Specifying consistent subsets of UML

Jean-Louis Sourrouille, Mohammed Hindawi, Lionel Morel, Régis Aubry

Specifying consistent subsets of UML

Jean-Louis Sourrouille, Mohammed Hindawi, Lionel Morel, Régis Aubry

INSA Lyon, LIESP, Bât. B. Pascal, 69621 Villeurbanne, France
{Jean-Louis.Sourrouille, Mohammed.Hindawi, Lionel.Morel, Régis.Aubry}@insa-lyon.fr

Abstract. While increasing progressively its expressive power, UML has become more and more difficult to read and understand, especially for beginners. To teach the whole UML is not possible, therefore teachers only deal with a subset of UML. We present a framework for defining precisely a consistent subset of a language, allowing everybody to define his/her own subset. An extended example shows a way to simplify UML sequence diagrams. Our approach use standard UML tools; models in the simplified language are fully compatible with UML; model mapping between metamodels is done automatically from specifications based on marking with stereotypes; constraints are automatically translated from the simplified language to UML.

1. Introduction

Since its first version, UML [5] has evolved to increase its expressive power, its coherence and more generally the quality of its description, i.e., its metamodel. In return, especially from UML 2, this metamodel has become larger, more complex and more difficult to read. But who needs to read the UML metamodel?

Students that rely on courses, books, and a tool that enforces the language syntax do not apparently need to read the metamodel: they gain enough knowledge from this material to deal with exercises that teachers give. Thus, they build their own view of the structure and semantics of UML. To go further, for instance to lead a team using intensively models, to set model element properties for code generators, to understand language extensions, or very simply to find why expressions are rejected by tools, they need a reference. Programming languages' reference lies on a bible (e.g., [4] for C++), and their compilers ensure syntactic correctness. Quite differently, UML tools enforce an incomplete non-unique syntax, and the only reference is a metamodel [5]: to gain enough knowledge to read this metamodel is a heavy investment. As a result, practitioners do not refer to the metamodel and rarely well-know UML. On the other hand, some tasks require UML experts, for instance specifying a style guide as restrictions to the language, which requires adding OCL constraints to the metamodel, or specifying a Domain Specific Language.

There is a gap between UML experts that know the metamodel and UML users that do not. To broaden a better UML knowledge beyond the group of UML experts, we advocate that the metamodel should be made understandable by users according to their UML level. Since the UML metamodel is too complex, we propose a framework to define a simplified UML. This language is intended first for educational purposes:

teachers and students could refer to the metamodel during UML courses. This simplified language should be an accessible yet coherent subset of UML, compatible with the standard. It should allow advanced users to write profiles and constraints.

The goal of this paper is neither to discuss what should be taught about UML nor to define *a simplified UML*, but to describe an approach for simplifying the UML metamodel. Next, *the* suitable language will depend on specific needs: teachers will typically limit the metamodel to the subset of notions used in their course. While the objective is the use of a simplified language in education, the paper mainly deals with the technical solution we propose to define such a language. The rest of the paper is organized as follows: section 2 describes the motivations; section 3 gives the principles of the proposed approach; section 4 shows an extended example; section 5 deals with implementation issues before the conclusion.

2. Motivations and Related Works

The UML metamodel [5] was designed for allowing an easy description of the language (abstraction of concepts, systematic organization etc.), but not for easing its reading. Readers, in particular beginners, encounter difficulties due to various reasons: they should be familiar with UML to read the metamodel; understanding the meaning of the notions and verifying what is authorized/ forbidden is a long way from the leaves of the metamodel to their ancestors and/or previous definitions; there are a lot of abstract classes; redefinitions of relations and names introduce redundancy; information is split into numerous small pieces with no synthetic views.

How to teach the UML in this context? Most teachers describe a subset of UML notions, their semantics and the induced constraints without referring to the metamodel. The semantics defines the limits for expression correctness. These limits are fuzzy because without a more precise support than natural language, it is not possible to explain completely what is correct and what is not. Taught notions have generally a graphical representation and mostly correspond to leaves in the inheritance hierarchy of the metamodel. Links between notions are buried in the metamodel and are more difficult to explain.

When modeling manually, students generally use a rough syntax, models are rather informal and the interpretation is sometimes impossible. Obviously, syntactic checks are necessary, and manual modeling is to be reserved for small exercises. When modeling with a tool, the syntactic correctness is better but it depends on the tool. The model part used to generate the code is generally complete since a model analyzer detects and even corrects errors. The rest of the model is rarely complete and consistent, and links between models elements are vague. As long as the code provides expected results, most students do not attach importance to models.

However, software development more and more focuses on models. Verifications from the very beginning of the development will allow finding errors earlier. Hence, students will have in the future to provide precise models with the wished properties, and enhanced tools will check a kind of style guide. In the context of model engineering, developers have to know UML more precisely, and to understand rules often requires knowing corresponding language notions. Again, the UML metamodel

complexity is a barrier. To define a simplified metamodel for a subset of UML has some significant advantages:

- To provide a precise description that will be a reference for students and teachers,
- To delimit the subset that students should know,
- To ensure that the taught UML subset is consistent,
- To allow defining small language extensions and checking style rules,
- To prepare students to deepening of one's knowledge in the context of MDE,
- To increase the credibility of UML as a language.

The last point might be the most important: as long as a language has no formal description with a precise meaning, students do not use it as a language but rather as an alternative to natural language with boxes, lines and a vague meaning. In our opinion, the first condition for UML to have the status of a language is to provide the usual elements of a language, and first of all a formal description of the syntax that can be used by students. Of course, a checker is a second condition and we pursue parallel works about the implementation of a style guide.

The main drawback of this proposal is the need for a complicated tool. To define a UML subset could also be long, but at the same time, it is interesting for a teacher.

How to simplify UML while conforming to UML? Anyway, UML is to be taken as it is. To tackle this issue we propose two complementary directions:

- Many teachers start their course saying that will not describe the entire UML. They choose a subset of notions according to their own criteria: representative notions, need for a lab work, etc. The first direction to simplify UML is thus to reduce the language, hence its power of expression, by canceling notions.
- For a given language, there are an infinite number of possible descriptions. The second direction is to choose a language description, i.e., a metamodel, that is equivalent to the original metamodel but has better properties such as easier to read. One can suppress abstract classes, reorganize the hierarchy, etc. These two approaches are complementary since canceling notions facilitates reorganization.

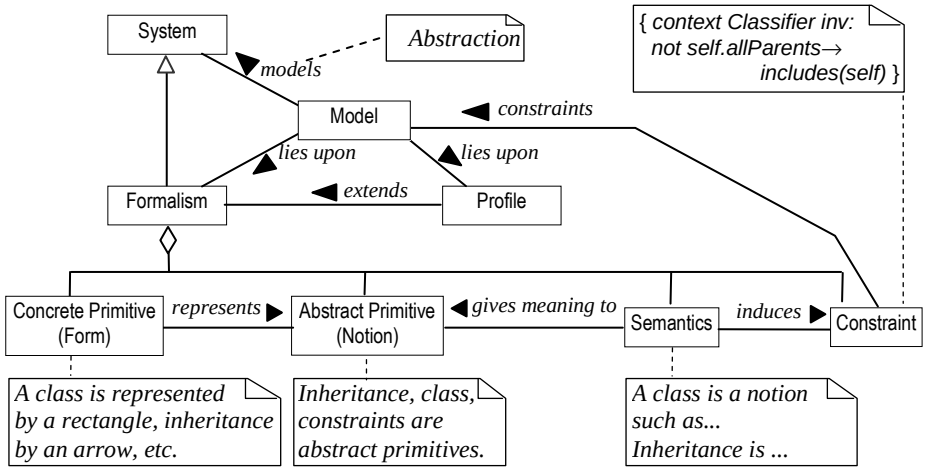


Fig. 1. A basic model of System, Language and Model in UML.

Related works. Several works confirm that in both industry and academia, UML teachers deal with only a subset of UML due to language complexity and an overwhelming number of notions [3]. These UML subsets are selected based on diagrams and their associated constructs, for instance *UseCase/Class/Activity/ Sequence* in [6], *UseCase/Class/Sequence* in [3]. The expressive power is dramatically reduced when dropping diagrams. A possible consequence could be the lower use of some point of views and finally a misuse of UML [2]. Even within a single diagram such as the sequence diagram, the complexity remains high. At least for teaching, to select purposefully a set of consistent notions in the whole UML seems a better compromise still to cover the whole modeling domain.

3. Principle of the Proposed Approach

3.1. Definitions and Issues

The model in Fig. 1 makes explicit our definition of formalism **[**1**]**. A *Model* is a representation of a system expressed in a given *Formalism* or language. A metamodel is a model of a formalism viewed as a *System* (see *definition* below). The *Semantics* describes the meaning of the notions, i.e., the *Abstract Primitives* of the formalism. This semantics induces *Constraints* on models. Finally, the classes of a metamodel are called metaclasses, and a model is a set of instances of metaclasses.

Notations. $m(s)/f$ denotes a model m of the system s in the formalism f, and to avoid confusion we write $mm(f)/F$ to denote a metamodel of f expressed in F, i.e., a model mm of the formalism f, viewed as a system, expressed in the formalism F. According to this notation, $M_{UML} = mm(UML)/UML$ is the UML metamodel. A transformation of x into y is written $x \rightarrow y$. A model mapping is a transformation $m_1(s)/f_1 \rightarrow m_2(s)/f_2$. A profile $M_p = m(P)/L$ is a model of a language fragment P, i.e., a metamodel, directly usable in a model: $m(s)/(L+P)$.

Definition. A metamodel is a tuple $M=\{f_M,n_M,s_M,c_M\}$ where f_M is a set of forms, i.e., concrete graphical or textual primitives; n_M is a set of notions, i.e., abstract primitives; s_M is the semantics of the described language; c_M is a set of constraints on models.

Issue 1 – Ensuring backward compatibility. Our objective is to define a simplified version L_S of a language L by withdrawing notions. From the root language UML, this process can be repeated. The metamodel of L is $M = mm(L)/UML$. The language L_S is defined by $M_S = mm_S(L_S)/UML$ such that the transformation $m_S(s)/L_S \rightarrow m(s)/L$ of any model in L_S be loss-less. Since M_S is a subset of M, this model mapping is theoretically loss-less. Practically, we have to provide a fully automatable process.

Issue 2 – Automatic constraint translation. Reducing the language described by $M = \{f_M,n_M,s_M,c_M\}$ by canceling notions amounts to defining $M_S = \{f_{Ms},n_{Ms},s_{Ms},c_{Ms}\}$ such that $n_{Ms} \subseteq n_M$. As a possible consequence of this transformation, some forms may become useless in M_S, hence $f_{Ms} \subseteq f_M$. The transformation $s_M \rightarrow s_{Ms}$ is an adaptation of the english natural text that describes the semantics. The last

transformation $c_M \rightarrow c_{Ms}$ deals with constraints. An automatic transformation would be nice but in practice, it depends on the nature of the simplifications. Fortunately, a few constraints will generally be translated.

Issue 3 – Automatic profile transformation. To translate into L_S a profile P expressed in L boils down to the above *issue* 2. A profile $M_{Ps} = m_S(P)/L_S$ expressed in L_S requires the transformation $m_S(P)/L_S \rightarrow m(P)/L$. This transformation is similar to *issue* 1 but differs when a profile explicitly uses notions of M_S within OCL expressions, which complicates automatic translation.

Issue 4 – Metamodel Equivalence. The other approach we consider earlier aims to change the metamodel without changing the language, that is defining $M_1 = mm_1(L)/UML$ from $M_0 = mm_0(L)/UML$, where M_1 and M_0 are two different metamodels of the same formalism L viewed as a system. A model is composed of instances of metaclasses, hence modifying the metamodel implies model changes. Although this transformation $m_1(s)/L_{M1} \rightarrow m_0(s)/L_{M0}$ seams to be similar to *issue* 1, it is different since M_1 is not a subset of M_0. Currently, there is no proof that this transformation is always possible, therefore among all potential metamodel transformations, only the ones allowing automatic model transformation will be to consider. Finally, constraint translation from L_{M1} to L_{M0} is part of *issue* 3.

3.2. Principle Illustration

This section introduces an example to lighten the solutions to the previous points. The basic principle of the solution is as follows: to avoid a risky transformation (*issues* 1-4), it is sufficient to represent a model in original UML. This is always possible when notions of the simplified language belong to the UML. When a metamodel has been exchanged for an equivalent one (*issue* 4), models representations and visible properties should be the same, but expressions using the metamodel such as OCL constraints are to be translated (*issue* 3). Except for profiles to be translated, this solution ensures full compatibility with UML and allows using standard tools.

As a second principle, automatic processing is more important than simplifications: simplifications that cannot be processed either automatically or manually with a low effort (*issue* 2) are to be canceled. Not to disturb or burden the modeler is a necessary condition for users to support this approach. For a modeler using the simplified language, apart from the metamodel that describes the simplified language, the only visible differences are limitations of language, and possibly explicit references to the simplified metamodel in OCL constraints.

The example Fig. 2 illustrates the simplification process mixing metamodel equivalence and withdrawal of notions. Fig. 2a gives the metamodel to be simplified. Fig. 2c shows a model and the *instanceOf* relationship between elements of the model and their metaclass. Let us assume that we want to remove notion E and to fix the multiplicity of d to exactly 1. Due to their association (1,1), C and D are indivisible. Since B is an abstract class, the simplified metamodel (Fig. 2b) can be reduced to notions A and $F = B \cup C \cup D$. We finally suppose that the attribute a_B is to withdraw.

Compared with a model in the full language, a model in the simplified language (Fig. 2d) does not allow creating instances of the metaclass E (canceled) nor instances

of *C* unrelated to *D* (multiplicity 1). The Fig. 2d shows the relationship *instanceOf* for the simplified metamodel. This relationship corresponds to what the modeler think, but model elements are in fact instances of the full metamodel (dark lines Fig. 2c).

OCL expressions in the simplified language, e.g., *bd->size*(), should be translated into UML (*issue* 3). The comment Fig. 2b gives a list of rewriting rules in the context of *A*. The translation of *bd*[*i*].*a_D* is *b*[*i*].*d*.*a_D* while *bd->size*() becomes *b*->size(). In the context *F*, assuming that instances of *F* become instances of *C*, *a_C* is unchanged, but *a_D* becomes *d*.*a_D*. The translation of expressions related to the merging $C \cup D$ is more complicated, for instance: *bd->collect*(*x* | ... *x*.*a_C* ... *x*.*a_D* ...) becomes after translation *b*.*collect*(*x* | ... *x*.*a_C* ... *x*.*d*.*a_D* ...).

From this example, the requirements become clearer. Automating the translation of OCL expressions will be the most difficult task (*issue* 3). The proposed approach is as follows: (i) to aid the construction of the simplified metamodel by a tool supplying model-to-model transformations and ensuring consistency; (ii) to save during this step a list of rewriting rules; (iii) as far as possible, to do automatically the translation of UML semantic constraints into the simplified metamodel, otherwise a manual translation is required and traces should be kept (*issue* 2); (iv) to translate automatically OCL constraints expressed in simplified UML into UML using rewriting rules (*issue* 3). When this translation proves to be impossible or too intricate, restrictions apply to OCL expressions.

4. Example Using the UML Metamodel

Let us assume that we need a simplified metamodel of the UML sequence diagram. We separate general and detailed notions to ease their representation (the translation

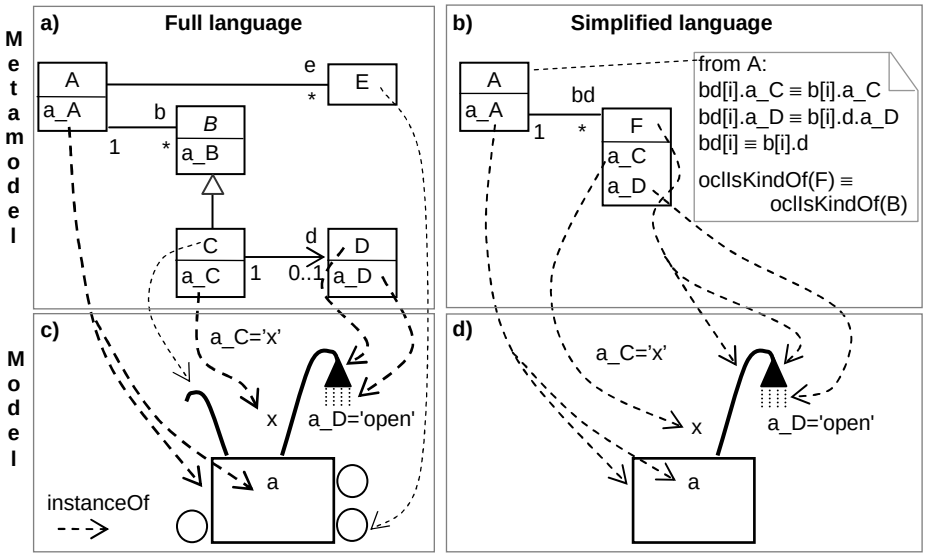


Fig. 2. Two models from two metamodels

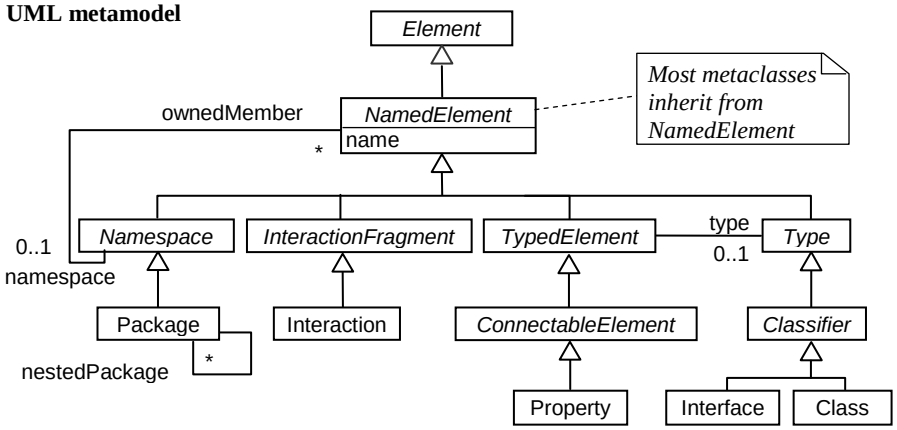
of a profile aiming to enforce a style guide is given in the extended version [7]).

4.1. General Notions

There is no metaclass for diagrams in UML. A sequence diagram is represented using *Package*s that describe *Interaction*s between *ConnectableElement*s whose *Type* may be a *Class*. Fig. 3a shows the corresponding UML metamodel extract.

Simplified Metamodel. The minimum set of required notions includes the leaves plus *Type* to factorize its association. Such a reduction would be excessively strong because the notions of *NameSpace* and *nestedPackage* are easy to understand, hence a better compromise with an equivalent complexity is given Fig. 3b.
To keep only Class and Interface as Types is a minor restriction, adding Component will be easy. Property, which "represents a set of instances that are owned by a containing classifier instance", is not the only ConnectableElement but it is enough to begin. InteractionFragment is needed for the detailed description of the sequence diagram below. The word Property has many meanings and is confusing, therefore renaming the class could have been considered. In this part of the UML metamodel, no OCL semantic constraint is to translate into the simplified metamodel (issue 2).

a) UML metamodel



b) Simplified UML metamodel

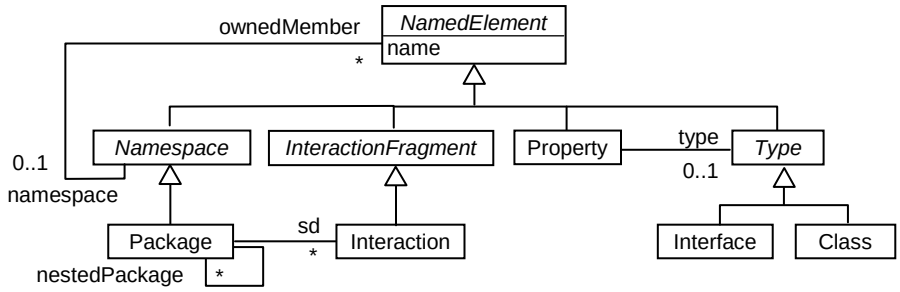


Fig. 3. UML (a) and simplified UML (b) metamodels

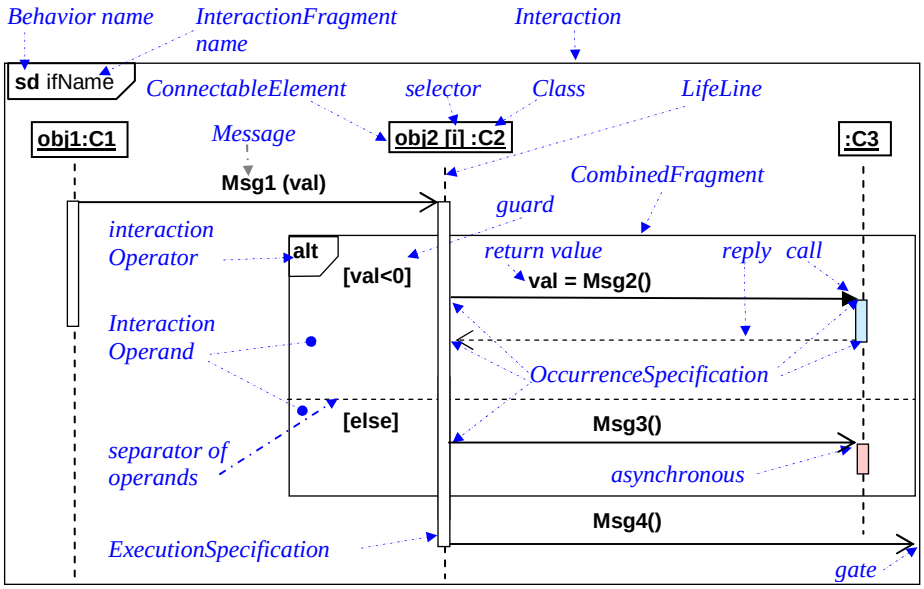


Fig. 4. Sequence diagram main notions.

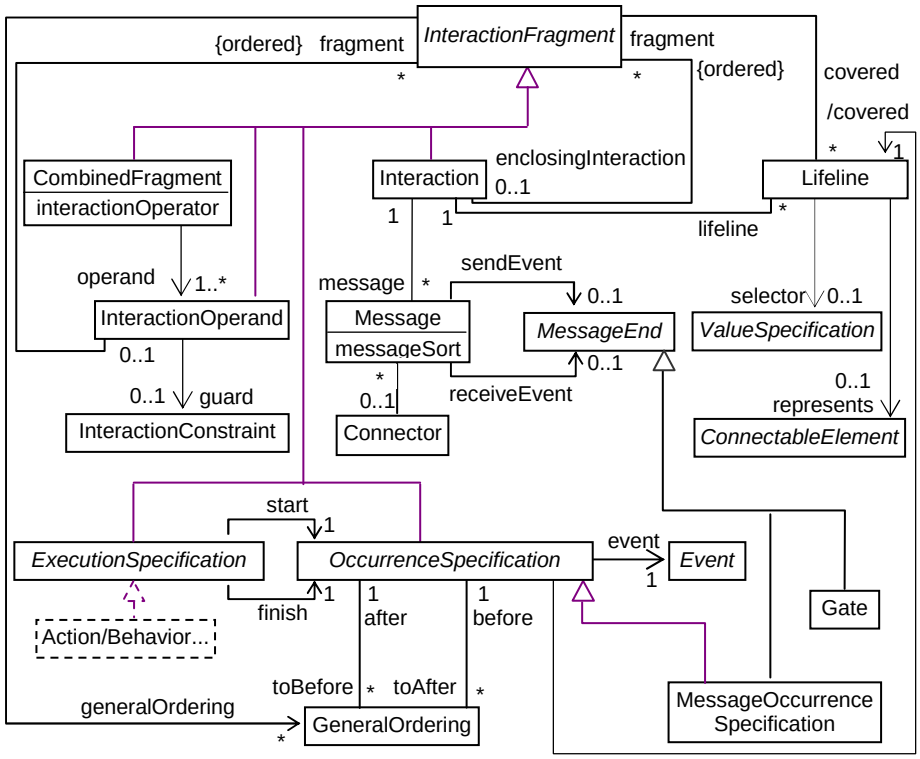


Fig. 5. Part of the Interaction Metamodel

Profile transformation. Translating OCL expressions from the simplified metamodel to UML requires only rewriting paths (*issue* 3). An interesting point in Fig. **3** is the translation of the association ***sd*** from ***Package*** to ***Interaction*** that provides directly all the ***Package ownedMembers*** that are ***Interaction*****s** (or inherit from):

```
Package::sd : Set(Interaction) ;
        sd=self.ownedMember->select( i | i.oclIsKindOf(Interaction) )
```

A *NamedElement* (*self*) contains a sequence diagram when:

```
        self.oclIsKindOf(Package) and self.sd->notEmpty()
```

To search all the packages that contain a sequence diagram:

```
NamedElement::allSdPackages(): Set(Package) ;
        allSdPackages=NamedElement.allInstances->select( p |
                                p.oclIsKindOf(Package) and p.sd->notEmpty() )
```

4.2. Detailed Description of Interaction Notions

The minimal set of notions suitable for a beginner to describe objects' interactions is obviously subjective. The Fig. 4 shows a representative selection of notions. A *LifeLine represents* a *ConnectableElement*. *Interaction*s are described within fragments, from which *CombinedFragment*s that precise *interactionOperator*s and *InteractionOperand*s. *OccurrenceSpecification*s ordering specify the arrival of *Event*s that trigger *Message*s exchanged between *ConnectableElement*s.

Metamodel. The relevant metamodel is rather voluminous but Fig. 5 shows a representative extract in which leaves that are still abstract classes could be made explicit (*Event*, *ValueSpecification*, *ExecutionSpecification* for execution within the lifeline). The use of interactions (*InteractionUse*) misses as well as relationships with other classes (*Connector* ends with *ConnectableElement*s, *Interaction* inherits from *Behavior*), but these notions do not introduce new description issues.
How difficult is it for beginners to read this metamodel? Beginners know at least intuitively the design pattern *Composite* that could apply to the root of interactions: an interaction is either a simple *Interaction* (*Message* exchange or *Interaction* use) or an interaction enclosing an ordered collection of interactions. But the UML metamodel is more complex since *InteractionFragment* has numerous descendants such as *InteractionOperand* whose obviousness does not appear immediately, and even worst *OccurrenceSpecification* that is both an interaction and the end of an interaction. The consistency of this description escapes occasional readers. About abstraction level, a beginner would focus on message exchange while ignoring details in descendants, but in this description the class *MessageOccurrenceSpecification* is central because it specifies the sender, via *OccurrenceSpecification–Lifeline–ConnectableElement*, the receiver, and the order of events and messages.

Simplified Metamodel (Fig. 6). In the general part, we have only removed abstract classes and added an association to simplify the metamodel. This case is harder because the original metamodel is complex. To simplify, we can translate associations into attributes and use *AssociationClass*es. The expressive power of the description of

the *OccurrenceSpecification* order goes beyond the needs of a beginner. The proposed solution, which lies on the design pattern *Composite*, is enough to express a *weak* sequencing. The metamodel owns a bi-directional association from *InteractionFragment* to a descendant *Interaction*. Besides the cyclic dependency with the descendant, this kind of description prevents the reader from reading at one level of abstraction since descendants explain their ancestors. This association can be replaced by an association (*enclosingFragment*) subjected to a constraint. Finally, the metamodel supplies degrees of freedom that are not useful for a beginner, for instance a *Lifeline* can be associated with zero *ConnectableElement*, which is generally not desirable since the lifeline is no longer linked with the rest of the model.

Translation of UML Semantic Constraints (*issue* 2). Since most constraints are unchanged, the only remaining work is to change the original document. In the simplified metamodel, very specific constraints can be ignored when they have no practical existence, for instance (from *Lifeline*) "*The classifier containing the referenced ConnectableElement must be the same classifier, or an ancestor, of the classifier that contains the interaction enclosing this lifeline*". *Interaction* inherits from *Behavior* (that inherits from *Class* and *Class* from *Classifier*), and a *Behavior* is owned by a *BehaviorClassifier* (that inherits from *Classifier*). Practically, a sequence diagram is not enclosed into a *Classifier* because it would require any set of *ConnectableElement*s contained by an interaction to have a common ancestor.

Most constraints expressed in natural Language such as "*If the*

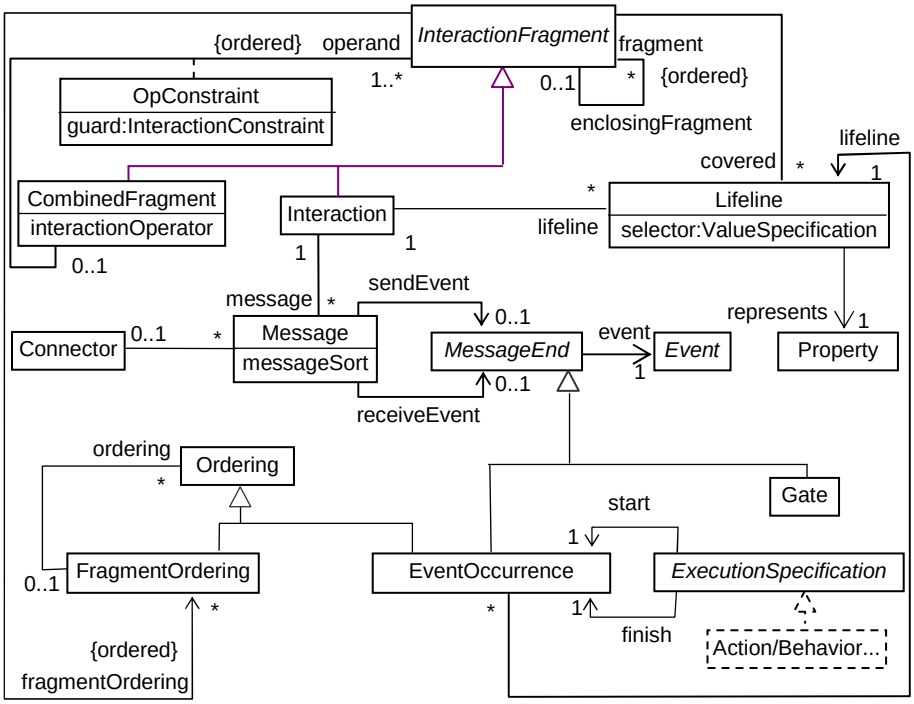


Fig. 6. Simplified Interaction Metamodel

interactionOperator is opt, loop, break, or neg, there must be exactly one operand" (from *CombinedFragment*) are unchanged. Constraints related to removed notions such as "*The interaction operators 'consider' and 'ignore' can only be used for the CombineIgnoreFragment...*" (from *CombinedFragment*) should be removed. Since *EventOccurrence* replaces *OccurrenceSpecification*, texts are to change accordingly as in "*The guard must be placed directly prior to (above) the OccurrenceSpecification that will become the first OccurrenceSpecification within this InteractionOperand*". Obviously, constraints still apply when an association such as *selector* turns into an attribute (from *Lifeline*): "*The selector for a Lifeline must only be specified if the referenced Part is multivalued.*

(self.selector->isEmpty() **implies not** self.represents.isMultivalued()) **or**
(**not** self.selector->isEmpty() **implies** self.represents.isMultivalued()) "

The new metaclass *EventOccurrence* gather the constraints of the metaclasses *OccurrenceSpecification* and *MessageOccurrenceSpecification* that it replaces (no constraint in fact). Due to this replacement, the path from *ExecutionSpecification* to *Lifeline* becomes *start.lifeline* (replaces *start.covered*) and the constraint becomes:

start.lifeline = finish.lifeline.

Surprisingly, no other constraint applies to the chosen subset of the metamodel, which is not exactly a toy example. As a result, the work to translate UML semantics constraints might not be as high as expected.

5. Implementation

The model of the simplified language L_S results from transformation of the model of the original language L. The model mapping $M = mm(L)/UML \rightarrow Ms = mm_s(L_S)/UML$ requires semantic decisions. On the other hand, the transformation actions should be kept to allow replay them when L is changed or to modify L_s. A good scenario could be: the language designer specifies basic transformations, such as abstract class removing, as well as the corresponding treatments; the model mapping is decomposed into a sequence of predefined basic transformations. To implement this scenario requires defining these basic transformations and their treatments, to provide a mean to specify transformations, and then to execute automatically these transformations.

5.1. Basic Transformations

Basic transformations aim to simplify the language by modifying its representation or by restricting the vocabulary, hence its expressive power, for instance:

- to *Remove*/*Replace* a Class,
- to *Remove* an Association, *Replace* it with an Attribute, *Move* it to another Class,
- to *Add* an AssociationClass to remove a Class that holds attributes only,
- to *Replace* an Association with an OCL operation (example Fig. 3),
- to transform a model part using a design pattern,
- to change multiplicities to forbid expressions (e.g., an object should have a Class).

5.2. Transformations

Stereotypes provide a simple way to specify transformations by marking model elements. To define a stereotype for each basic transformation is not required because treatments depend on marked elements: «Remove» is interpreted in a different way by *Class* and *Association*. Moreover, marking elements by several stereotypes provides a composition operator. The Fig. 7 shows an example of marking to transform the UML metamodel into a simplified one. Each stereotype may specify properties using tagged values, for instance «Move» specifies the new target of an association (not shown).

Finally, a document explains the model mapping. For each type of element and for each stereotype, the designer needs to know the associated transformation rules. Let *C* inherits from *B* and *B* inherits from *A*. To remove *B* requires copying all the inheritable elements of *B* into *C* and then to set *A* as the ancestor of *C*. Moreover, all the properties of type *B*, in particular class attributes, should be removed. Thus, we have to explain the multiple consequences of each basic transformation.

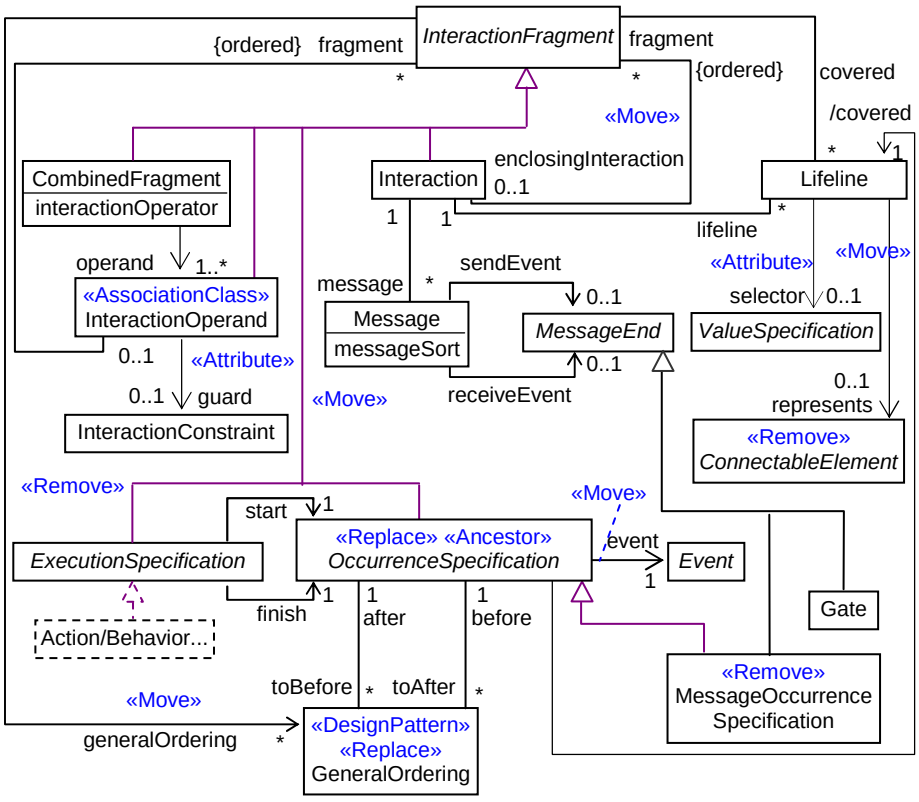


Fig. 7. Marked metamodel for automatic transformation

6. Conclusion

Past efforts to increase the expressive power of UML and to rationalize its description (evolution from UML 1. to UML 2.) did not lead to an easy to learn language. We intuitively feel that the more the language becomes powerful, the more beginners will find it difficult to learn it, particularly students. As a complementary observation, developers do not use the whole UML, no course deals with the entire UML, and beginners start with a small subset of UML. On the other hand, we want to keep the UML as the reference.

From this statement, we propose an approach to define a simplified UML, fully compatible with UML, aiming to ease the use of UML for beginners but also for non-expert users. The main issue was to automate the translations from and to UML, since beginners should not have to cope with translations. To avoid risky translations, we represent the models in UML even when expressed in simplified UML. As an additional advantage, market tools still apply just as with original UML. The only remaining translations are semantic constraints of the UML metamodel into simplified UML, and profiles expressed in simplified UML into UML, mainly OCL constraints. The former are done once by the language designer and may be exceptionally translated manually. The latter are processed automatically from rewriting rules saved during the model mapping from UML to simplified UML. From specifications of transformations based on marking with stereotypes, this model mapping is done automatically, allowing replay when changes are required in one or both languages.

We have not still completed the development of a tool, in particular the translation of profiles into UML is not achieved. Several future works could complete and improve this approach: theoretical issues such as to find properties of metamodels of the same language; technical issues such as automatic generation during model mapping of constraints aiming to limit tools to the simplified language, or even systematic transformation of metamodel using design patterns or templates.

7. References

1. G. Caplat, J.L. Sourrouille, "MDA: Model Mapping using Formalism Extension", *IEEE Software*, Vol. 22(2), pp.44-51, 2005
2. Brian Dobing , Jeffrey Parsons, How UML is used, Communications of the ACM, v.49 n.5, p.109-113, 2006
3. Erickson, J., Siau, K.: Can UML Be Simplified? Practitioner Use of UML in Separate Domains. In: EMMSAD 2007. pp. 89–98, 2007
4. B. Stroustrup: The C++ programming language, 3rd Edition, Addison-Wesley
5. UML, "OMG Unified Modeling Language", Version 2.1.2, 2007
6. S. Wrycza, B. Marcinkowski, "A Light Version of UML 2: Survey And Outcomes", Proc. Computer Science and IT Education Conference, pp.739-749, 2007
7. J-L. Sourrouille, M. Hindawi, L. Morel, R. Aubry: An approach to simplify UML – Ext. Version, http://www.if.insa-lyon.fr/liesp/~sou/Reports/sUML-RR2008_1-V1.pdf, 2008