation makeReservation(res:ReservationDetails,cus:CustomerDetails,
                           ref:ReservationReference):Integer
 let custId = CustId(); resRef = ReservationReference() then
  r = rICustomerHndl.getCustomerMatching(cus,custId) in
  if r = 2 then format(stdout,"Reservation failed!
                               Give customer postal code!");1
  else if r = 1 then rICustomerHndl.createCustomer(cus,custId) end;
       if rIHotelHndl.makeReservation(res,custId,resRef) = true
       then format(stdout,"Reservation succeded!
                           Reference = ˜S",Seq{resRef.id});0
       else format(stdout,"Reservation failed! No room available!");1
       end
  end
 end
end
```

Having all three component types (`CustomerMgt`, `HotelMgt`, and `ReservationMgt`) and their associated realization contracts, we have created a `ComponentInstance` object corresponding to each of them, and wired these into an `architecture`, by means of two `assemblyBindings`. The `componentInstance`s are `customerMgr`, `hotelMgr`, and `reservationMgr` respectively. As mentioned in the previous section, in order to simulate the services provided by `reservationMgr`, it is necessary to create a corresponding `Simulator` object. Afterwards, simulation of any of the provided services is performed by calling the `Simulator` object's `simulate()` method, sending the service itself and its actual parameters list as arguments.

XOCL tests have been written for simulating all services provided by `reservationMgr`. These tests have been executed through the XMF-Mosaic console. For the given specifications and input data, the outputed results have indicated correct interoperability of the components in question.

Chapter 6
Avoiding Hastiness in Modeling

This chapter presents an approach to teaching software modeling that has been put into practice by its authors at the Babeş-Bolyai University of Cluj-Napoca. This approach aims at persuading students of the advantages that may derive from the usage of rigorous models. The development of such models, which relies on the technique known as Design by Contract, is a must in the context of the Model-Driven Engineering (MDE) paradigm. Another goal of our approach is for students to acquire core software modeling principles and techniques. Nowadays, the web provides students and professionals with a large amount of modeling examples. Many of these enclose various pitfalls, therefore modelers should have competence to evaluate them correctly in order to acquire efficient modeling skills. Obviously, the general principles and techniques used in assertion specification are far more important then the particular assertion language employed - OCL (Object Constraint Language), in our case. Following a decade of use and teaching of OCL, the conclusion that we have reached is that starting with a defense of the necessity and advantages offered by the use of constraints is a far more efficient teaching method compared to a pure technical introduction into the language itself. In this respect, the approach that we propose falls into the category of “inverted curriculum”.

6.1 Motivation

In MDE, “models are not only the primary artifacts of development, they are also the primary means by which developers and other systems understand, interact with, configure and modify the runtime behavior of software” [44].

The final goal of MDE technologies is to generate (almost) complete applications in various programming languages, such as Java, C#, C++, and so on. This requires starting from complete and unambiguous models, specified by means of rigorous modeling languages. The current technique used to specify languages is metamodeling. Irrespective of the abstraction level involved (meta-meta, meta or model level), in order to achieve rigorous model descriptions, the use of constraints (assertions) is a must.

The use of assertions in software development is promoted by the Design by Contract technique. In [59], the author identifies four applications of their use, namely: help in writing correct software, documentation aid, support for testing, debugging and quality assurance and support for software fault tolerance.

Working with assertions is therefore a core technique that model designers must manage. Despite this, practice shows that Design by Contract is not yet employed at its full potential. This may be due to both the lack of relevant examples showing the advantages of using OCL, and the availability of a large number of examples which, at best, cause confusion among readers. An experience of over ten years in teaching OCL to computer science students (at both bachelor and master levels) has allowed us to conclude that, apart from providing positive recommendations (books, papers, tools), warning potential OCL users on the pitfalls enclosed by negative examples is mandatory. As web users, students are exposed to both clear, well-written documents and to documents containing various drawbacks, on whose potential occurrence teachers have the duty of raising warnings. However, merely showing that particular models or specifications are inadequate or even incorrect with respect to the purpose they were created for is not enough. Presenting at least one correct solution and arguing on its advantages is a must.

Complementing models with OCL is meant at eliminating specifications ambiguities, increasing rigor, reaching a full and clear definition of query operations, as well as promoting Design by Contract through the specification of pre and post-conditions.

Development of models and applications takes place as an iterative incremental process, allowing return to earlier stages whenever the case. Enhancing models with OCL specifications facilitates their deeper understanding, through both rigor and extra detail. Whenever the results of evaluating OCL specifications suggest a model change, this change should only be done if the new version is more convenient compared to the previous ones, as a whole. The use of OCL specifications should contribute to the requirements validation. An application is considered as finished only when there is full compliance among its requirements, its model, and the application itself.

The remaining of this chapter is organized as follows. Section 6.2 explains the reasons why teaching OCL through examples integrated in models is more advantageous compared to the classical way of teaching OCL. In Section 6.3, we argue on the necessity of understanding the model's semantics, which is the first prerequisite for achieving a good specification. Section 6.4 emphasizes the fact that we need to consider several modeling solutions to a problem and choose the most convenient one with respect to the aspects under consideration. Section 6.5 shows the role of OCL in specifying the various model uses, while Section 6.6 justifies through an example the need of using snapshots for validating specifications. The chapter ends with conclusions in Section 6.7.

6.2 Teaching OCL through Examples Integrated in Models

There are various ways of teaching OCL. The classical approach emphasizes the main language features: its declarative nature and first order logic roots, the type system, the management of undefined values, the collection types together with their operations and syntax specificities, and so on [29], [12]. Many of the examples used to introduce collections employ expressions with literals, which are context-independent and easy to understand.

OCL is a textual language which complements MOF (Meta Object Facility)-based modeling languages. The students' interest in understanding and using the language increases if there are persuaded of the advantages earned from enriching models with OCL specifications. To convince students of the usefulness of OCL, the chosen examples should be suggestive in terms of models and enlightening in terms of earned benefits. That is the reason why we have considered more appropriate taking an "inverted curriculum"-type of approach, by introducing OCL through examples in which the OCL specifications are naturally integrated in models. Unfortunately, along with positive OCL specification examples, the existing literature offers also plenty of negative ones, starting with the WFRs (well-formedness rules) that define the static semantics of modeling languages. The negative examples may wrongly influence students' perception. Therefore, we argue that a major issue in teaching OCL to students is explaining them the basic principles that should be obeyed when designing OCL specifications, principles that should help them avoid potential pitfalls.

Two modeling examples that have been probably meant to argue for the use and usefulness of OCL (taking into account the title of the paper in question) are those presented in [79]. The examples and solutions proposed by this article provide an excellent framework for highlighting important aspects that should be taken into account within the modeling process. In the second semester of the 2010-2011 academic year, we have used these examples in order to warn students on the pitfalls that should be avoided when enriching models with OCL specifications.

6.3 Understanding the Model's Semantics

A model is an abstract description of a problem from a particular viewpoint, given by its intended usage. The design model represents one of the possible solutions to the requirements of the problem to solve. It is therefore essential for the students to realize the necessity of choosing a suitable solution with respect to the aspects under consideration. The first prerequisite for designing such a model is a full understanding of the problem at hand, reflected in a thorough informal requirements specification. Nygaard's statement "Programming is Understanding" [80] is to be read as "Modeling is Understanding", since "Object-oriented development promotes the view that programming is modeling" [62]. Understanding is generally acquired through an iterative and incremental process, in which OCL specifications play a major role. This is due to the fact that "if you don't understand something, you can't code it, and you gain understanding trying to code it." [80]. To be rigorous, "understanding" is only the first mandatory step to be accomplished in both programming and modeling. Finding the problem solution and describing it intelligibly must follow and take advantage of problem understanding. The informal specification of constraints is part of model understanding. Whenever constraints are missing from the initial problem requirements, they should be added in the process of validation and refinement of the informal problem specification.

To illustrate these statements, we will use one of the modeling examples from [79] (shown in Figure 6.1), which describes parents-children relationships in a community of persons. The model requirements description is incomplete with respect to both its intended functionalities and its contained information. In such cases, the model specification, both the graphical and the complementary textual one (through Additional Operations - AOs, invariants, pre and post-conditions), should contribute to enriching the requirements description. The process is iterative and incremental, marked by repeated discussions among clients and developers, until the convergence of views from both parties.

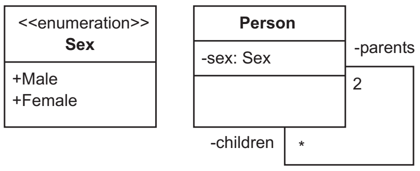


Fig. 6.1: Genealogical tree model [79]

The proposed solution should allow a correct management of information related to persons, even when this information is incomplete. Unknown ancestors of a particular person is such a case (sometimes not even the natural parents are known). For such cases, the model provided in [79] and reproduced in Figure 1 is inadequate, due to the infinite recursion induced by the self-association requiring each person to have valid references towards both parents. Snapshots containing persons with at least one parent reference missing will be thus qualified as invalid.

Both this problem and its solution, consisting in relaxing the `parents` multiplicity to 0..2, are now "classical" [27]. Partial or total lack of references (1 or 0 multiplicity) indicates that either one or both parents are unknown at the time.

The only constraint imposed in [79] on the above-mentioned model requires the parents of a person to be of different sexes. Following, there is its OCL specification, as given in [79].

```
self.parents->asSequence()->at(1).sex<>self.parents->asSequence()->at(2).sex
```

The technical quality of formal constraints (stated in either OCL or a different constraint language) follows from the fulfillment of several quality factors. Among them, there is the conformance of their behavior to their informal counterparts, the intelligibility of their formal specification, the debugging support offered in case of constraint violation, the similarity of results following both their evaluation in different constraint-supporting tools and the evaluation of their programming language equivalents (code snippets resulted from translating constraints into a programming language, using the constraint-supporting tools in question)[1]. Based on these considerations, the above specification, although apparently correct, encloses a few pitfalls:

1. In case at least one parent reference is missing and the multiplicity is 2, the evaluation of WFRs should signal the lack of conformance among the multiplicity of links between instances and the multiplicity of their corresponding association. To be meaningful, the

[1] Apart from the technical issues involved in evaluating the quality of assertions, there is also an efficiency issue, concerned with aspects such as the efficiency of the assertion specifications themselves, the amount of undesirable system states that can be monitored by using assertions, as well as their level of detail.

evaluation of model-level constraints should be performed only in case the model satisfies all WFRs. Unfortunately, such model compilability checks are not current practice. In case the `parents` multiplicity is 0..2 and one of the parents is left unspecified, the model will comply with the WFRs, but the constraint evaluation will end up in an exception when trying to access the missing parent (due to the `at(2)` call).

2. In case there are valid references to both parents, but the sex of one of them is not specified, the value of the corresponding subexpression is `undefined` and the whole expression reduces to either `Sex::Male <> undefined` or `Sex::Female <> undefined`. This later expressions provide tool-dependent evaluation results (`true` in case of USE [13] or Dresden OCL [2] and `undefined` in case of OCLE [52]). The results produced by OCLE comply with the latest OCL 2.3 specification [73]. However, as the topic of evaluating undefined values has not yet reached a common agreement, students should be warned on this.
3. Moreover, in case one of the parents reference is missing, the code generated for the above constraint will provide evaluation results which depend on the position of the `undefined` value with respect to the comparison operator. According to the tests we have performed[2], the Java code generated by OCLE and Dresden OCL throws a `NullPointerException` when the `undefined` value is located at the left of the `<>` operator, while evaluating to `true` in case the `undefined` value is at the right of the operator and the reference at the left is a valid one. As we are in the context of the MDE paradigm, which relies extensively on automatic code generation, the results provided by the execution of the code corresponding to constraints is an aspect that has to be taken into account when judging the quality of the formal constraints in question.
4. The OCL expression would have been simpler (not needing the `asSequence()` call), in case an ordering relation on `parents` had been imposed at the model level.

Similar to most of the activities involved in software production, the specification of assertions is an iterative and incremental process. That is why, in the following, we will illustrate such a process for the considered family case study. For the purpose of this section, we will work with the model from Figure 6.1, assuming though that the multiplicity of the `parents` reference has been set to `0..2`, so as to avoid infinite recursion.

Ordering the `parents` collection with respect to sex (such that the first element points to the mother and the second to the father) allows writing an invariant that is more detailed compared to the one proposed in [79] for the constraint regarding the parents' sex. Following, there is the OCL specification we propose in this respect, when both parents are known. In case of invariant violation, the debugging information is precise, allowing to easily eliminate the error's cause.

```
context Person
 inv parentsSexP1:
  self.parents->size() = 2 implies
   Sex::Female = self.parents->first().sex and
   Sex::Male = self.parents->last().sex
```

When any of the parents' sex is `undefined`, the invariant above evaluates to `false` in Dresden OCL and to `undefined` in OCLE. In similar circumstances, both Java code snippets generated for this invariant by the two tools return `false` when executed. Therefore,

[2] the corresponding Java/AspectJ projects can be downloaded from [5]

this invariant shape overcomes the drawbacks of the one from [79] previously pointed at items 1, 3 and 4. The triggering of a `NullPointerException` by the generated code in case of absence of one of the parents' sex has been avoided by placing the defined values (the `Sex::Female` and `Sex::Male` literals) on the left-hand side of equalities.

The solution to the problem mentioned at item 2 above comes from obeying to the separation of concerns principle. In order to avoid comparisons involving `undefined` values, whose results may vary with the OCL-supporting tool used, the equality tests of the parents' sex with the corresponding enumeration literals should be conditioned by both of them being specified. Such a solution is illustrated by means of the invariant proposal below.

```
context Person
 inv parentsSexP2:
  self.parents->size() = 2 implies
  ( let mother = self.parents->first() in
    let father = self.parents->last() in
     if (not mother.sex.oclIsUndefined() and not father.sex.oclIsUndefined())
     then mother.sex = Sex::Female and father.sex = Sex::Male
     else false
     endif
  )
```

The invariant above evaluates to `false` when any of the parents' sex is `undefined`, as well as when they are both defined but set inappropriately (the first parent's sex is not `Sex::Female` or the second is not `Sex::Male`, as previously established by the ordering rule). The evaluation results are the same for both the OCL constraint and its Java equivalent, irrespective of the tool used, OCLE or Dresden OCL. Therefore, this last invariant shape provides solutions to all pitfalls previously detected for its analogue from [79].

Yet, a correct understanding of the model in question leads to the conclusion that the mere constraint regarding the parents' sex is insufficient, despite its explicit specification for each parent. As rightly noticed in [27], a person cannot be its own child. A corresponding OCL constraint should be therefore explicitly specified.

```
context Person
  inv notSelfParent:
    self.parents->select(p | p = self)->isEmpty()
```

However, restricting the age difference among parents and children to be at least the minimum age starting from which human reproduction is possible (we have considered the age of sixteen) leads to a stronger and finer constraint than the previous, that may be stated as follows.

```
context Person
  inv parentsAge:
    self.parents->reject(p | p.age - self.age >= 16)->isEmpty()
```

In the above expression, each `Person` is assumed to own an `age` attribute. In case both the contextual instance and its parents have valid values for the `age` slot, the `reject(...)` subexpression evaluates to the collection of parents breaking the constraint in question.

The fulfillment of this constraint could be also required at any point in the construction of the genealogical tree. Assuming any parent to be created prior to any of its children, this restriction could be stated by means of the precondition included in the contract below.

```
context Person::addChildren(p:Person)
```

```
  pre childrenAge:
    self.children->excludes(p) and self.age - p.age >= 16
  post chidrenAge:
    self.children->includes(p)
```

The conclusion that emerges so far is that the lack of OCL specifications prohibiting undesired model instances (such as parents having the same sex, self-parentship or the lack of a minimum age difference among parents and children) seriously compromises model's integrity. The first prerequisite for models to reach their purpose is to have a complete and correct specification of requirements, and to deeply understand them. An incomplete specification reveals its limits when trying to answer questions on various situations that may arise. Specifying and evaluating OCL constraints should enable us to identify and eliminate bugs, by correcting the requirements and the OCL specifications themselves. Moreover, in the context of MDE, care should be taken to the shape of constraint specifications, ensuring that their evaluation using OCL-supporting tools provides identical results to those obtained by executing their equivalent code generated by those tools. Another conclusion, as important, is that the model proposed in the analyzed paper does not fully meet the needs of the addressed problem[3] and we are therefore invited to seek for a better solution.

6.4 Modeling Alternatives

A model equivalent to that of Figure 1, but which is more adequate to the specification of the required constraints, is the one illustrated in Figure 2. The model in question contains two recursive associations: one named `MotherChildren`, with roles `mother[0..1]` and `mChildren[0..*]` and the other `FatherChildren`, with roles `father[0..1]` and `fChildren[0..*]`.

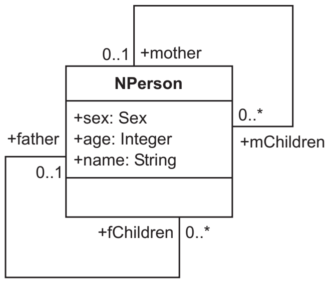


Fig. 6.2: An alternative model for expressing parents-children relationships

Within this model, the constraint regarding the parent' sex can be stated as proposed below.

```
context NPerson
```

[3] in case there is a single parent specified, we have no means to check whether the sex has been set appropriately, according to its role (mother or father) and we may need extra attributes (e.g. `age`) for specifying finer constraints

```
 inv parentsSexP1:
  (self.mother->size() = 1 implies Sex::Female = self.mother.sex) and
  (self.father->size() = 1 implies Sex::Male = self.father.sex)
```

Compared to its equivalent constraint stated for the model in Figure 1, the above one is wider, since it also covers the case with a single parent and checks the sex constraint corresponding to the parent in question. As previously pointed out, the problem with the initial model (the one in Figure 1) is that we cannot count on an ordering when there is a single parent reference available. The parent in question would always be on the first position, irrespective of its sex. As opposed to this, in Figure 2, the parents' roles are explicitely specified, with no extra memory required. When at least one parent's sex is `undefined`, the evaluation of this invariant returns `undefined` in OCLE and `false` in Dresden OCL, while the execution of the corresponding Java code outputs `false` in both cases. Given the invariant shape, the identification of the person breaking it is quite straightforward, therefore the problem can be rapidly fixed.

An alternative invariant shape, providing the same evaluation result in both OCLE and Dresden OCL, for both OCL and Java code is the one below.

```
context NPerson
 inv parentsSexP2:
  (self.mother->size() = 1 implies
   (let ms:Sex = self.mother.sex in
     if not ms.oclIsUndefined() then ms = Sex::Female
     else false endif )
  ) and
  (self.father->size() = 1 implies
   (let fs:Sex = self.father.sex in
     if not fs.oclIsUndefined() then fs = Sex::Male
     else false endif)
  )
```

With respect to the second constraint, we propose the following specification in context of the model from Figure 2.

```
context NPerson
 inv parentsAge:
  self.mChildren->reject(p | self.age - p.age >= 16)->isEmpty() and
  self.fChildren->reject(p | self.age - p.age >= 16)->isEmpty()
```

The `parentsAge` invariant above uses one of the specification patterns that we have proposed in [31] for the *For All* constraint pattern. If we were to follow the classical specification patterns available in the literature, the invariant would have looked as follows.

```
context NPerson
 inv parentsAgeL:
  self.mChildren->forAll(p | self.age - p.age >= 16) and
  self.fChildren->forAll(p | self.age - p.age >= 16)
```

A simple analysis of these two proposals reveals that the `parentsAgeL` invariant shape is closer to first order logic. However, in case of constraint violation, this does not provide any useful information concerning those persons breaking the invariant, as the first one does. The specification style used for the `parentsAge` invariant offers model-debugging support [31], a major concern when writing assertions.

The corresponding pre and post-conditions are similar to their equivalents from the previous section, therefore their specification could be left to students, as homework.

6.5 Explaining the Intended Model Uses

Any requirements specification should include a detailed description of the intended model uses. In case of the model under consideration, it is important to know what kind of information may be required from it. Is it merely the list of parents and that of all ancestors? Do we want the list of ancestors ordered, with each element containing parents-related information, in case such information is available? Do we only need information regarding the male descendents of a person?

In case of the initial model in which the recursive association is ordered, the list of all ancestors of a person can be easily computed as follows.

```
context Person
  def allAncestors():Sequence(Person) =
    self.parents->union(self.parents.allAncestors())
```

The evaluation result for the constraint above is correct only if we assume the genealogical tree as loop-free. This latter constraint is implied by the one restricting the minimum age difference between parents and children. In the absence of this assumption, the OCL expression's complexity increases.

A simpler alternative for this case employs the semantic closure operation on collections. This operation, now included in OCL 2.3, has been implemented in OCLE ever since its first release and returns a set.

```
context Person
  def allAncestors():Sequence(Person) =
   (Sequence{self}->closure(p | p.parents))->asSequence()
```

The advantages offered by the modeling solution proposed in Figure 6.2 are clear in case we are interested to compute all ancestors of a person, specifying explicitly which of them are unknown (not stored in the database). Following, there is a possible OCL query to be used in this purpose, that employs the tuple type.

```
context NPerson
 def tParents: TupleType(ch:NPerson, mo:NPerson, fa:NPerson) =
 Tuple{ch = self, mo = self.mother, fa = self.father}

 def allTParents: Sequence(TupleType(ch:NPerson, mo:NPerson, fa:NPerson)) =
 Sequence{self.tParents}->closure(i | Sequence{i.mo.tParents, i.fa.tParents})
                                    ->asSequence()->prepend(self.tParents)
```

Evaluating in OCLE the family tree presented in Figure 6.3, we obtain:

```
Sequence{Tuple{np1,np2,np3,}, Tuple{np2,np4,np5}, Tuple{np3,np12,Undefined},
  Tuple{np4,np8,np9}, Tuple{np5,np6,np7}, Tuple{np12,Undefined,np13},
  Undefined, Tuple{np8,Undefined,Undefined}, Tuple{np9,np11,Undefined},
  Tuple{np6,Undefined,np10}, Tuple{np7,Undefined,Undefined},
  Tuple{np13,Undefined,Undefined}, Tuple{np11,Undefined,Undefined},
  Tuple{np10,Undefined,Undefined}
}
```

Since the members of each tuple are (child, mother, father), in this particular order, the analysis of the above evaluation result allows an easy representation of the corresponding genealogical tree.

As regarding a potential query meant to compute all descendants of a person, the only difference between the two proposed models concerns the computation of a person's children.

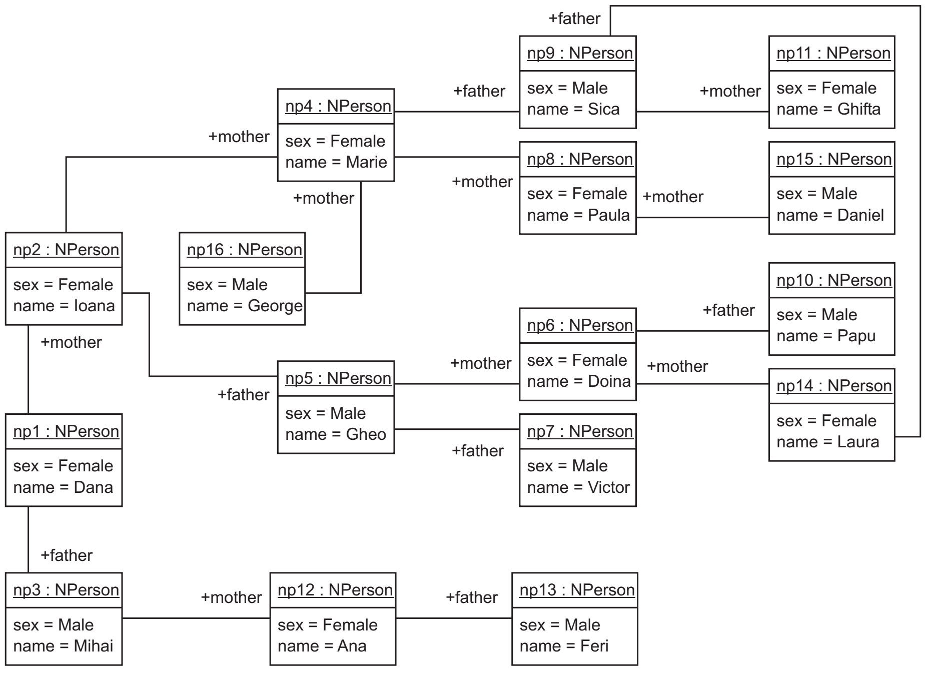


Fig. 6.3: Sample snapshot of the model from Figure 6.2

In this respect, the model in Figure 6.1 already contains a `children` reference, while in case of the one from Figure 6.2, a corresponding query needs to be defined, as shown below.

```
context NPerson
 def children: Set(NPerson) =
  if self.sex = Sex::Female
  then self.m_children
  else self.f_children
  endif
```

6.6 Using Snapshots to Better Understand and Improve the Requirements and the Model

One of the primary roles of constraints is to avoid different interpretations of the same model. Therefore, the specification process must be seen as an invitation for a complete and rigorous description of the problem, including the constraints that are part of the model. The model must conform to the informally described requirements, even before attaching constraints. In case this condition is not fulfilled, the constraints specification process must ask for additional information, meant to support an improved description of requirements, a deeper understanding of the problem, and by consequence, a clear model specification.

Despite its importance, as far as we know, this issue has not been approached in the literature. That is why, in the following, we will try to analyze the second example presented in [79], concerning a library model. This example aims to model the contractual relationships between a library, its users and companies associated with the library. The only informal specification provided is the following: "In this example, we'll assume that the library offers a subscription to each person employed in an associated company. In this case, the employee does not have a contract with the library but with the society he works for, instead. So we add the following constraint (also shown in Figure 10): . . . ".

First of all, we would like to remind the definition of a contract, as taken from [6]: "A binding agreement between two or more parties for performing, or refraining from performing, some specified act(s) in exchange for lawful consideration." According to this definition and to the informal description of requirements, we conclude that, in our case, the parts in the contract are: the user on the one hand, and the library or the company, on the other hand. As one of the involved parts is always the user, the other part is either the library (in case the user is not employed in any of the library's associated companies), or the company (in case the user is an employee of the company in question).

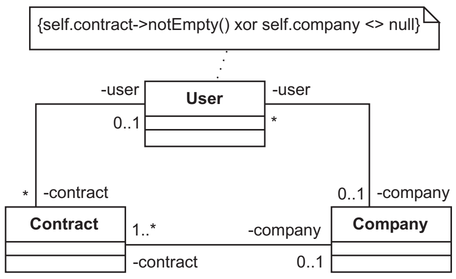


Fig. 6.4: The library model from [79], Figure 10

Regarding the conformance among requirements, on the one side, and model, on the other side (the class diagram, the invariant presented in Figure 10 and the snapshots given in Figures 12 and 13 of [79]), several questions arise. Since a thorough analysis is not allowed by the space constraints, in the following, we will only approach the major aspects related to the probable usage of the model. In our opinion, this concerns the information system of a library, that stores information about library users, associated companies, books, book copies and loans. The library may have many users and different associated companies.

Since the `Library` concept is missing from the model proposed in [79], we have no guaranty that, in case the user is unemployed, the second participant to the contract is the library. Moreover, in case the user is employed, the invariant proposed in [79] does not ensure that both the user and the corresponding company are the participants to the contract. As a solution to this, we propose an improved model for the Library case study (see Figure 6.5), as well as two corresponding invariants, one in the context of `Contract` and the other in the context of `User`.

```
context Contract
 inv onlyOneSecondParticipant:
  self.library->isEmpty() xor self.company->isEmpty()
```

```
context User
 inv theContractIsWithTheEmployer:
  if self.employer->isEmpty()
   then self.contract.library->notEmpty()
   else self.employer = self.contract.company
  endif
```

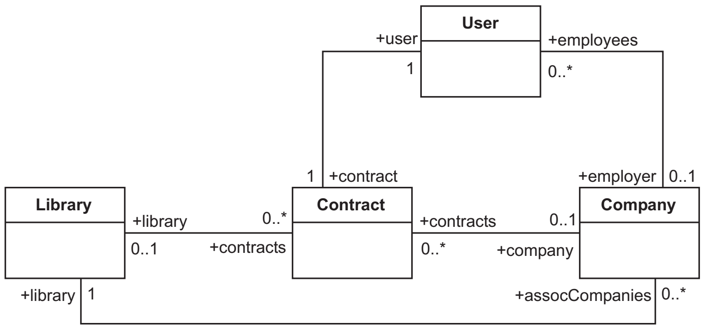


Fig. 6.5: A revised version of an excerpt of the library model from [79]

The above constraints forbid situations like those from Figure 6.6 (in which the user `u1` has a contract `c1` both with the library `l1` and the company `comp1`) and Figure 6.7 (in which the user is employed by `comp3`, but its contract `c2` is with `comp2`). These undesirable model instantiations are not ruled out by the invariant proposed in [79] in the `User` context.

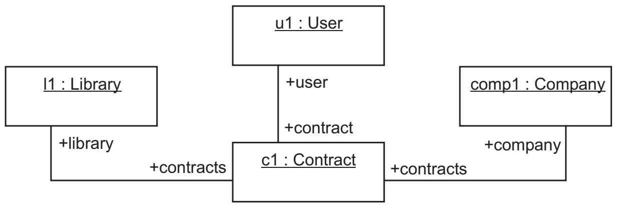


Fig. 6.6: The user has a contract with both the library and the company

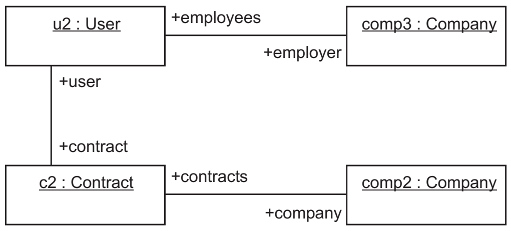


Fig. 6.7: The user is employed by `comp3`, but its contract `c2` is with `comp2`

Even more, in Figure 12 from [79], `contractB65` and `contractR43` have only one participant, `company80Y`, a stange situation in our oppinion. Also, in the same figure, if `userT6D` is unemployed by `company80Y`, and, by consequence, `contractQVR` is between `userT6D` and the library, we cannot understand why `company80Y` (which does not include among its employees `userT6D`) has a reference towards `contractQVR` between `userT6D` and the library.

Unfortunately, as stated before, our questions do not stop here. In Figure 10 from [79], a user may have many contracts, but in the requirements a different situation is mentioned. In the class diagram of Figure 10, all role names are implicit, which burdens the inteligibility of the model.

In this example, the snapshots meant to be used for testing have supported us in understanding that the requirements are incomplete and, by consequence, so are the model and the proposed invariant. In such cases, improving the requirements is mandatory.

6.7 Conclusions

The building of rigorous models, which are consistent with the problem requirements and have predictable behavior, relies on the use of constraints. Such constraints are not independent, they refer to the model in question. Consequently, the model's accuracy (in terms of the concepts used, their inter-relationships, as well as conformance to the problem requirements) is a mandatory precondition for the specification of correct and effective constraints. In turn, a full understanding of the model's semantics and usage requires a complete and unambiguous requirements specification. Requirements' validation is therefore mandatory for the specification of useful constraints.

The examples presented in this chapter illustrate a number of bugs caused by failure to fulfill the above-mentioned requirements. Unfortunately, the literature contains many erroneous OCL specifications, including those concerning the UML static semantics, in all its available releases. Having free access to public resources offered via the web, students should know how to identify and correct errors such as those presented in this chapter. Our conclusion is that the common denominator for all the analyzed errors is *hastiness*: hastiness in specifying requirements, hastiness in designing the model (OCL specifications included), hastiness in building and interpreting snapshots (test data).

There are, undoubtedly, several ways of teaching OCL. The most popular (which we have referred as the "classic" one, due to its early use in teaching programming languages), focuses on introducing the language features. OCL being a complementary language, we deemed important to emphasize from the start the gain that can be achieved in terms of model accuracy by an inverted curriculum approach. In this context, we have insisted on the need of a complete and accurate requirements specification, on various possible design approaches for the same problem, on the necessity of testing all specifications by means of snapshots, as well as on the need to consider the effects of a particular OCL constraint shape on the execution of the code generated for the constraint in question.

However, the teaching and using of OCL has a number of other very important issues that have been either not addressed or merely mentioned in this chapter, such as the specifications' intelligibility, their support for model testing and debugging, test data generation, language features, etc. The theme approached by this chapter only concerns, in our view, a first introduction to the language and its purpose.

Chapter 7
Producing Java Applications Using Models: Conclusions Drawn from Case Studies

7.1 Motivation

The main objective of Model Driven Development paradigms is the large scale use of models in software development. Usually, models used to produce software may contain different views, describing different software layers: user interface, business layer, database. Moreover, specifying the behavior of business layer components may require different formalisms and techniques, such as state transition diagrams (in case of reactive objects), activity diagrams (in case of transformational objects), sequence diagrams, algebraic specifications or OCL to specify observers, pre and post-conditions associated to all kinds of methods.

As known, models can be specified by using different modeling languages. Similar to programming languages, a criterion used to classify modeling languages groups them in General Modeling Languages (shortly GMLs) and Domain Specific Modeling Languages (shortly DSMLs). The support for a more efficient code generation was among the main arguments used in favor of DSMLs against GMLs. The price to pay for this advantage is the considerable effort required to design and validate/test a DSML. The latest approaches in this domain argue for mixing programming with modeling [7].

Therefore, we may conclude that an exhaustive approach of producing software by transforming different views in code (model-to-code, M2C) and interleaving the code corresponding to views is a very complex problem.

7.2 Our Approach: Using UML Models Complemented with OCL Specifications

In the framework of the EMF_SIVLA project [4], we have been interested in using UML models and OCL specifications describing the business layer. More precisely, we have been interested in using such models to produce software applications in Java. We have used the OCLE built-in code generator to transform model classes, interfaces, datatypes, enumeration and tuple types into their corresponding Java source code equivalents. In its current version, OCLE also generates the code required for the runtime management of associations. Additionally, attached OCL specifications are turned into runnable code, which is integrated in the code corresponding the UML model, producing a Java application. Thus, one may also check invariants specified in the model and observers at runtime. As concerning pre and post-conditions, runtime verification is even more important, since the current version of OCLE only enables checking their compilability and not evaluating them. Moreover, as mentioned in section 6.3, it is important to test the conformance of results obtained when evaluating assertions in OCL tools with those obtained when evaluating their equivalent specifications in Java. The code generation process has been conceived to simplify user interaction as much as possible. Our main goal has been to convince users that:

- by using appropriate techniques and tools, transforming models to code is a natural and very convenient means of producing software;
- Design by Contract is a key design technique that supports the production of reliable software. In addition, this technique enables users to focus on real modeling problems and to describe complete and unequivocal models.

As opposed to the code generated by the EMF tool, the code generated by OCLE is simpler as concerning the model architecture. In our tool, the transformation of the UML static model in Java code is accomplished as straightforward as possible, without introducing observer patterns or generating both an interface and a class corresponding to each model-level class. However, our code is richer due to the code generated for OCL specifications.

7.3 Code Generation for UML Models

The source code generator allows you to turn design models into skeleton code. Skeleton code consists of the structure code for all model entities that have an equivalent in the target programming language. The goal of the generator is to translate most of the model elements in their corresponding source code equivalents. OCLE code generator considers only the elements composing the static view of the model. An outstanding capability is the management of accessor methods for associations, including association classes and qualified associations.

The code generation process has two phases. The first consists in the validation of the active UML model against a set of profile rules. Currently, two profiles are considered: the UML profile and the Java profile. The UML profile specifies the rules that the model must fulfill in order to be correct against the UML static semantics (well-formedness rules). The Java profile imposes additional, Java specific rules (e.g. no feature may have a Java reserved word as its name). The model checking process is a batch evaluation process, since profile rules are expressed in OCL. Therefore, any model errors are reported in the same manner as with plain batch evaluation. If at least one error is identified, the user will be prompted to confirm that he wants to go further with code generation, because the resulting Java code may have syntactic / semantic errors. However, at this step, he may choose to stop the code generation process.

The errors related to the UML WFRs can be classified in two categories:

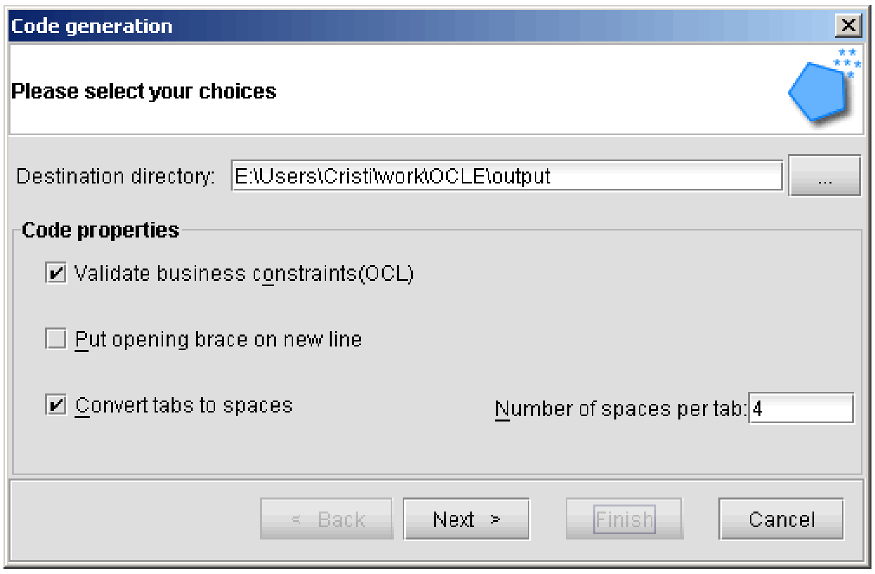
- errors that impact the Java code generated, such as server types (classes) invisible in the client namespace, name conflicts, forbidden relationships and so on;
- errors that cannot impact the Java code generated, such as those referring to the fact that different instances (objects) or relationships between instances (links) do not comply with the declarations of their generators (classes, associations).

Some of the errors included in the second category were introduced in a conscious manner when constructing snapshots breaking different assertions, needed for testing assertion specifications.

The code generated for the UML model is actually a solved problem. As stated above, the code generated for managing associations is very reliable. At the moment of designing and implementing the OCLE code generator, in 2003, the Java language did not implement the support for generic types. Therefore, the implementation of collection classes does not use generic types. An excellent paper on this topic is [17]. However, even if the paper has been published four years after OCLE, our implementation supports all the functionalities mentioned by its authors. Refactoring collections by using Java generic collections and the code generated for managing associations is one of our objectives of future work. In the following, we will focus on the code generated for OCL expressions.

7.4 Code Generation for OCL Expressions

If the *Validate business constraints* checkbox (included in the code generation wizard in Fig-


Code generation

Please select your choices

Destination directory: E:\Users\Cristi\work\OCLE\output

Code properties

[✓] Validate business constraints(OCL)

[ ] Put opening brace on new line

[✓] Convert tabs to spaces

Number of spaces per tab: 4

Fig. 7.1: OCLE code generator wizard

ure 7.1) is selected, apart of the code corresponding to the UML model, OCLE will also generate source code equivalents for the OCL expressions attached to the active user model. The resulting code fragments are properly integrated in the source code for the structure of the model. Only OCL specifications corresponding to the user model level are taken into account; no code is generated for OCL specifications corresponding to the metamodel level (WFRs and Additional Operations).

In its current version, OCLE translates the following entities:

- class invariants
- method preconditions and postconditions
- observer operations, specified using the `def let` mechanism

In the following, we will describe the way in which each of the above-mentioned entities is integrated into the code generated for the UML model architecture.

7.5 Code Integration

7.5.1 Code Integration for Class Invariants

If a class in the UML model has at least one invariant attached to it, the corresponding source code is modified as described in the following. A public inner class named `ConstraintChecker` is declared. This class is responsible for the validation of class invariants at runtime. Actually, the developer must decide upon where to trigger the verification of each class invariant. The constraint checker class inherits either from a default class `BasicConstraintChecker` provided by the framework `OCLFramework` included in the OCLE package (in case the constrained class has no parents or its parents are not constrained) or from the constraint checker class attached to the parent class (in case the parent class is also constrained). Invariant inheritance is ensured in this way. The constraint checker class has a public method, `checkConstraints`, which is responsible for triggering the verification of all class invariants, including any inherited invariants. For any specific class invariant, a public method that verifies it is inserted in the constraint checker class. The `checkConstraints` method contains one call for each such method. Thus, wherever the Java compiler allows it, one may trigger the verification of one or more class invariants using the following code template:

```
new ConstraintChecker().<checkerMethod>();
```

where `<checkerMethod>` is either `checkConstraints` for all constraints or a specific method for checking a particular invariant. Since the `ConstraintChecker` class is public, one may trigger the evaluation of class invariants even outside the body of the constrained class.

7.5.1.1 Managing Class Invariants' Violation at Runtime

In the above mentioned approach, invariant violation is simply managed by printing a message. This works fine to compare the results obtained in OCLE with those obtained at runtime. However, if we intend to perform assertion monitoring in the application, then we have to use the exception handling mechanisms provided in Java. A solution we have tested was to use ”custom exceptions”. In this case, the class `InvariantException` will be included in the default application package.

```
@SuppressWarnings("serial")
public class InvariantException extends Exception {
 public InvariantException(String message) {
  super(message);
 }
}
```

In order to understand the changes we propose for the `ConstraintChecker` class, we have included below an excerpt from the code generated and run for the `CompanyExample` application included in the `Examples` folder of the OCLE 2.0 distribution package. The modifications we propose are highlighted. Due to space constraints, we have only kept the code corresponding to the invariant `managerEmployed`. For all the other invariants, the changes are similar.

```
public class ConstraintChecker extends BasicConstraintChecker {

 public void checkConstraintsThrowingException()
  throws InvariantException{
   super.checkConstraints();
   check_Company_managerEmployed();
   ...
  }

  public void check_Company_managerEmployed()
   throws InvariantException{
    Person personManager = Company.this.getManager();
    boolean bIsDefined = Ocl.isDefined(personManager);
    Person personManager0 = Company.this.getManager();
    boolean bIsUnemployed = personManager0.isUnemployed;
    boolean bNot = !bIsUnemployed;
    boolean bImplies = !bIsDefined || bNot;
    if (!bImplies) {
     throw new InvariantException("invariant␣’managerEmplyoed’␣failed" +
      "for␣object" + Company.this);
    }
   }
}
```

As we may notice, there are very few changes with respect to the initial version. Moreover, in case of the first three changes, the code was only added, while, in the last one, `throw new InvariantException`” has replaced `System.err.println`.

7.5.2 Code Integration for Method Pre/Post-conditions

For operation constraints, OCLE uses the same code pattern as for class invariants. Some differences exist, however. The constraint checker class is inserted at the beginning of the body code for the constrained method. It has no base class. It contains two public methods, `checkPreconditions` and `checkPostconditions`. Both methods have the same arguments as the constrained method itself, so that the names of the formal parameters are available to the code that checks the constraints. The constraint checker class also contains a private checker method for each method constraint. The resulting private methods are called by either `checkPreconditions` or `checkPostconditions`. The code for a constraint method looks like this:

```
public boolean hireEmployee(Person p) {
 class ConstraintChecker {
  public void checkPreconditions(Person p) {
   check_precondition(p);
  }

  public void checkPostconditions(Person p) {
  }

  public void check_precondition(Person p) {
   Set setEmployee = Company.this.directGetEmployee();
   boolean bIncludes = CollectionUtilities.includes(setEmployee, p);
   boolean bNot = !bIncludes;
   if (!bNot) {
   System.err.println("precondition␣'precondition'␣failed␣for␣object␣"
    + Company.this);
   }
  }
  boolean result;
 }

 ConstraintChecker checker = new ConstraintChecker();
 checker.checkPreconditions(p);
 //checker.result = internal_hireEmployee(p);
 //checker.checkPostconditions(p);
 if (checker.result == true) {
  checker.result = internal_hireEmployee(p);
  checker.checkPostconditions(p);
 }
 return checker.result;
}
```

As we can see, the real body of the constrained method is moved in another method (in this case `internal_hireEmployee()`). Moreover, the constraint checker class also has a public field, `result`, which maps to the result variable available in OCL postconditions. This variable is properly assigned before triggering the verification of any postcondition. However, some postcondition specific constructs, such as `@pre` and the `oclIsNew` operation in `OclAny` are not properly mapped in the generated source code. If the code generator encounters such constructs, it issues adequate warning messages. The highlighted code above represents another example of fixing bugs in the OCLE code generator. In the current version, the highlighted code is missing and the commented code is included. The problem is due to the fact that in case the precondition fails, then, after printing the error message `System.err.println(...)`, the internal method is called. The highlighted code fixes this bug. Moreover, in order to manage exceptions, an approach similar to the one mentioned for invariants can be proposed.

7.5.3 Code Integration for Observers Specified in OCL

All OCL observers are mapped to Java methods in the generated source code. The corresponding method for such an observer (also referred as definition constraint) is public and is placed within the class in the context of which the definition constraint has been specified. The following example (Finance Business, also included in OCLE distribution) illustrates the translation of definition constraints.

```
context Portofolio
 def prices:
  let priceOfOrders: Real = orders.security.price->sum()
```

In this case, class `Portofolio` would contain the following code (completely generated by OCLE):

```
public float priceOfOrders() {

 Set setOrders = Portofolio.this.getOrders();

 //evaluate 'collect(security)':
 List bagCollect = CollectionUtilities.newBag();
 final Iterator iter = setOrders.iterator();
 while (iter.hasNext()) {
  final Order decl = (Order)iter.next();
  Security securitySecurity = decl.getSecurity();
  bagCollect.add(securitySecurity);
 }
 bagCollect = CollectionUtilities.flatten(bagCollect);

 //evaluate 'collect(price)':
 List bagCollect0 = CollectionUtilities.newBag();
 final Iterator iter0 = bagCollect.iterator();
 while (iter0.hasNext()) {
  final Security decl = (Security)iter0.next();
  float fPrice = decl.price;
  bagCollect0.add(Real.toReal(fPrice));
 }
 bagCollect0 = CollectionUtilities.flatten(bagCollect0);

 float fSum = CollectionUtilities.sum(bagCollect0);
 return fSum;
}
```

7.5.4 Code Integration for Tuple Types

If business constraints use OCL tuple types, OCLE will turn each identified (distinct) tuple type into a class. Tuple parts are converted to public fields with the names deduced from the names of the involved tuple parts. This mapping is in fact similar to C/C++ structs. The `equals()` method is also generated, to ensure proper comparison between tuples (tuple type instances). As an example, let us consider the following OCL tuple type
`TupleType(array:String, length:Integer)`.
OCLE will generate the following code for it:

```
//File TupleType1.java
package tupleTypes;

publc class TupleType1 {
 public boolean equals(Object arg) {
  if (!(arg instanceof TupleType1)) {
   return false;
  }
  boolean result = true;
  TupleType1 local = (TupleType1)arg;
  result&= (array != null ? array.equals(local.array)
                          : local.array == null);
  result&= (length == local.length);
  return result;
 }

 public String array;
 public int length;
}
```

The name of the resulting class is generated by OCLE, since OCL does not allow for the specification of tuple type names. All tuple types identified in the OCL expressions are placed in a dedicated package, named `tupleTypes` and this behaviour cannot be changed in this version of OCLE - this means that one cannot change the location of a class corresponding to a tuple type.

7.6 Code Patterns Used and Limitations

The translation from OCL to Java follows a syntax-directed approach. Support for metamodeling (`OclAny`, `allInstances` and so on) is provided by means of a simple library shipped with the distribution. The library, named `OCLFramework.jar` is located in the `/lib` subdirectory of your OCLE installation directory. Since the generated code contains calls to classes in the framework, you will have to include this archive in the classpath of the generated application. Basically, the entire OCL 2.0 grammar is supported. There are however two drawbacks. First of them has already been mentioned: the `@pre` construct is not properly mapped. Moreover, this version provides limited degree of customization as far as constraint violations are concerned. More precisely, the same pattern, based on

`System.err.println` is used whenever a constraint is violated. Additionally, although supported by the compiler, messaging expressions are not translated in equivalent source code.

7.6.1 OCL Datatypes

The source code generator was designed with efficiency in mind. Therefore, primitive OCL types are mapped to primitive types provided by the Java programming language, thus avoiding excessive wrapping. More precisely, `Boolean` maps to `boolean`, `Integer` maps to `int` and `Real` maps to `float`. The `String` type provided by OCL is translated using the `java.lang.String` type. These mappings are not customizable in the actual version of OCLE.

7.6.2 The `OclAny` Metaclass

The `OclAny` metaclass is equated with the root class `java.lang.Object`. The operations in `OclAny`, including the navigation to `OclType` are translated by redirecting them to the `Ocl` class in the above mentioned library. Each method in this class has an object as its first argument. This argument is a placeholder for `self`. The rest of the arguments are those mentioned in the OCL specification. Whenever an object is expected and the previous result is of primitive type, the expected object is obtained by wrapping the primitive value. Please note, however, that some operations are not properly implemented in this version (this is the case for `oclIsNew()` and `oclInState()`).

7.6.3 The `OclType` Metaclass and OCL Collection Types

The `OclType` metaclass in OCL is mapped to a class `OclType` specifically designed for this purpose in the `OCLFramework` library. Basically, `OclType` objects are wrappers around `java.lang.Class` objects. However, `OclType` can be used to model nested collection types, such as `Set(Bag(Integer))`. The following table lists the mappings used for OCL collection types.

OCL Collection type	**Java equivalent type**
Set	java.util.Set
OrderedSet	ro.ubbcluj.lci.codegen.framework.dt.OrderedSet
Sequence	java.util.List
Bag	java.util.List

Type casts and conversion to primitive values are generated whenever necessary, so that the resulting source code is semantically correct from the point of view of the Java compiler. Operations in OCL collection types are translated by redirecting them to the framework class `CollectionUtilities`, as in the case of `OclAny`. As in that case, the first of argument of the methods in the `CollectionUtilities` class is a placeholder for `self` - the collection on which the operation is called.

7.6.4 OCL `Undefined` Values

This version of the code generator does not handle OCL `Undefined` values properly. The biggest problem we encountered when modeling the `Undefined` value was its type, `OclVoid`. The OCL specification asserts that this type must conform to all types in the user model or in the OCL datatype system. Therefore, the code generator does not map the `OclVoid` type explicitly. Instead, the semantics mentioned in the OCL grammar is partially ensured using the following conventions:

- The `null` literal is used as a placeholder for `Undefined` in the case of non primitive types.
- `Integer.MAX_VALUE` is used as a placeholder for undefined integer values, while for undefined real numbers `Float.POSITIVE_INFINITY` is used . As a consequence, undefined boolean values cannot be modeled in the generated code. Due to these limitations, one may encounter some compilation problems in the generated source code.

7.7 Conclusions

Despite some above-mentioned inconveniencies, the OCLE generator has been studied and used successfully not just by us and by our students. The tool has also been analyzed at British Computing Society [61] and used to produce banking software. In this respect, we provide an excerpt from [8].

> Before John's session, I sat in on a session by Richard Mitchell and Rob Day entitled "Heuristics for improving precision in UML models". This was relatively familiar territory for me, as it was essentially about tightening up domain models - in their case study it was for defining business rules for web service XML schemas - using the Object Constraint Language. Interesting to hear about a team who's actually using OCL in a bank (that they refused to name, but the clues suggested I might have been there at some point in my murky past!) They also seem to be getting some mileage out of the OCLE tool I reviewed a while back. It's one of the better OCL editors, so I'm not altogether surprised that they chose it. (It's also free, so that's another incentive.)

On the web, one may find suggestive references about using OCLE in research, education and even in software industry.

OCLE supports users in generating more than 75 percent of the application's code. The code is fully conformant with the model, increasing users belief that producing applications by means of a direct-engineering process is feasible. Obviously, this only happens when the model is complete and all assertions and observers are fully specified in OCL.

The code generated by OCLE can be further improved both regarding the quantity and the quality. As regarding the quantitative aspect, in the future, we intend to generate:

- explicit constructors,
- get and set methods,
- the creation of object configurations corresponding to OCLE snapshots,
- the code for automated testing.

The quality may be improved by:

- using generic types,
- refining the Java code obtained by direct translation of OCL elements,
- offering different options for managing exceptions in case of assertion violation.

Our belief is that, sooner or later, using models and assertions in producing software will become the current practice.

References

1. Borland Together. http://www.borland.com/us/products/together/index.html
2. Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net/index.php
3. eXecutable Metamodeling Facility (XMF) Home Page, Ceteva Ltd. 2007. http://itcentre.tvu.ac.uk/~clark/xmf.html
4. Frame Based on the Extensive Use of Metamodeling for the Specification, Implementation and Validation of Languages and Applications (EMF_SIVLA). http://www.cs.ubbcluj.ro/~chiorean/CUEM_SIVLA
5. Frame Based on the Extensive Use of Metamodeling for the Specification, Implementation and Validation of Languages and Applications (EMF_SIVLA) - Project Deliverables. http://www.cs.ubbcluj.ro/~vladi/CUEM_SIVLA/deliverables/EduSymp2011/workspaces.zip
6. InvestorWords. http://www.investorwords.com/1079/contract.html
7. Markus Voelter homepage. http://voelter.de/
8. miniSPA 2005 REPORT. http://codemanship.co.uk/parlezuml/blog/?postid=34
9. MOdeling LAnguages portal. http://modeling-languages.com/blog/content/poor-validation-uml-models-eclipse-uml2-tools
10. Object Constraint Language 2.3.1. Formally published by ISO as the 2012 edition standard: ISO/IEC 19507:2012(E) 12-05-09.pdf
11. openArchitectureWare (oAW). http://www.openarchitectureware.org/
12. The OCL portal. http://st.inf.tu-dresden.de/ocl/index.php?option=com_content&view=category&id=5&Itemid=30
13. UML-based Specification Environment (USE). http://www.db.informatik.uni-bremen.de/projects/USE/
14. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
15. Ackermann, J.: Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. In: T. Baar (ed.) Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005, Technical Report LGL-REPORT-2005-001, pp. 15–29. EPFL (2005)
16. Ackermann, J.: Frequently Occurring Patterns in Behavioral Specification of Software Components. In: COEA, pp. 41–56 (2005)
17. Akehurst, D.H., Gareth, W., Howells, J., McDonald-Maier, K.D.: Implementing associations: UML 2.0 to Java 5. Software and System Modeling **6**(1), 3–35 (2007)
18. Baar, T.: Non-deterministic Constructs in OCL - What Does any() Mean. In: A. Prinz, R. Reed, J. Reed (eds.) SDL 2005: Model Driven Systems Design - Proceedings of 12th International SDL Forum, *Lecture Notes in Computer Science*, vol. 3530, pp. 32–46. Springer (2005)
19. Baar, T., Hähnle, R., Sattler, T., Schmitt, P.H.: Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In: K. Mehlhorn, G. Snelting (eds.) Informatik 2000, 30. Jahrestagung der Gesellschaft für Infomatik, pp. 389–404. Springer (2000)
20. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making Components Contract Aware. Computer **32**(7), 38–45 (1999)
21. Bezivin, J.: On the Unification Power of Models. Software and System Modeling (SoSyM) **4**(2), 171–188 (2005). http://www.sciences.univ-nantes.fr/lina/atl/www/papers/OnTheUnificationPowerOfModels.pdf
22. Bezivin, J.: Introduction to Model Engineering (2006). http://www.modelware-ist.org/index.php?option=com_remository&Itemid=74&func=fileinfo&id=72
23. Brucker, A.D., Wolff, B.: HOL-OCL. http://www.brucker.ch/projects/hol-ocl/
24. Bruel, J.M., Henderson-Sellers, B., Barbier, F., Le Parc, A., France, R.: Improving the UML Metamodel to Rigorously Specify Aggregation and Composition (2001)
25. Bunyakiati, P., Finkelstein, A.: The Compliance Testing of Software Tools with Respect to the UML Standards Specification - the ArgoUML Case Study. In: Proceedings of the Fourth International Workshop on Automation of Software Test (AST'09), pp. 138–143. IEEE CS Press (2009). http://www.cs.ucl.ac.uk/staff/A.Finkelstein/papers/icseast.pdf

26. Buttner, F., Gogolla, M., Hamann, L., Kuhlmann, M., Lindow, A.: On Better Understanding OCL Collections or An OCL Ordered Set is not an OCL Set. In: Proceedings of the 2009 international conference on Models in Software Engineering (MODELS'09), no. 6002 in LNCS, pp. 276–290. Springer-Verlag Berlin, Heidelberg (2009)
27. Cabot, J.: Common UML errors (I): Infinite recursive associations (2011). `http://modeling-languages.com/common-uml-errors-i-infinite-recursive-associations/`
28. Cheesman, J., Daniels, J.: UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley (2000)
29. Chimiak-Opoka, J., Demuth, B.: Teaching OCL Standard Library: First Part of an OCL 2.x Course. ECEASST **34** (2010)
30. Chiorean, D., Corutiu, D., Bortes, M., Chiorean, I.: Good Practices for Creating Correct, Clear and Efficient OCL Specifications. In: K. Koskimies, L. Kuzniarz, J. Lilius, I. Porres (eds.) Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language (NWUML'2004), no. 35 in TUCS General Publications, pp. 127–142. Turku Center for Computer Science (TUCS), Finland (2004)
31. Chiorean, D., Petraşcu, V., Ober, I.: Testing-Oriented Improvements of OCL Specification Patterns. In: Proceedings of the 2010 IEEE International Conference on Automation, Quality and Testing, Robotics - AQTR, vol. II, pp. 143–148. IEEE Computer Society (2010)
32. Chouali, S., Heiser, M., Souquières, J.: Proving Component Interoperability with B Refinement. Electronic Notes in Theoretical Computer Science **160**, 157–172 (2006)
33. Clark, T., Sammut, P., Willans, J.: Applied Metamodeling: A Foundation for Language Driven Development (Second Edition). Ceteva (2008)
34. Clark, T., Sammut, P., Willans, J.: Superlanguages: Developing Languages and Applications with XMF (First Edition). Ceteva (2008). `http://itcentre.tvu.ac.uk/~clark/docs/Superlanguages.pdf`
35. Cook, S., Daniels, J.: Designing object systems: object-oriented modelling with Syntropy. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1994)
36. Costal, D., Gómez, C., Queralt, A., Raventós, R., Teniente, E.: Facilitating the Definition of General Constraints in UML. In: MoDELS, pp. 260–274 (2006)
37. Crnkovic, I., Larsson, M. (eds.): Building Reliable Component-Based Software Systems. Artech House, Inc. (2002)
38. Damus, C.W.: Implementing Model Integrity in EMF with MDT OCL. Eclipse Corner Articles. Eclipse Foundation (2007). `http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html`
39. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML: the Catalysis Approach. Addison-Wesley (1999)
40. Eclipse Foundation: Eclipse Modeling Framework (EMF). `http://www.eclipse.org/modeling/emf`
41. Eclipse Foundation: Eclipse Modeling Project (EMP). `http://www.eclipse.org/modeling/`
42. Eclipse Foundation: Graphical Modeling Project (GMP). `http://www.eclipse.org/modeling/gmp/`
43. Eclipse Foundation: Model Development Tools (MDT) OCL. `http://www.eclipse.org/modeling/mdt/?project=ocl`
44. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering, FOSE '07, pp. 37–54. IEEE Computer Society, Washington, DC, USA (2007). DOI http://dx.doi.org/10.1109/FOSE.2007.14. URL `http://dx.doi.org/10.1109/FOSE.2007.14`
45. Friese, P., Kolb, B.: Validating Ecore models using oAW Workflow and OCL. In: Eclipse Summit Europe 2007 (2007)
46. Fuentes, J.M., Quintana, V., Llorens, J., Génova, G., Prieto-Díaz, R.: Errors in the UML metamodel? ACM SIGSOFT Software Engineering Notes **28**(6), 3–3 (2003)
47. Garcia, M.: Rules for Type-checking of Parametric Polymorphism in EMF Generics. In: W.G. Bleek, H. Schwentner, H. Züllighoven (eds.) Software Engineering (Workshops), *Lecture Notes in Informatics (LNI)*, vol. 106, pp. 261–270. GI (2007)
48. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification (Third Edition). Addison-Wesley Longman (2005)

49. Groher, I., Voelter, M.: XWeave: Models and Aspects in Concert. In: AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling, pp. 35–40. ACM Press (2007)
50. Hoare, C.: An Axiomatic Basis for Computer Programming. CACM **12**(10) (1969)
51. Jézéquel, J.M., Meyer, B.: Design by Contract: The Lessons of Ariane. Computer **30**, 129–130 (1997)
52. Laboratorul de Cercetare în Informatică (LCI): Object Constraint Language Environment (OCLE). `http://lci.cs.ubbcluj.ro/ocle/`
53. Martin, J., Odell, J.: Object-Oriented Methods: A Foundation. Prentice-Hall (1994)
54. Merks, E., Paternostro, M.: Modeling Generics with Ecore. In: EclipseCon 2007 (2007)
55. Meyer, B.: Design by Contract (1986). Technical Report TR-EI-12/CO, Interactive Software Engineering Inc.
56. Meyer, B.: Object-Oriented Software Construction (First Edition). Prentice Hall (1988)
57. Meyer, B.: Design by Contract. In: Advances in Object-Oriented Software Engineering, pp. 1–50. Prentice Hall (1991)
58. Meyer, B.: Applying "Design by Contract". Computer **25**(10), 40–51 (1992)
59. Meyer, B.: Object-Oriented Software Construction (Second Edition). Prentice Hall (1997)
60. Miliauskaitė, E., Nemuraitė, L.: Representation of Integrity Constraints in Conceptual Models. Information Technology and Control **34**(4), 355–365 (2005)
61. Mitchell, R., James, R.: Heuristics for Improving Precision in UML Models. BRITISH COMPUTER SOCIETY SPA Specialist Group, online at `http://www.bcs-oops.org.uk/twiki/pub/SPA/NotesFromMiniSPA2005/20050805_ImprovingUML8605B.doc`
62. Nierstrasz, O.: Synchronizing Models and Code (2011). Invited Talk at TOOLS 2011 Federated Conference, `http://toolseurope2011.lcc.uma.es/#speakers`
63. Object Management Group (OMG): Common Warehouse Metamodel (CWM), Version 1.1 (2003). `http://www.omg.org/spec/CWM/1.1`
64. Object Management Group (OMG): Model Driven Architecture (MDA) Guide, Version 1.0.1 (2003). `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`
65. Object Management Group (OMG): Unified Modeling Language (UML) Specification, Version 1.5 (2003). `http://www.omg.org/spec/UML/1.5/PDF/`
66. Object Management Group (OMG): Unified Modeling Language (UML) Specification, Version 1.4.2 (2005). `http://www.omg.org/spec/UML/ISO/19501/PDF/`
67. Object Management Group (OMG): Meta Object Facility (MOF) Core Specification, Version 2.0 (2006). `http://www.omg.org/spec/MOF/2.0/PDF`
68. Object Management Group (OMG): Object Constraint Language (OCL), Version 2.2 (2010). `http://www.omg.org/spec/OCL/2.2/PDF/`
69. Object Management Group (OMG): Unified Modeling Language (UML), Infrastructure, Version 2.3 (2010). `http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/`
70. Object Management Group (OMG): Unified Modeling Language (UML), Superstructure, Version 2.3 (2010). `http://www.omg.org/spec/UML/2.3/Superstructure/PDF/`
71. Object Management Group (OMG): Query/View/Transformation (QVT), Version 1.1 (2011). `http://www.omg.org/spec/QVT/1.1`
72. Object Management Group (OMG): XML Metadata Interchange (XMI), Version 2.4 - Beta 2 (2011). `http://www.omg.org/spec/XMI/2.4/Beta2`
73. OMG (Object Management Group): Object Constraint Language (OCL), Version 2.3 Beta 2 (2011). `http://www.omg.org/spec/OCL/2.3/Beta2/PDF`
74. Plasil, F., Visnovsky, S.: Behavior Protocols for Software Components. IEEE Transactions on Software Engineering **28**(11), 1056–1076 (2002)
75. Richters, M., Gogolla, M.: Validating UML models and OCL constraints. In: A. Evans, S. Kent, B. Selic (eds.) UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference Proceedings, *Lecture Notes in Computer Science*, vol. 1939, pp. 265–277. Springer (2000)
76. Schmidt, D.C.: Model-Driven Engineering. Computer **39**(2), 25–31 (2006)
77. Seifert, M., Samlaus, R.: Static Source Code Analysis using OCL. In: J. Cabot, M. Gogolla, P.V. Gorp (eds.) Proceedings of the 8th International Workshop on OCL Concepts and Tools (OCL 2008) at MoDELS 2008, *Electronic Communications of the EASST*, vol. 15, p. 15 pages. European Association of Software Science and Technology (EASST) (2008). `http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/174/171`

78. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (Second Edition). Addison-Wesley Professional (2008)
79. Todorova, A.: Produce more accurate domain models by using OCL constraints (2011). `https://www.ibm.com/developerworks/rational/library/accurate-domain-models-using-ocl-constraints-rational-software-architect/`
80. Venners, B.: Abstraction and Efficiency. A Conversation with Bjarne Stroustrup - Part III (2004). `http://www.artima.com/intv/abstreffi2.html`
81. Voelter, M., Kolb, B., Efftinge, S., Haase., A.: From Front End To Code - MDSD in Practice (2006). `http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html`
82. Wahler, M.: Using Patterns to Develop Consistent Design Constraints. Ph.D. thesis, ETH Zurich, Switzerland (2008). `http://e-collection.ethbib.ethz.ch/eserv/eth:30499/eth-30499-02.pdf`
83. Wahler, M., Basin, D., Brucker, A.D., Koehler, J.: Efficient Analysis of Pattern-Based Constraint Specifications. Software and Systems Modeling **9**(2), 225–255 (2010). DOI 10.1007/s10270-009-0123-6. URL `http://www.brucker.ch/bibliography/abstract/wahler.ea-efficient-2010`
84. Wahler, M., Koehler, J., Brucker, A.D.: Model-Driven Constraint Engineering. Electronic Communications of the EASST **5** (2006)
85. Warmer, J., Kleppe, A.: Object Constraint Language: Precise Modeling with UML, first edn. Addison-Wesley (1999)